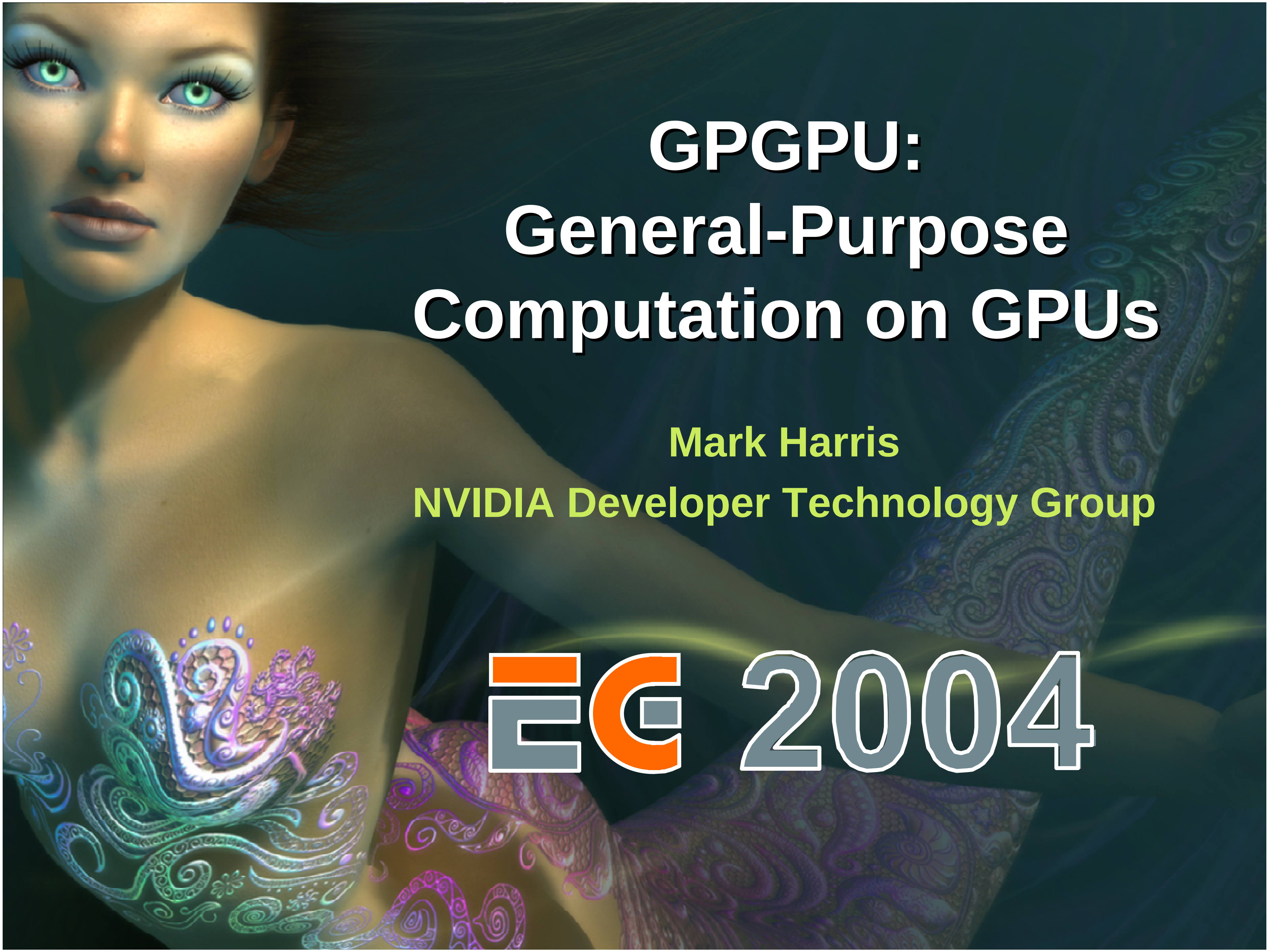


GPGPU: General-Purpose Computation on GPUs

Mark Harris
NVIDIA Developer Technology Group

EG 2004

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*: 6 GFLOPS
    - 5.96 GB/sec peak
  - GeForce FX 5900 *observed*[*]: 20 GFLOPs
    - 25.6 GB/sec peak
  - GeForce 6800 Ultra *observed*[*]: 40 GFLOPs
    - 35.2 GB/sec peak

[*] Observed on a synthetic benchmark:

- A long pixel shader with nothing but MUL 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 www.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
  - Stream Processing
  - 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 Computational Concepts to GPUs

- Remainder of the Talk:

- Data Parallelism and Stream Processing
- Computational Resources Inventory
- CPU-GPU Analogies
- Flow Control Techniques
- Examples and Future Directions

Importance of Data Parallelism

- GPUs are designed for graphics
  - Highly parallel tasks
- GPUs process *independent* vertices & fragments
  - Temporary registers are zeroed
  - No shared or static data
  - No read-modify-write buffers
- Data-parallel processing
  - GPU architecture is ALU-heavy
    - Multiple vertex & pixel pipelines, multiple ALUs per pipe
  - Hide memory latency (with more computation)

Arithmetic Intensity

- Arithmetic intensity = ops per word transferred
- “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

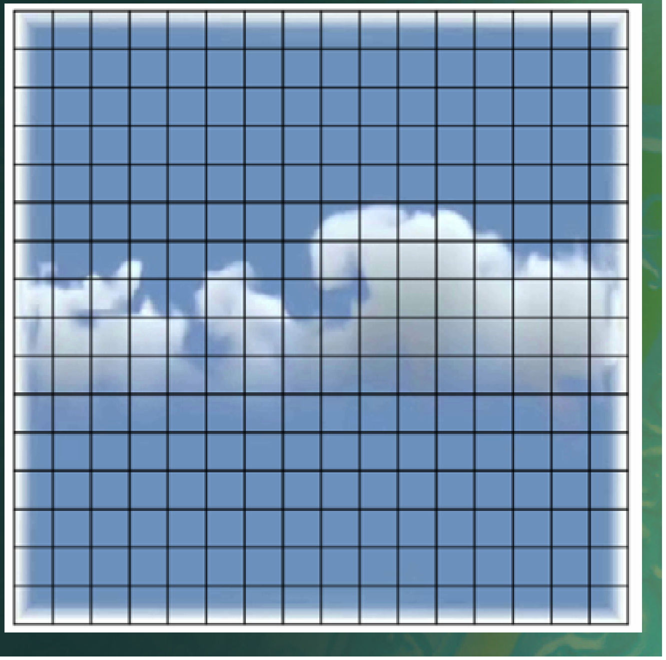
Data Streams & Kernels

- 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

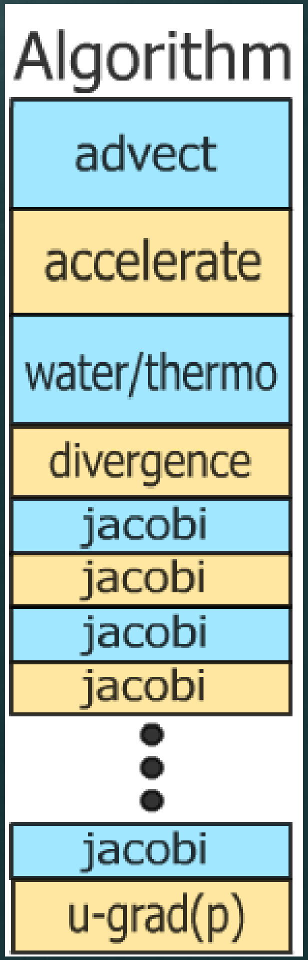
Courtesy of Ian Buck

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



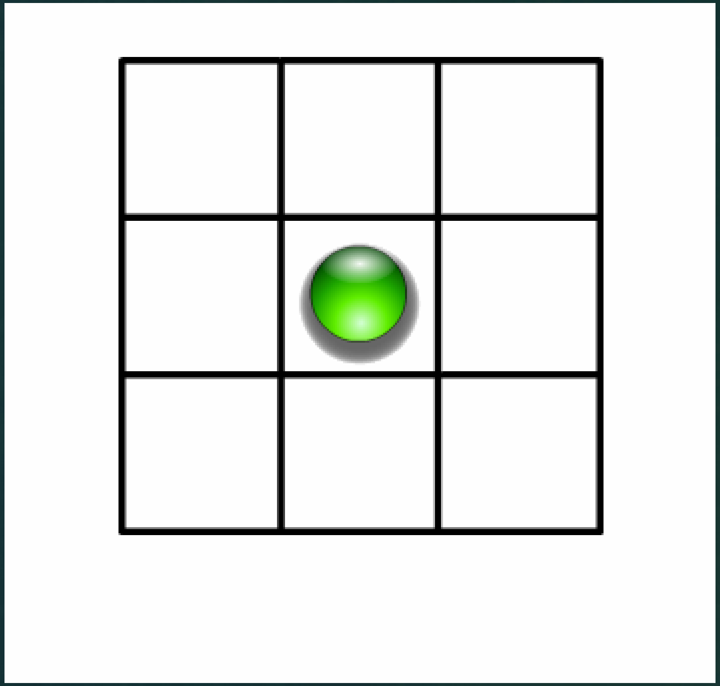
Stream Computation



- 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

Scatter vs. Gather

- Grid communication (a necessary evil)
  - Grid cells share information
  - Two ways:



Computational Resources Inventory

- Programmable parallel processors
  - Vertex & Fragment pipelines
- Rasterizer
  - Mostly useful for interpolating addresses (texture coordinates) and per-vertex constants
- Texture unit
  - Read-only memory interface
- Render to texture
  - Write-only memory interface

Vertex Processor

- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex (scatter)
  - Cannot read info from other vertices (gather)
  - Small constant memory
- New GeForce 6 Series features:
  - Pseudo-gather: read textures in the vertex program
  - MIMD: independent per-vertex branching, early exit

Fragment Processor

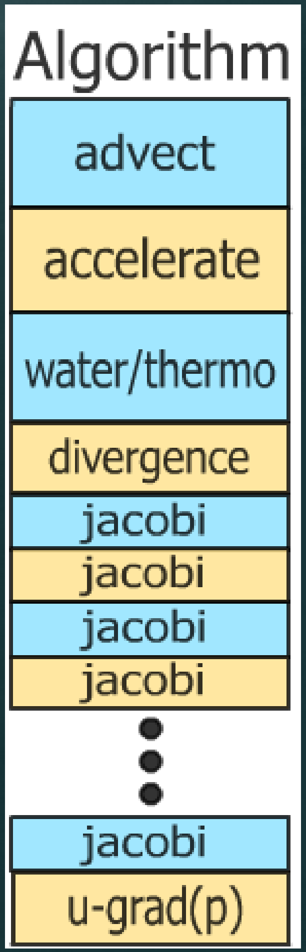
- Fully programmable (SIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of gather but not scatter
  - Random access memory read (textures)
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
  - More fragment pipelines than vertex pipelines
  - Gather / RAM read
  - Direct output
- GeForce 6 Series adds SIMD branching
  - GeForce FX only has conditional writes

CPU-GPU Analogies

- CPU programming is (assumed) familiar
  - GPU programming is graphics-centric
- Analogies can aid understanding

CPU-GPU Analogies

GPU Simulation Overview


Algorithm

advect

accelerate

water/thermo

divergence

jacobi

jacobi

jacobi

jacobi

⋮

jacobi

u-grad(p)

- Analogies lead to implementation
  - Algorithm steps are fragment programs
    - Computational *kernels*
  - Current state variables stored in textures
    - Data *streams*
  - Feedback via render to texture
- One question:
  - How do we invoke computation?

Invoking Computation

- Must invoke computation at each pixel
  - Just draw geometry!
  - Most common GPGPU invocation is a full-screen quad

Standard “Grid” Computation

- Initialize “view” (so that 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)

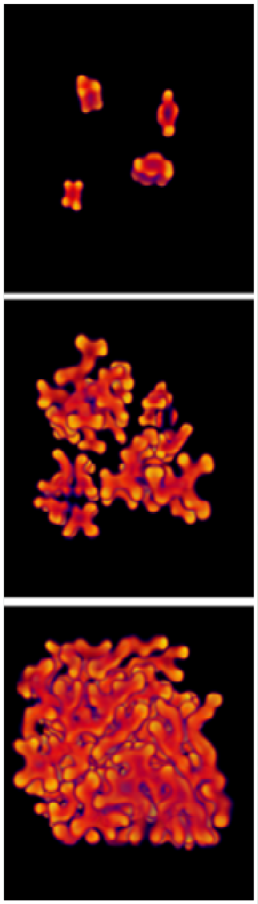
Reaction-Diffusion

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

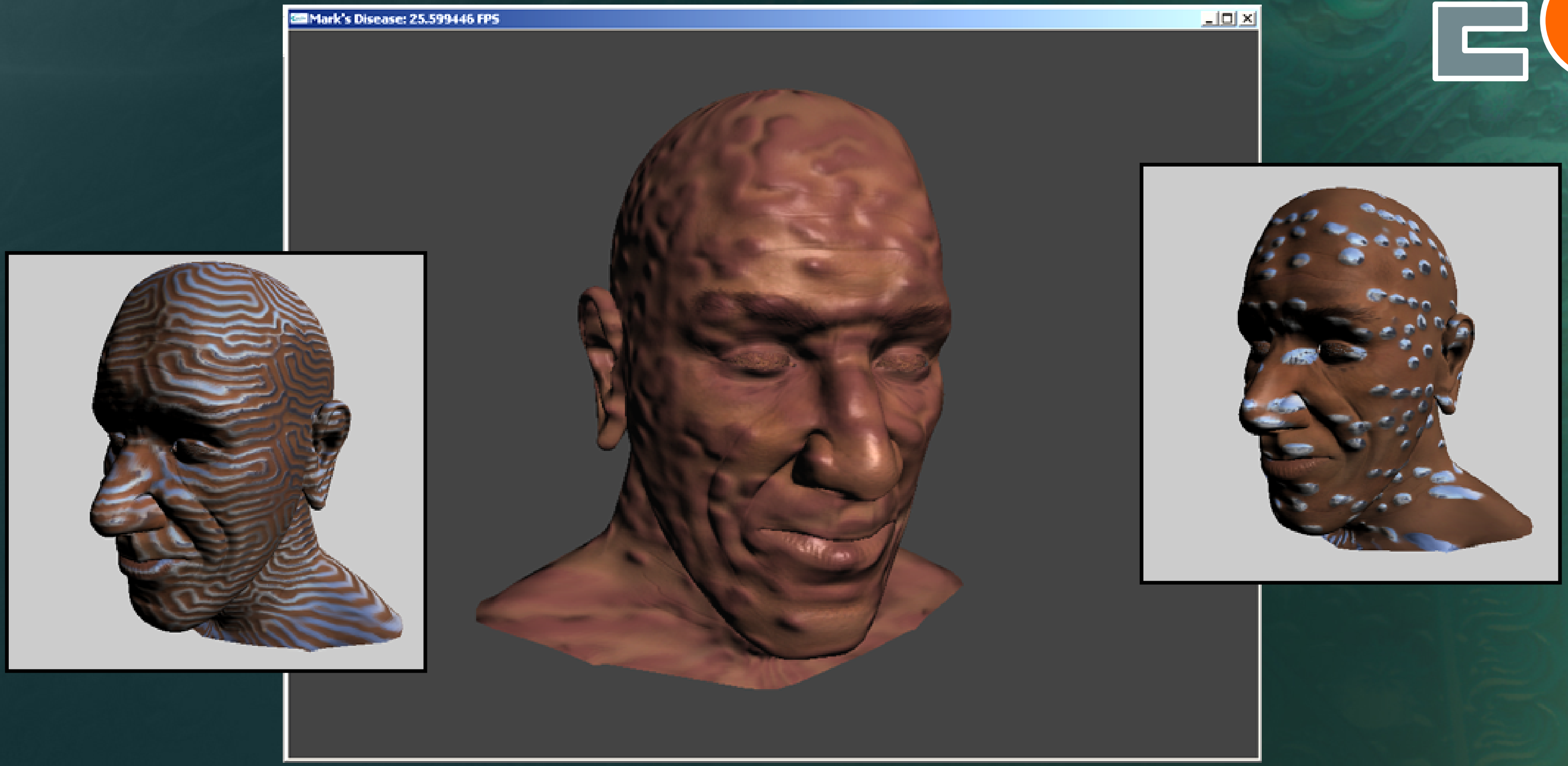
$$\frac{\partial U}{\partial t} = D_u \nabla^2 U - UV^2 + F(1-U),$$

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

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



Demo: “Disease”



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

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

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 = likely performance win
  - Incoherent branching = likely performance loss

Fragment Flow Control Techniques

- Try to move decisions up the pipeline
  - Replace with math
  - Occlusion Query
  - Domain decomposition
  - Z-cull
  - Pre-computation

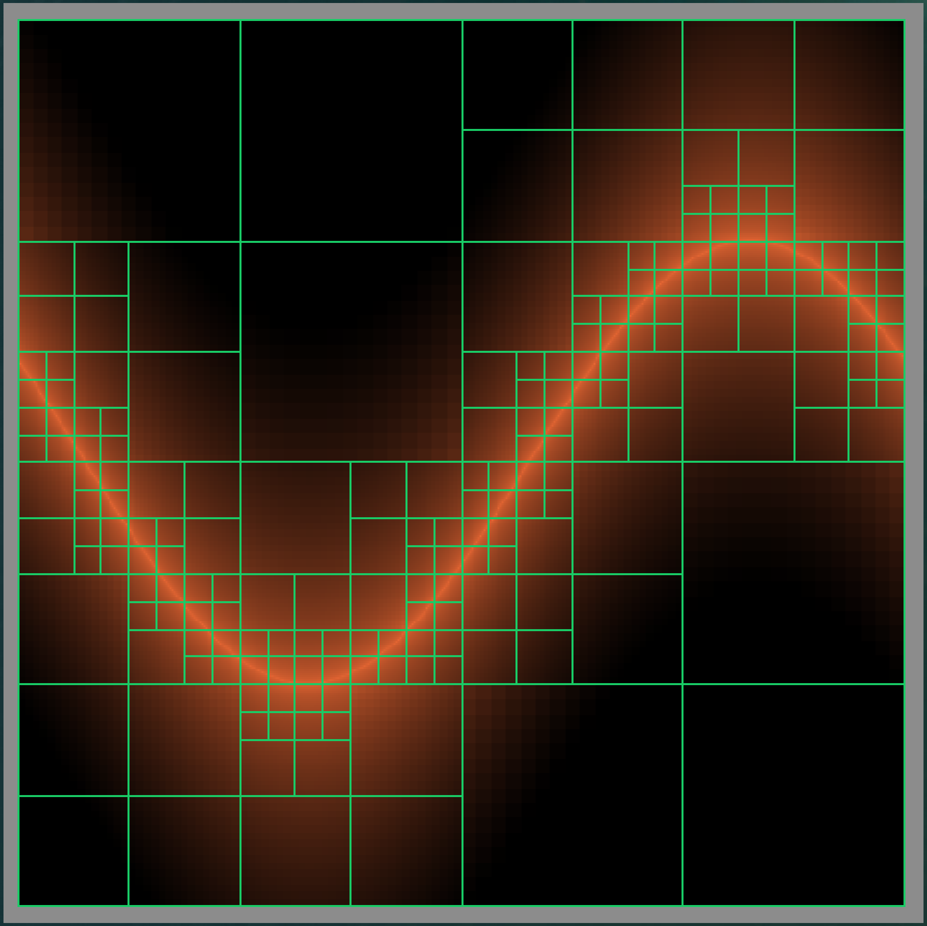
Branching with Occlusion Query

- 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

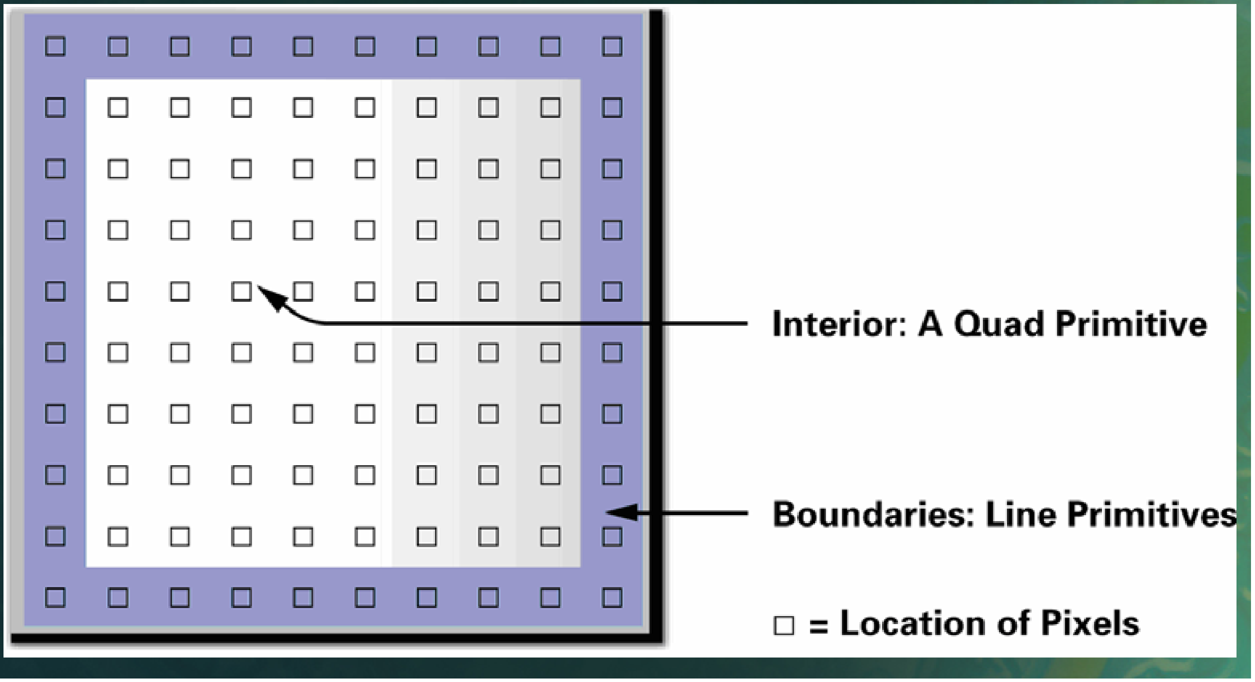
Example: OQ-based Subdivision



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
  - Draw quad at Z=0
  - Discard pixels that should be modified in later passes
- Subsequent passes
  - Enable depth test (GL_LESS), disable depth write
  - Draw full-screen quad at z=0.5
  - Only pixels with previous depth=1 will be processed

- Can also use early stencil test on GeForce 6

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!

GeForce 6 Series Branching

- True, SIMD branching
  - Lots of 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 overhead of branch instructions
  - Branching over only a few instructions not worth it
- Use branching for early exit from loops
  - Save a lot of computation
- GeForce 6 vertex branching is fully MIMD
  - very small overhead and no penalty for divergent branching

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…

- Limitations of communication from GPU to CPU
  - PCI-Express helps
    - GeForce 6 Quadro GPUs: 1.2 GB/s observed
    - Will improve in the near future
  - Frame buffer read can cause pipeline flush
    - Avoid frequent communication to CPU

Brook for GPUs

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

Examples

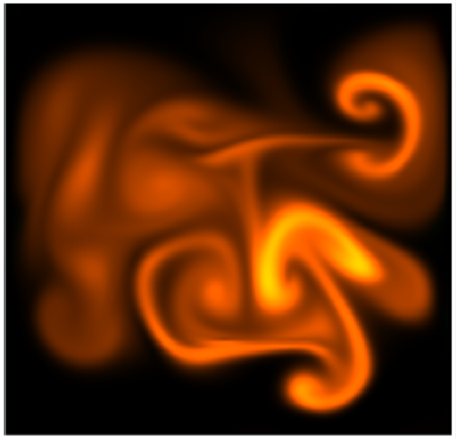
Example: Fluid Simulation

- Navier-Stokes fluid simulation on the GPU
  - Based on Stam's "Stable Fluids"
  - Vorticity Confinement step
    - [Fedkiw et al., 2001]
- Interior obstacles
  - Without branching
- Fast on latest GPUs
  - ~120 fps at 256x256 on GeForce 6800 Ultra
- Available in NVIDIA SDK 8.0

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

Fluid Dynamics

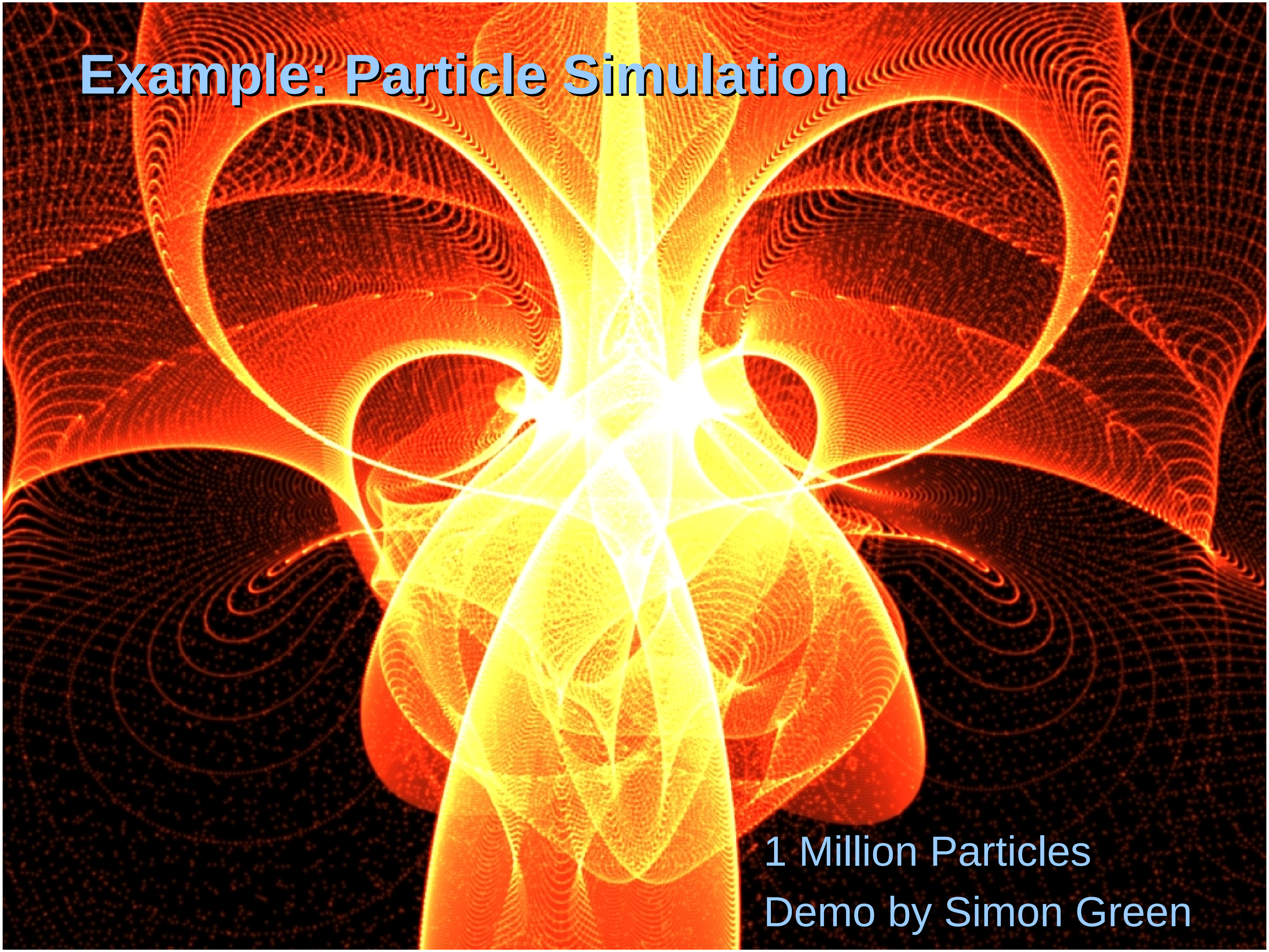
- Solution of Navier-Stokes flow equations
  - Stable for arbitrary time steps
  - [Stam 1999], [Fedkiw et al. 2001]
- Fast on latest GPUs
  - 100+ fps at 256x256 on GeForce 6800 Ultra
- See “Fast Fluid Dynamics Simulation on the GPU”
  - Harris, *GPU Gems*, 2004



Fluid Simulator Demo

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

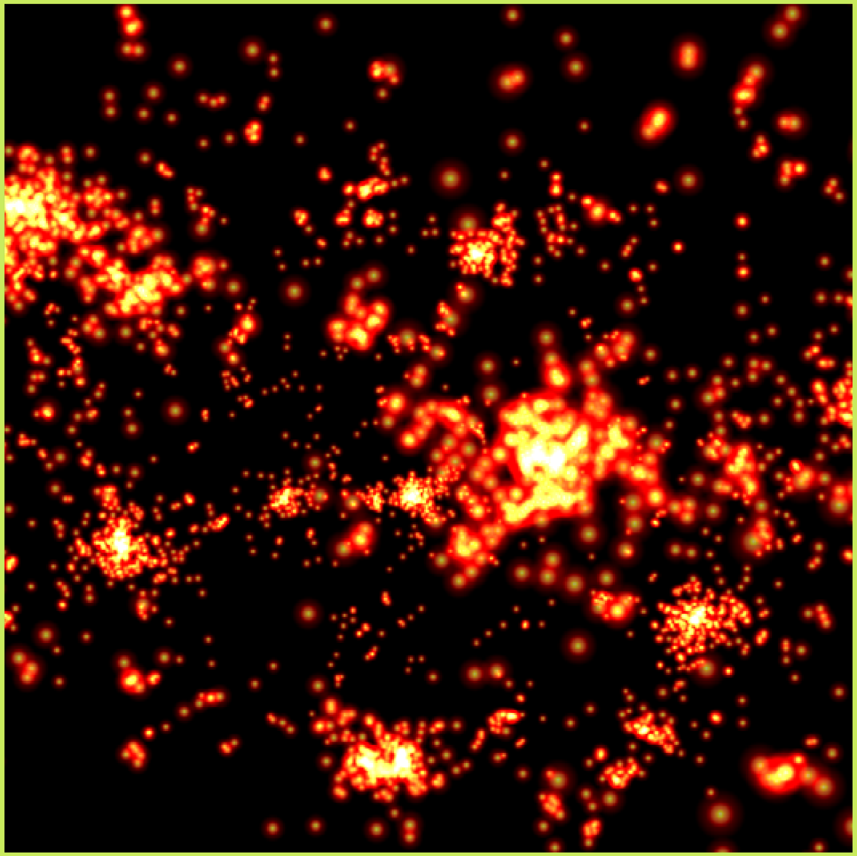
Example: Particle Simulation



1 Million Particles

Demo by Simon Green

Example: N-Body Simulation


Nyland et al., GP^2 poster

- Brute force ☹
- N = 4096 particles
- N^2 gravity computations

- 16M force comps. / frame
- ~25 flops per force
- 17+ fps

- 7+ GFLOPs sustained

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

New Functionality Overview

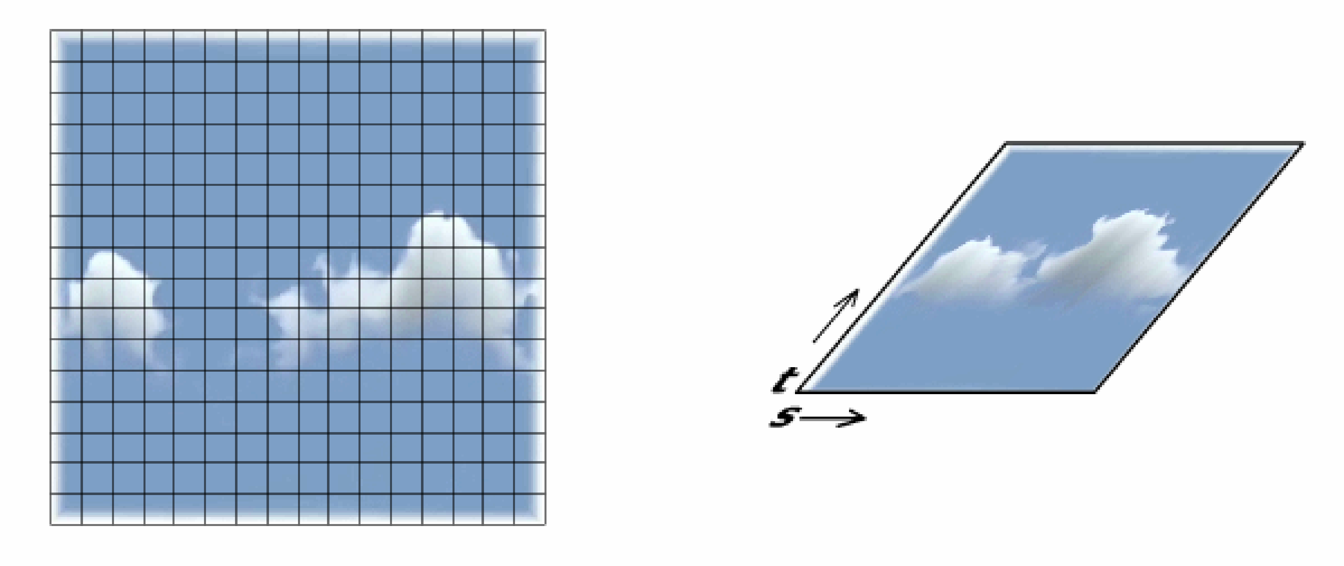
- Vertex Programs
  - Vertex Textures: gather
  - MIMD processing: full-speed branching
- Fragment Programs
  - Looping, branching, subroutines, indexed input arrays, explicit texture LOD, facing register
- Multiple Render Targets
  - More outputs from a single shader
  - Fewer passes, side effects

New Functionality Overview

- VBO / PBO & Superbuffers
  - Feedback texture to vertex input
  - Render simulation output as geometry
  - Not as flexible as vertex textures
    - No random access, no filtering
  - Demos
- PCI-Express
  - Higher GPU←→CPU bandwidth

CPU-GPU Analogies

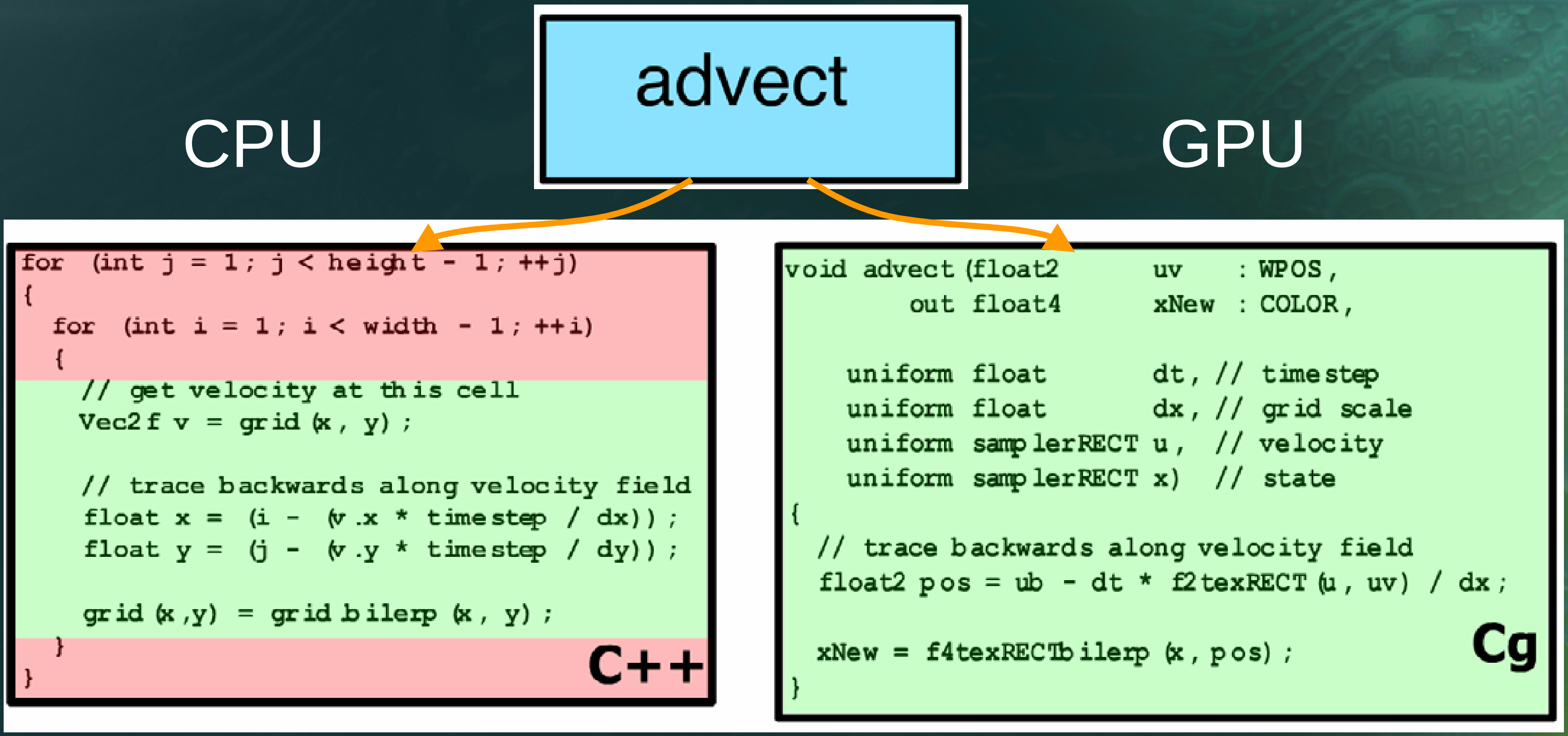
CPU GPU



Stream / Data Array = Texture

Memory Read = Texture Sample

CPU-GPU Analogies



advect

CPU

GPU

```
for  (int j = 1; j < height - 1; ++j)
{
  for  (int i = 1; i < width - 1; ++i)
  {
    // get velocity at this cell
    Vec2f v = grid(x, y);

    // trace backwards along velocity field
    float x = (i - (v.x * timestep / dx));
    float y = (j - (v.y * timestep / dy));

    grid(x,y) = grid.bilerp(x, y);
  }
}
```

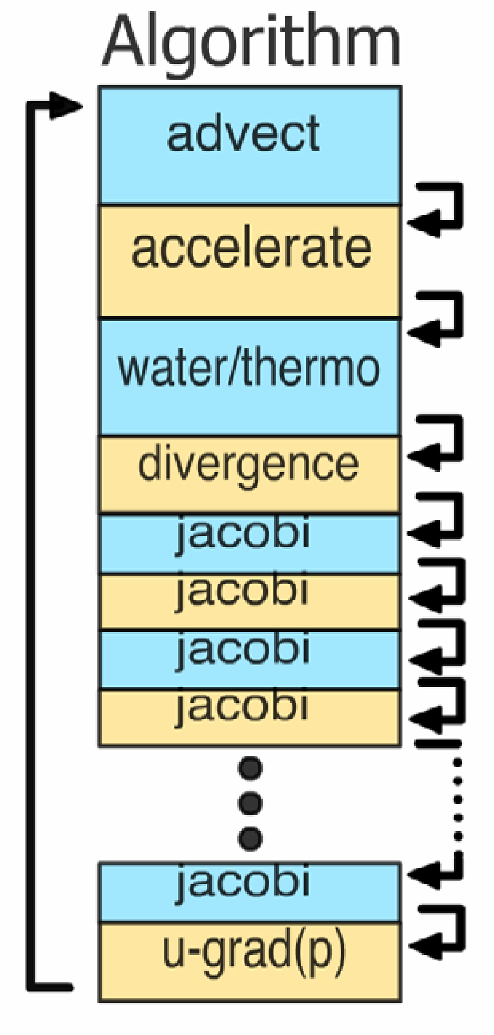
C++

```
void advect(float2         uv    : WPOS,
        out float4         xNew  : COLOR,

    uniform float          dt, // timestep
    uniform float          dx, // grid scale
    uniform samplerRECT u,  // velocity
    uniform samplerRECT x)  // state
{
  // trace backwards along velocity field
  float2 pos = ub - dt * f2texRECT(u, uv) / dx;

  xNew = f4texRECTbilerp(x, pos);
}
```

Cg

Loop body / kernel / algorithm step = Fragment Program

Feedback



- Each algorithm step depend on the results of previous steps

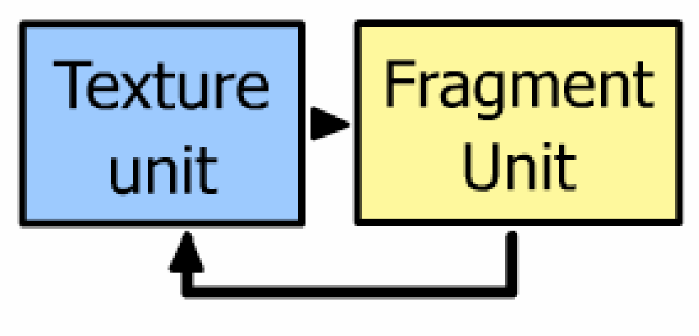
- Each time step depends on the results of the previous time step

CPU-GPU Analogies

CPU

GPU

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



Array Write = Render to Texture

Navier-Stokes Equations

- Describe flow of an incompressible fluid

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u}\cdot\nabla)\mathbf{u} - \frac{1}{\rho}\nabla p - \nu\nabla^2\mathbf{u} + \mathbf{f}$$

Advection

Pressure Gradient

Diffusion (viscosity)

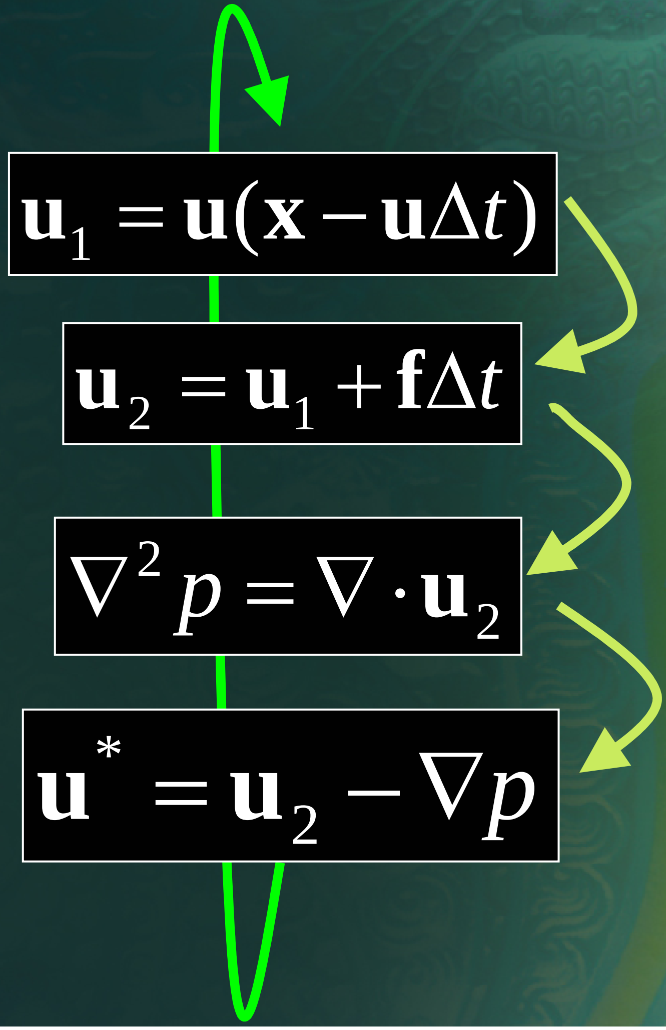
External Force

$$\nabla\cdot\mathbf{u} = 0$$

← Velocity is divergence-free

Fluid Algorithm

- Break it down [Stam 1999]:

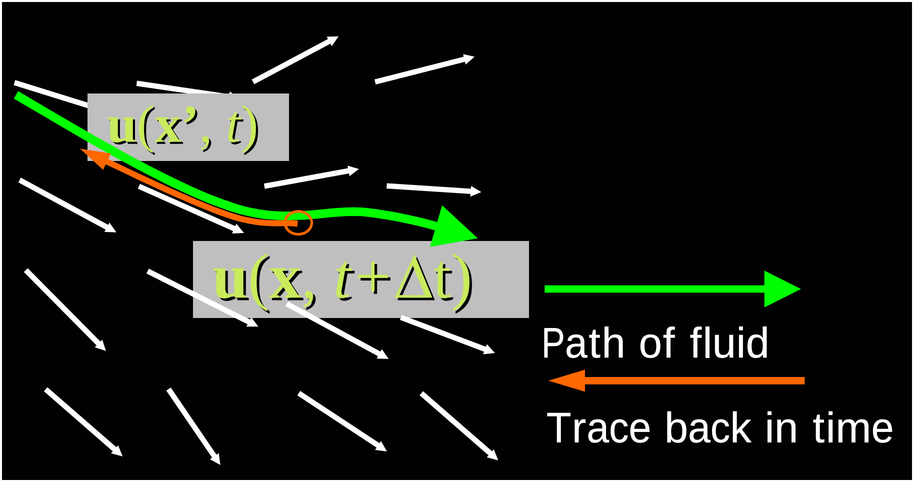


  - Advect: $\mathbf{u}_1 = \mathbf{u}(\mathbf{x} - \mathbf{u}\Delta t)$
  - Add forces: $\mathbf{u}_2 = \mathbf{u}_1 + \mathbf{f}\Delta t$
  - Solve for pressure: $\nabla^2 p = \nabla \cdot \mathbf{u}_2$
  - Subtract pressure gradient: $\mathbf{u}^* = \mathbf{u}_2 - \nabla p$

Advection

- Advection: quantities in a fluid are carried along by velocity
  - Follow velocity field back
    position

$$\mathbf{u}_1 = \mathbf{u}(\mathbf{x} - \mathbf{u}\Delta t)$$



```
float2 pos =
    coords – delta_t * tex(u, coords);

uNew = texBilerp(u, pos);
```

Poisson-Pressure Solution

$$\nabla^2 p = \nabla \cdot \mathbf{u}_2$$

- Discretize equation, solve using iterative solver
  - Jacobi, multigrid, conjugate gradient, etc.
  - Jacobi easy on GPU, but others possible too
  - Demo uses Jacobi iteration (50 iterations by default)
- Compute divergence field, then repeatedly

```
float pL  = tex(pressure, coords + float2(-1, 0));
float pR  = tex(pressure, coords + float2( 1, 0));
float pB  = tex(pressure, coords + float2( 0,-1));
float pT  = tex(pressure, coords + float2( 0, 1));

float div = tex(divergence, coords);

pNew = 0.25 * (pL + pR + pB + pT – delta2 * div);
```