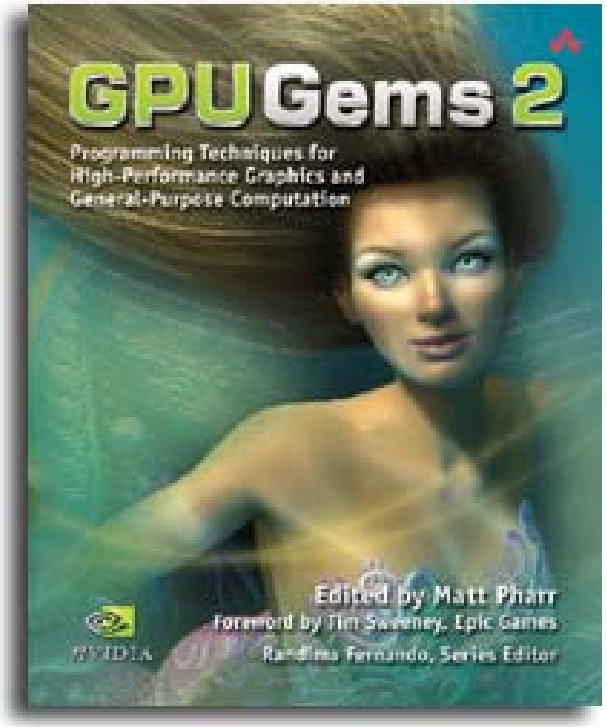
GPGPU: General-Purpose Computation on GPUs

Mark Harris

NVIDIA Corporation

GPU Gems 2

Programming Techniques for High-Performance Graphics and General-Purpose Computation



- **880 full-color pages, 330 figures, hard cover**
- **$59.99**
- **Experts from universities and industry**

18 GPGPU Chapters!

"The topics covered in *GPU Gems 2* are critical to the next generation of game engines."

— Gary McTaggart, Software Engineer at Valve, Creators of Half-Life and Counter-Strike

"*GPU Gems 2* isn't meant to simply adorn your bookshelf—it's required reading for anyone trying to keep pace with the rapid evolution of programmable graphics. If you're serious about graphics, this book will take you to the edge of what the GPU can do."

—Rémi Arnaud, Graphics Architect at Sony Computer Entertainment

Why GPGPU?

- The GPU has evolved into an extremely flexible and powerful processor
  - Programmability
  - Precision
  - Performance
- This talk addresses the basics of harnessing the GPU for general-purpose computation

Motivation: Computational Power

- GPUs are fast...
  - 3 GHz Pentium 4 *theoretical*: 12 GFLOPS
    - 5.96 GB/sec peak memory bandwidth
  - GeForce FX 5900 *observed**: 40 GFLOPs
    - 25.6 GB/sec peak memory bandwidth
  - GeForce 6800 Ultra *observed**: 53 GFLOPs
    - 35.2 GB/sec peak memory bandwidth

*Observed on a synthetic benchmark:

- A long pixel shader of nothing but MAD instructions

GPU: high performance growth

- CPU
  - Annual growth ~1.5× → decade growth ~ 60×
  - Moore's law
- GPU
  - Annual growth > 2.0× → decade growth > 1000×
  - Much faster than Moore's law

Why are GPUs getting faster so fast?

- **Computational intensity**
  - Specialized nature of GPUs makes it easier to use additional transistors for computation not cache
- **Economics**
  - Multi-billion dollar video game market is a pressure cooker that drives innovation

Motivation: Flexible and precise

- **Modern GPUs are programmable**
  - **Programmable pixel and vertex engines**
  - **High-level language support**
- **Modern GPUs support high precision**
  - **32-bit floating point throughout the pipeline**
  - **High enough for many (not all) applications**

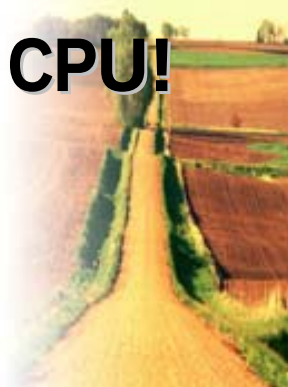
Motivation: The Potential of GPGPU

- **The performance and flexibility of GPUs makes them an attractive platform for general-purpose computation**

- **Example applications (from GPGPU.org)**
  - **Advanced Rendering: Global Illumination, Image-based Modeling**
  - **Computational Geometry**
  - **Computer Vision**
  - **Image And Volume Processing**
  - **Scientific Computing: physically-based simulation, linear system solution, PDEs**
  - **Database queries**
  - **Monte Carlo Methods**

The Problem: Difficult To Use

- GPUs are designed for and driven by graphics
  - Programming model is unusual & tied to graphics
  - Programming environment is tightly constrained
- Underlying architectures are:
  - Inherently parallel
  - Rapidly evolving (even in basic feature set!)
  - Largely secret
- Can't simply "port" code written for the CPU!



Mapping Computation to GPUs

- Remainder of the Talk:

- Data Parallelism and Stream Processing
- GPGPU Basics
- Example: N-body simulation
- Flow Control Techniques
- More Examples and Future Directions

Importance of Data Parallelism

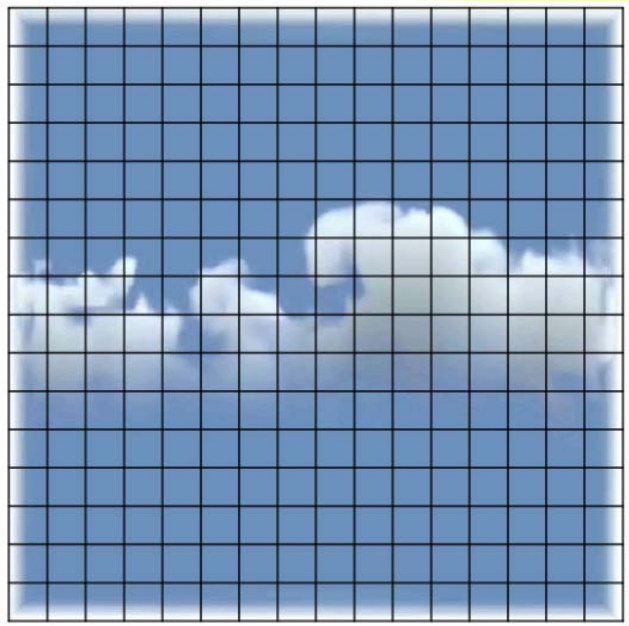
- GPUs are designed for graphics
  - Highly parallel tasks
- Process *independent* verts & fragments
  - No shared or static data
  - No read-modify-write buffers
- Data-parallel processing
  - GPU architecture is ALU-heavy
  - Performance depends on *arithmetic intensity*
    - Computation / Bandwidth ratio
  - Hide memory latency with more computation

Data Streams & Kernels

- **Streams**
  - **Collection of records requiring similar computation**
    - **Vertex positions, Voxels, FEM cells, etc.**
  - **Provide data parallelism**
- **Kernels**
  - **Functions applied to each element in stream**
    - **transforms, PDE, …**
  - **Few dependencies between stream elements encourage high Arithmetic Intensity**

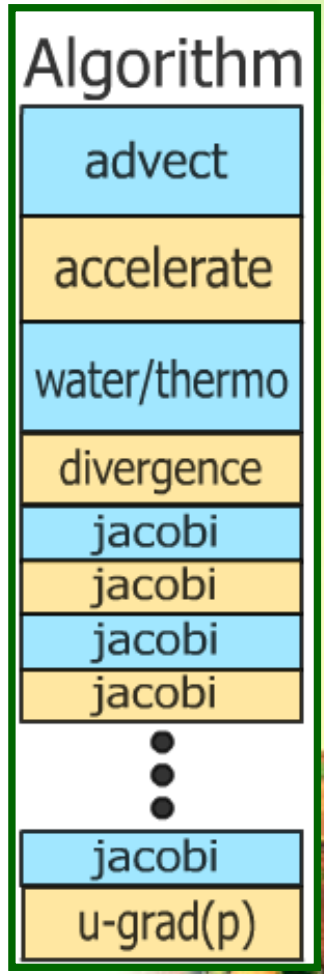
Example: Simulation Grid

- **Common GPGPU computation style**
  - Textures represent computational grids = streams
- **Many computations map to grids**
  - Matrix algebra
  - Image & Volume processing
  - Physical simulation
  - Global Illumination
    - ray tracing, photon mapping, radiosity
- **Non-grid streams can be mapped to grids**



Stream Computation

- **Grid Simulation algorithm**
  - **Made up of steps**
  - **Each step updates entire grid**
  - **Must complete before next step can begin**
- **Grid is a stream, steps are kernels**
  - **Kernel applied to each stream element**



The Basics: GPGPU Analogies

- Textures = Arrays = Data Streams
- Fragment Programs = Kernels
  - Inner loops over arrays
- Render to Texture = Feedback
- Rasterization = Kernel Invocation
- Texture Coordinates = Computational Domain
- Vertex Coordinates = Computational Range

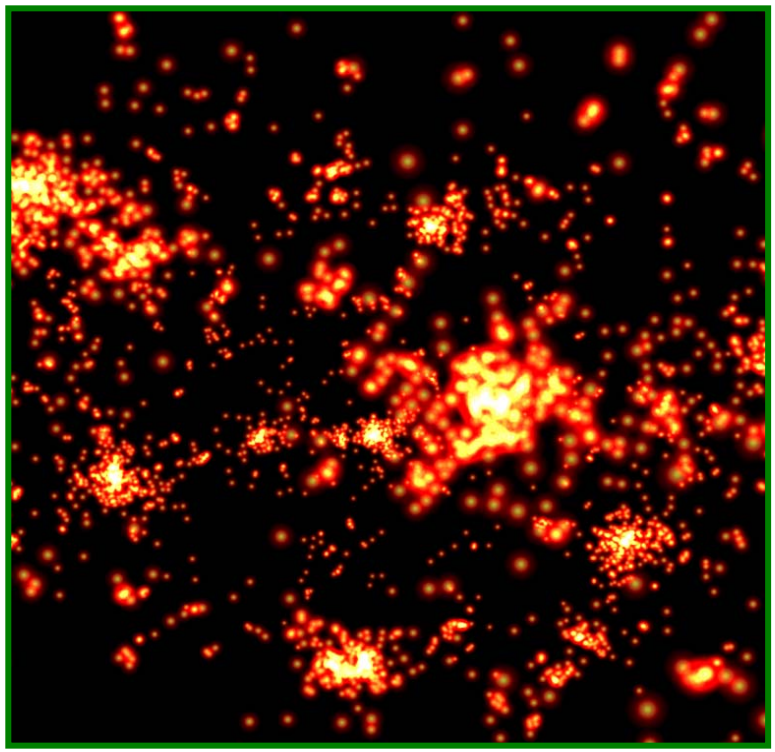
Standard "Grid" Computation

- **Initialize "view" (pixels:texels::1:1)**

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 1, 0, 1, 0, 1);
glViewport(0, 0, gridResX, gridResY);
```

- **For each algorithm step:**
  - **Activate render-to-texture**
  - **Setup input textures, fragment program**
  - **Draw a full-screen quad (1 unit x 1 unit)**

Example: N-Body Simulation



Nyland et al., GP² poster

- Brute force ☹
- N = 8192 bodies
- N^2 gravity computations

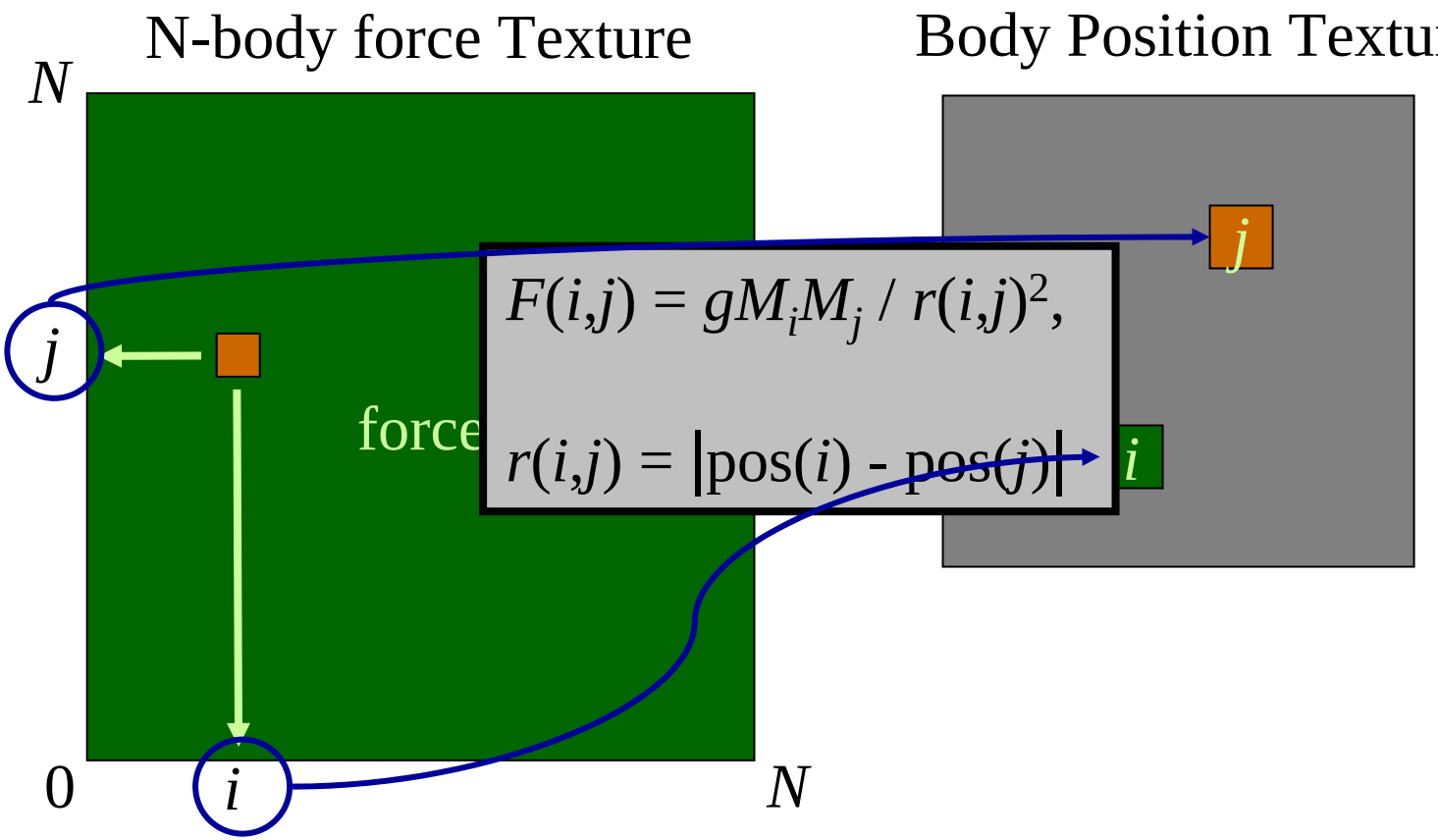
- 64M force comps. / frame
- ~25 flops per force
- 7.5 fps

- 12.5+ GFLOPs sustained
  - GeForce 6800 Ultra

Computing Gravitational Forces

- **Each particle attracts all other particles**
  - ***N* particles, so N^2 forces**
- **Draw into an *N*x*N* buffer**
  - **Fragment (*i*,*j*) computes force between particles *i* and *j***
  - **Very simple fragment program**
    - **More than 2048 particles makes it trickier**
    - **Limited by max pbuffer size…**
    - **“Left as an exercise for the reader”**

Computing Gravitational Forces



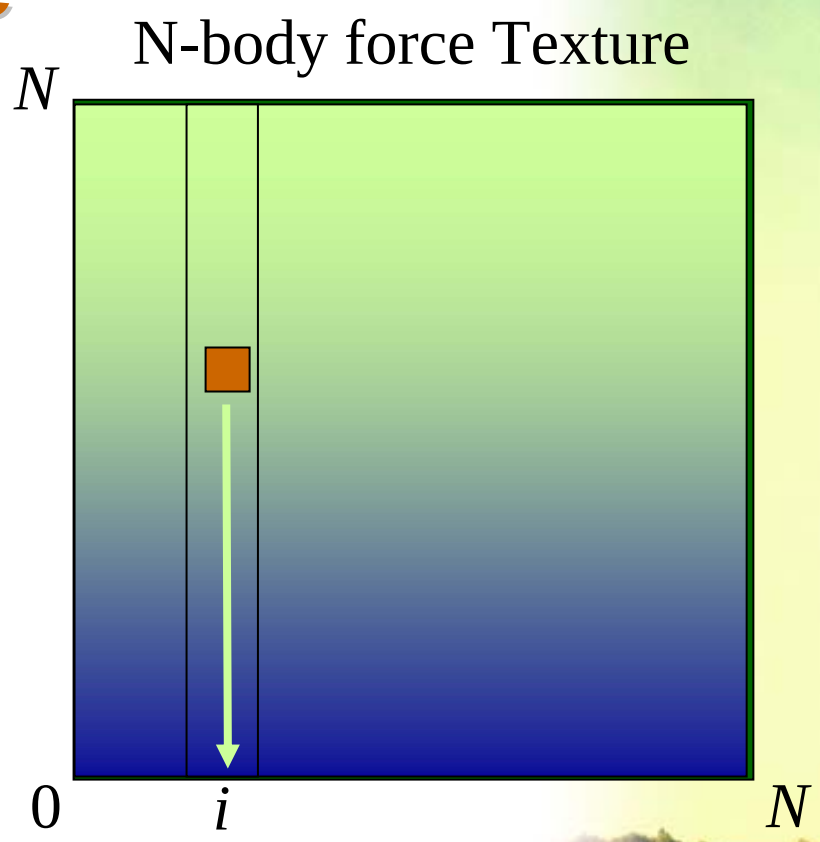
Force is proportional to the inverse square of the distance between bodies

Computing Gravitational Forces

```
float4 force( float2 ij : WPOS,
      uniform sampler2D pos) : COLOR0
{
   // Pos texture is 2D, not 1D, so we need to
   // convert body index into 2D coords for pos tex
   float4 iCoords = getBodyCoords(ij);
   float4 iPosMass = texture2D(pos, iCoords.xy);
   float4 jPosMass = texture2D(pos, iCoords.zw);
   float3 dir = iPos.xyz - jPos.xyz;
   float r2 = dot(dir, dir);
   return dir * g * iPosMass.w * jPosMass.w / r2;
}
```

Computing Total Force

- **Have: array of (i,j) forces**
- **Need: total force on each particle i**
  - **Sum of each column of the force array**
- **Can do all N columns in parallel**



This is called a *Parallel Reduction*

Parallel Reductions

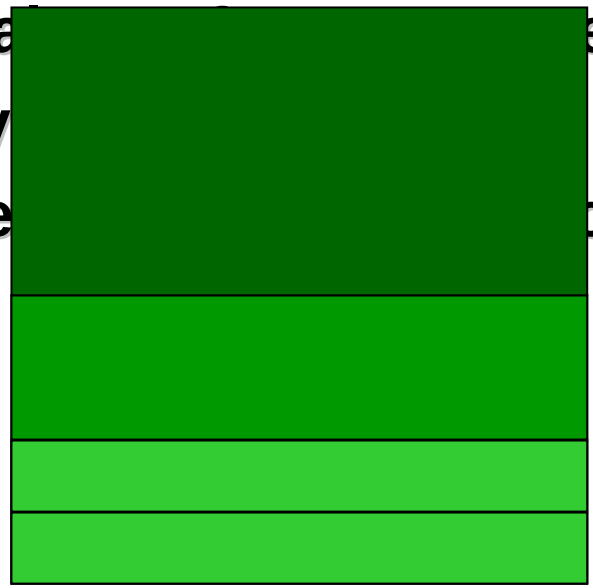
- **1D parallel reduction:**
  - **sum N columns or rows in parallel**
  - **add two ha ether**
  - **repeatedly**
  - **Until we're ow of texels**

*N*x*N*

*N*x(*N*/2)

*N*x(*N*/4)

*N*x1



+

Requires $\log_2 N$ steps

Update Positions and Velocities

- **Now we have an array of total forces**
  - **One per body**
- **Update Velocity**
  - **$u(i, t+dt) = u(i, t) + F_{total}(i) * dt$**
  - **Simple pixel shader reads previous velocity tex and force tex, creates new velocity tex**
- **Update Position**
  - **$x(i, t+dt) = x(i, t) + u(i, t) * dt$**
  - **Simple pixel shader reads previous position and velocity tex, creates new position tex**

GPGPU Flow Control Strategies

Branching and Looping

Per-Fragment Flow Control

- **No true branching on GeForce FX**
  - **Simulated with conditional writes: every instruction is executed, even in branches not taken**
- **GeForce 6 Series has SIMD branching**
  - **Lots of deep pixel pipelines → many pixels in flight**
  - **Coherent branching = good performance**
  - **Incoherent branching = likely performance loss**

Fragment Flow Control Techniques

- **Replace simple branches with math**
  - **x = (t == 0) ? a : b; → x = lerp(t = {0,1}, a, b);**
  - **select() → dot product with permutations of (1,0,0,0)**
- **Try to move decisions up the pipeline**
  - **Occlusion Query**
  - **Static Branch Resolution**
  - **Z-cull**
  - **Pre-computation**

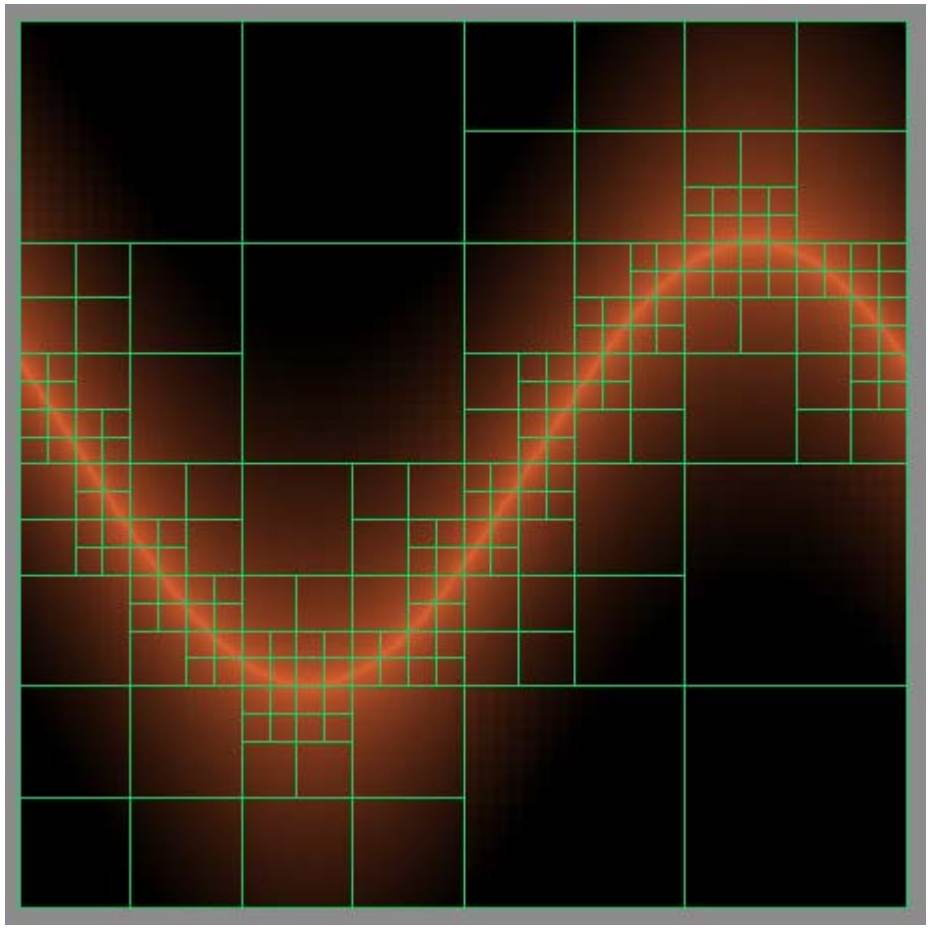
Branching with Occlusion Query

- OQ counts the number of fragments written
  - Use it for iteration termination

```
Do {  // outer loop on CPU
   BeginOcclusionQuery {
       // Render with fragment program
       // that discards fragments that
       // satisfy termination criteria
   } EndQuery
} While query returns > 0
```

  - Can be used for subdivision techniques

Example: OQ-based Subdivision

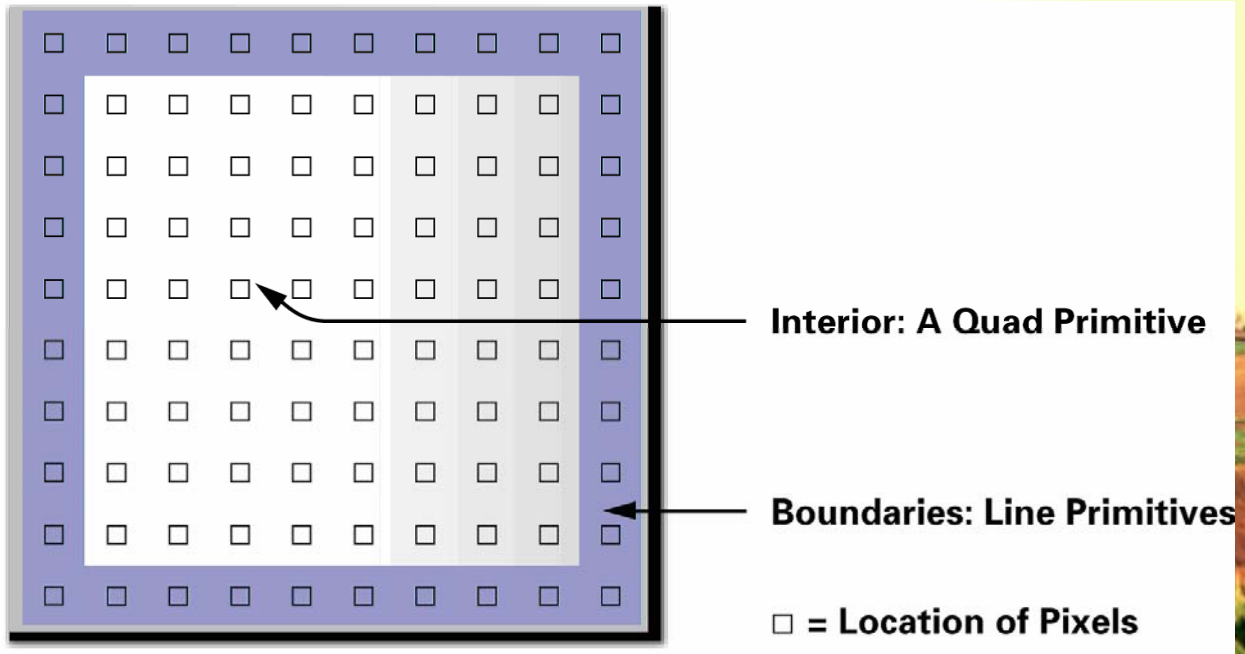


Used in Coombe et al., “Radiosity on Graphics Hardware”

Static Branch Resolution

- **Avoid branches where outcome is fixed**
  - **One region is always true, another false**
  - **Separate FP for each region, no branches**

- **Example: boundaries**



Z-Cull

- **In early pass, modify depth buffer**
  - **Clear Z to 1, enable depth test (GL_LESS)**
  - **Draw quad at Z=0**
  - **Discard pixels that should be modified in later passes**
- **Subsequent passes**
  - **Disable depth write**
  - **Draw full-screen quad at z=0.5**
  - **Only pixels with previous depth=1 will be processed**

Pre-computation

- Pre-compute anything that will not change every iteration!
- Example: arbitrary boundaries
  - When user draws boundaries, compute texture containing boundary info for cells
    - e.g. Offsets for applying PDE boundary conditions
  - Reuse that texture until boundaries modified
  - GeForce 6 Series: combine with Z-cull for higher performance!

Current GPGPU Limitations

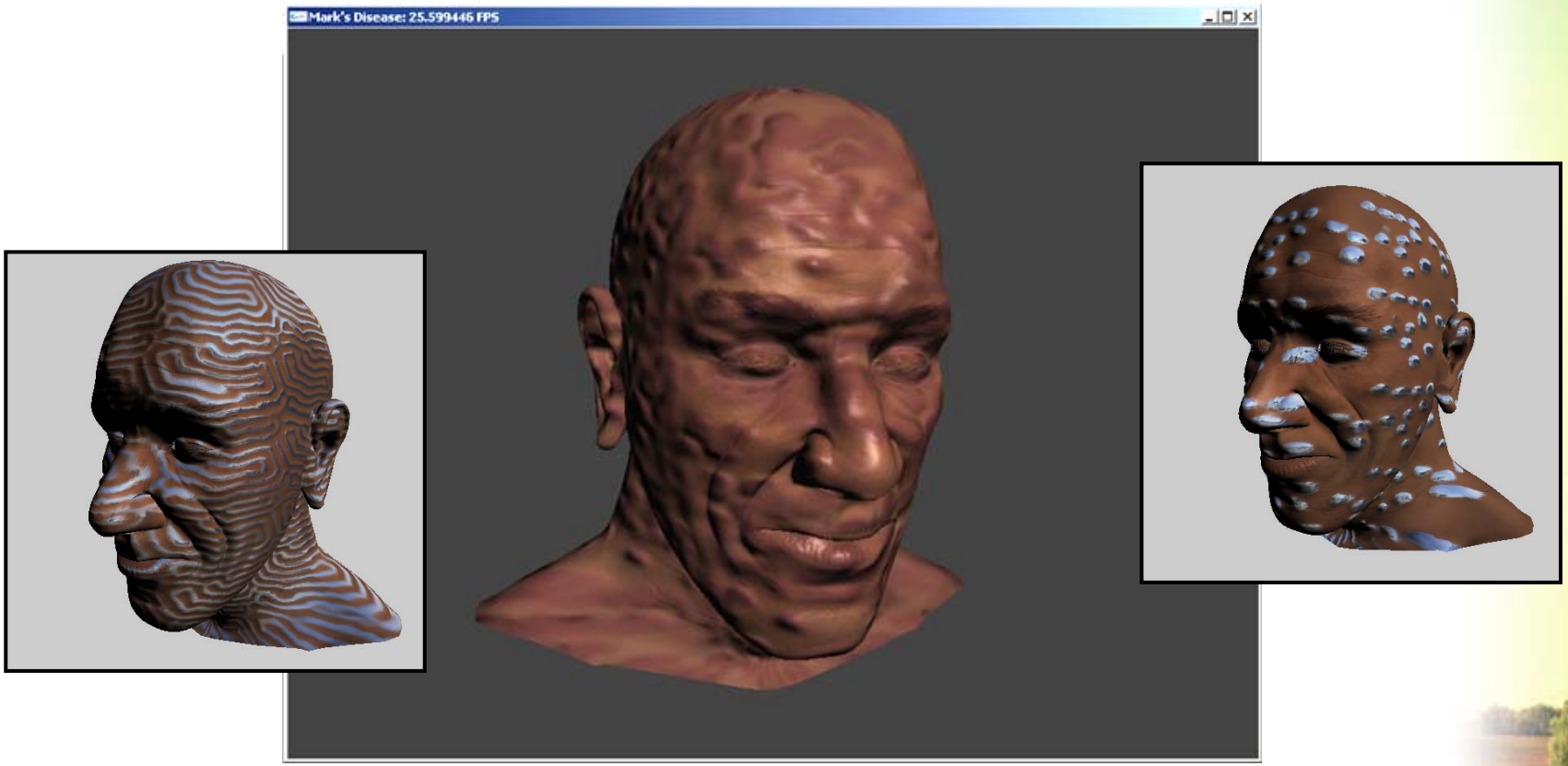
- **Programming is difficult**
  - **Limited memory interface**
  - **Usually “invert” algorithms (Scatter → Gather)**
  - **Not to mention that you have to use a graphics API…**

Brook for GPUs (Stanford)

- C with stream processing extensions
  - Cross compiler compiles to shading language
  - GPU becomes a streaming coprocessor
- A step in the right direction
  - Moving away from graphics APIs
- Stream programming model
  - enforce data parallel computing: streams
  - encourage arithmetic intensity: kernels
- See SIGGRAPH 2004 Paper and
  - http://graphics.stanford.edu/projects/brook
  - http://www.sourceforge.net/projects/brook

More Examples

Demo: "Disease": Reaction-Diffusion



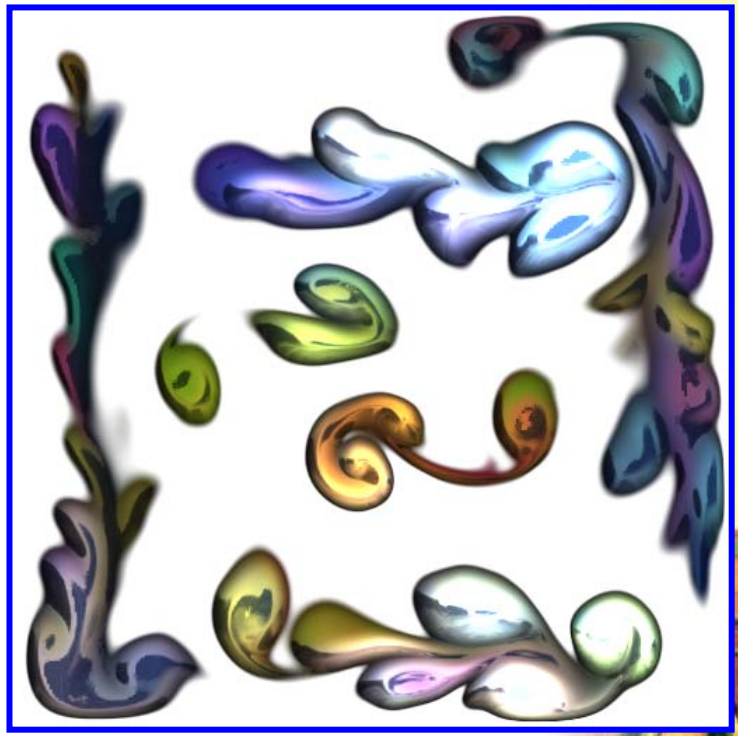
Available in NVIDIA SDK: http://developer.nvidia.com

"Physically-based visual simulation on the GPU",
Harris et al., Graphics Hardware 2002

Example: Fluid Simulation

- **Navier-Stokes fluid simulation on the GPU**
  - **Based on Stam's "Stable Fluids"**
  - **Vorticity Confinement step**
    - **[Fedkiw et al., 2001]**
- **Interior obstacles**
  - **Without branching**
  - **Zcull optimization**
- **Fast on latest GPUs**
  - **~120 fps at 256x256 on GeForce 6800 Ultra**

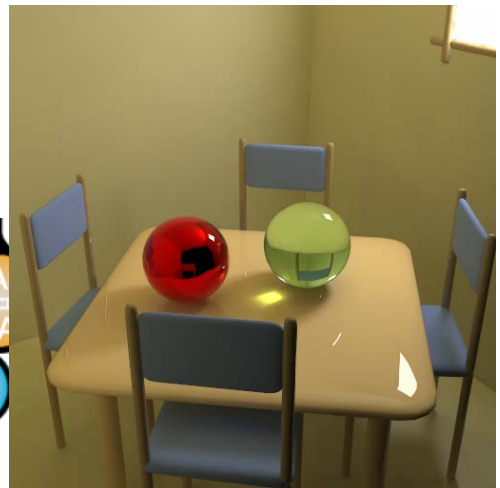
- **Available in NVIDIA SDK 8.5**

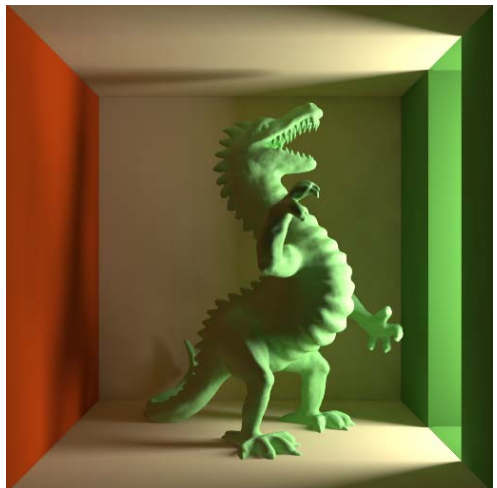


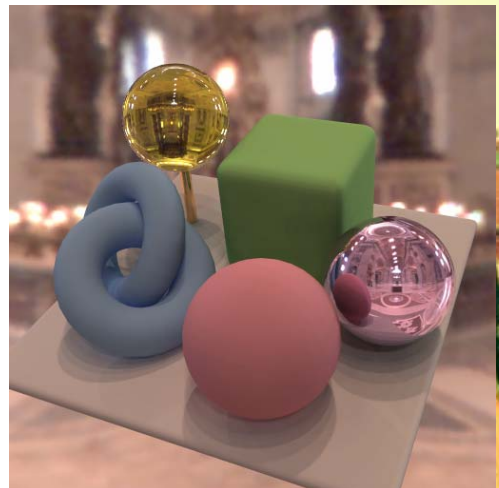
"Fast Fluid Dynamics Simulation on the GPU", Mark Harris. In *GPU Gems*.

Parthenon Renderer

- "High-Quality Global Illumination Using Rasterization", GPU Gems 2
  - Toshiya Hachisuka, U. of Tokyo
- Visibility and Final Gathering on the GPU
  - Final gathering using parallel ray casting + depth peeling







The Future

- **Increasing flexibility**
  - **Always adding new features**
  - **Improved vertex, fragment languages**
- **Easier programming**
  - **Non-graphics APIs and languages?**
  - **Brook for GPUs**
    - **http://graphics.stanford.edu/projects/brookgpu**

The Future

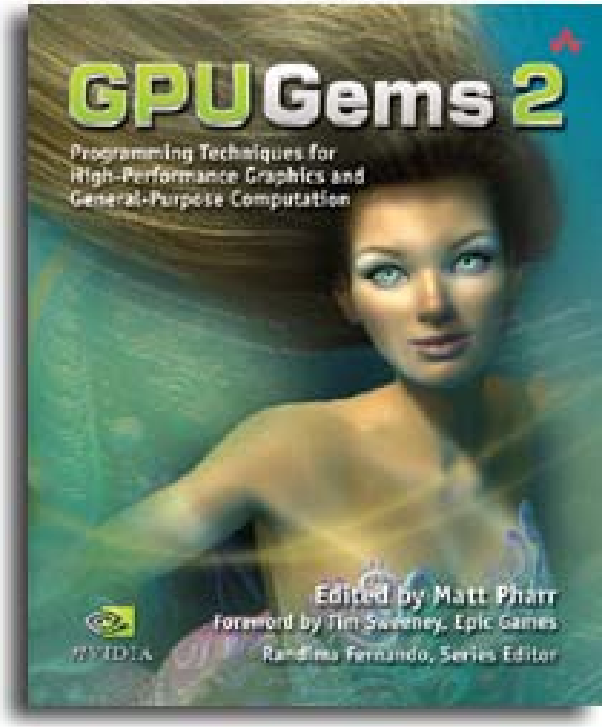
- **Increasing performance**
  - More vertex & fragment processors
  - More flexible with better branching
- **GFLOPs, GFLOPs, GFLOPs!**
  - Fast approaching TFLOPs!
  - Supercomputer on a chip
- **Start planning ways to use it!**

More Information

- GPGPU news, research links and forums
  - www.GPGPU.org
- developer.nvidia.org
- Questions?
  - mharris@nvidia.com

GPU Gems 2

Programming Techniques for High-Performance Graphics and General-Purpose Computation



- **880 full-color pages, 330 figures, hard cover**
- **$59.99**
- **Experts from universities and industry**

18 GPGPU Chapters!

"The topics covered in *GPU Gems 2* are critical to the next generation of game engines."

— Gary McTaggart, Software Engineer at Valve, Creators of Half-Life and Counter-Strike

"*GPU Gems 2* isn't meant to simply adorn your bookshelf—it's required reading for anyone trying to keep pace with the rapid evolution of programmable graphics. If you're serious about graphics, this book will take you to the edge of what the GPU can do."

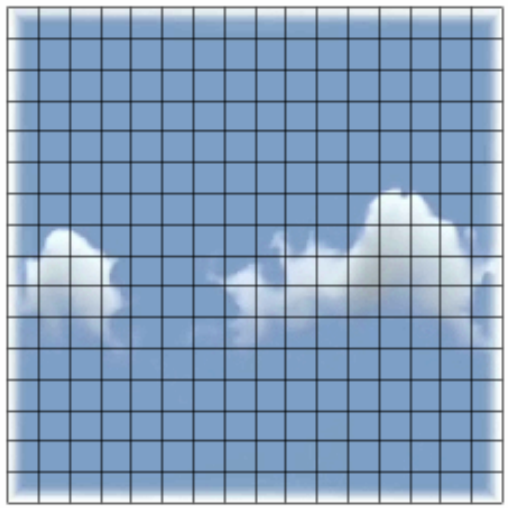
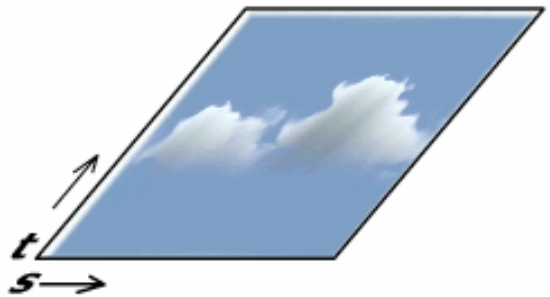
—Rémi Arnaud, Graphics Architect at Sony Computer Entertainment

Arithmetic Intensity

- Arithmetic intensity = ops / bandwidth
- "Classic" Graphics pipeline
  - Vertex
    - BW: 1 triangle = 32 bytes
    - OP: 100-500 f32-ops / triangle
  - Fragment
    - BW: 1 fragment = 10 bytes
    - OP: 300-1000 i8-ops/fragment

Courtesy of Pat Hanrahan

CPU-GPU Analogies

CPU	GPU
	

Stream / Data Array = Texture

Memory Read = Texture Sample

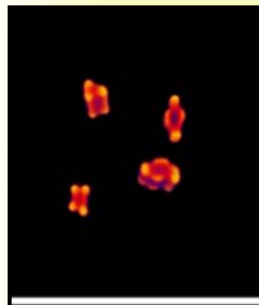
Reaction-Diffusion

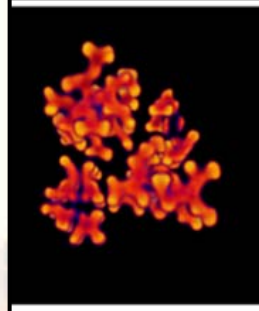
- Gray-Scott reaction-diffusion model [Pearson 1993]
- Streams = two scalar chemical concentrations
- Kernel: just Diffusion and Reaction ops

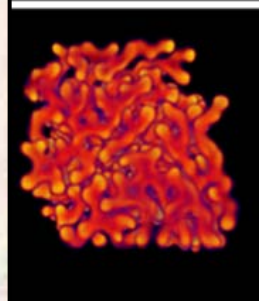
$$\frac{\partial U}{\partial t} = D_u \nabla^2 U - UV^2 + F(1-U),$$

$$\frac{\partial V}{\partial t} = D_v \nabla^2 V + UV^2 - (F+k)V$$

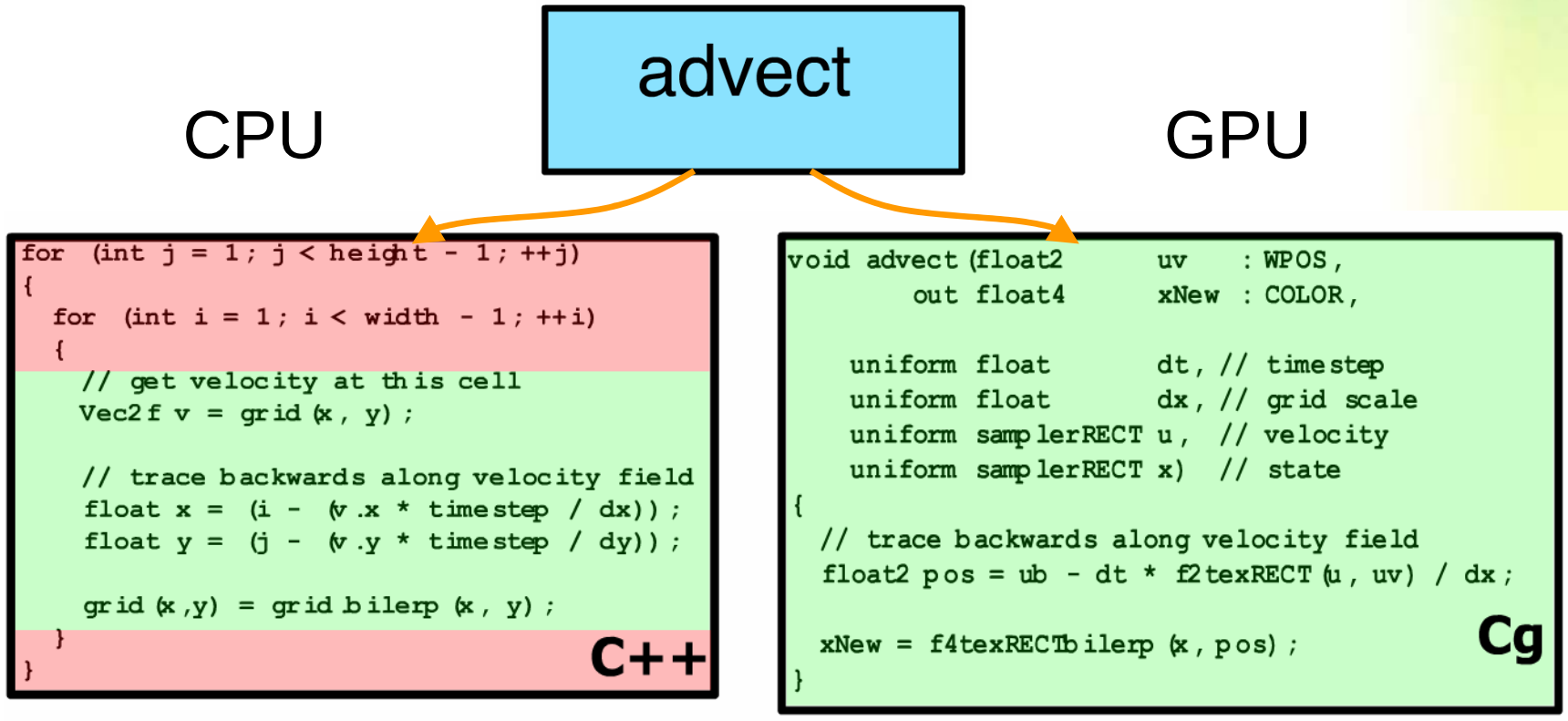
U, V are chemical concentrations,
F, k, D_u, D_v are constants





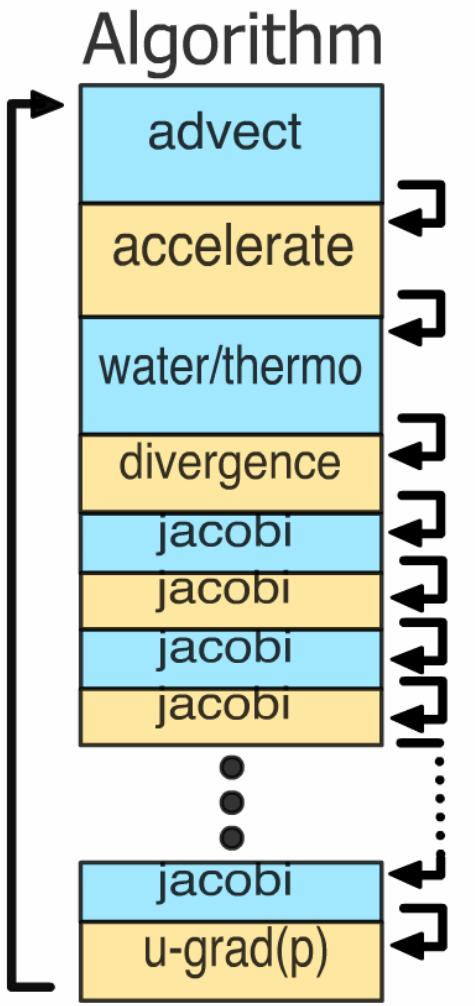


CPU-GPU Analogies



Loop body / kernel / algorithm step = Fragment Program

Feedback



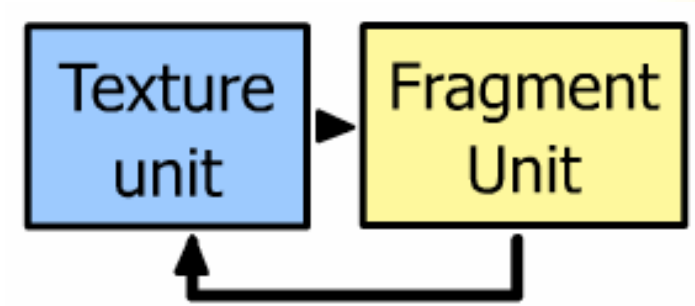
- **Each algorithm step depends on the results of previous steps**
- **Each time step depends on the results of the previous time step**

CPU-GPU Analogies

CPU

GPU

```
.
.
.
Grid[i][j]= x;
.
.
.
```



Array Write to Texture = **Render**

GeForce 6 Series Branching

- True, SIMD branching
  - Incoherent branching can hurt performance
  - Should have coherent regions of > 1000 pixels
    - That is only about 30x30 pixels, so still very useable!
- Don't ignore branch instruction overhead
  - Branching over a few instructions not worth it
- Use branching for early exit from shaders
- GeForce 6 vertex branching is fully MIMD
  - very small overhead and no penalty for divergent branching

A Closing Thought

- **“The Wheel of Reincarnation”**

General computing power, whatever its purpose, should come from the central resources of the system. If these resources should prove inadequate, then it is the system, not the display, that needs more computing power. This decision let us finally escape from the wheel of reincarnation.

-T.H. Myer and I.E. Sutherland. “On the Design of Display Processors”, *Communications of the ACM, Vol. 11, no. 6, June 1968*

My take on the wheel

- Modern applications have a variety of computational requirements
  - Data-Serial
  - Data-Parallel
  - High Arithmetic Intensity
  - High Memory Intensity
- Computer systems must handle this variety

GPGPU and the wheel

- GPUs are uniquely positioned to satisfy the data-parallel needs of modern computing
  - Built for highly data parallel computation
    - Computer Graphics
  - Driven by large-scale economics of a stable and growing industry
    - Computer Games