CUDA

Fluid simulation

Lattice Boltzmann Models

Cellular Automata

Please excuse my layout of slides for the remaining part of the talk!

Fluid Simulation

- Navier Stokes equations for incompressible fluids
- Well known technique from computer graphics
  - Stable fluids, Jos Stam, Siggraph ’99
  - Can take arbitrary large time steps
  - Finite difference approximations results in grid computations similar to LBM

SDK fluids

DEMO

SDK fluids

Not 100% identical to the following method…

Navier-Stokes (again)

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

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

Projection operator

- Define a projection operator $\mathcal{P}$ that projects a vector field **w** onto divergence-free component **u**.
- $\mathcal{P}\mathbf{w} = \mathcal{P}\mathbf{u} + \mathcal{P}(\nabla p)$
  - Since by definition $\mathcal{P}\mathbf{w} = \mathcal{P}\mathbf{u} = \mathbf{u}$ then $\mathcal{P}(\nabla p)=0$
- And then

$$\mathbb{P}\frac{\partial \mathrm{u}}{\partial t} = \mathbb{P}\left(-(\mathrm{u}\cdot\nabla)\mathrm{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\mathrm{u} + \mathrm{F}\right)$$

$$\frac{\partial \mathrm{u}}{\partial t} = \mathbb{P}\left(-(\mathrm{u}\cdot\nabla)\mathrm{u} + \nu\nabla^2\mathrm{u} + \mathrm{F}\right)$$

Algorithm

- Break it down [Stam 2000]:
  - Add forces
  - Advect
  - Diffuse
  - Solve for pressure
  - Subtract pressure gradient

Algorithm

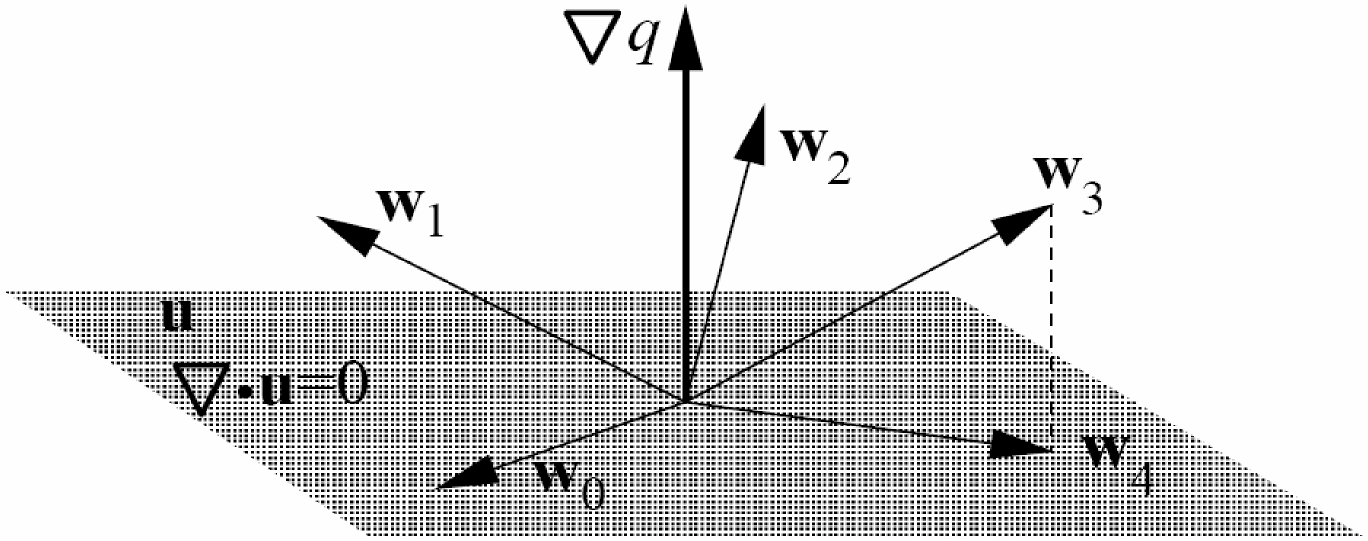


Figure 1: One simulation step of our solver is composed of steps. The first three steps may take the field out of the space of divergent free fields. The last projection step ensures that the field is divergent free after the entire simulation step.

From: Stam J. Stable Fluids. Siggraph 1999.

Algorithm

- Break it down [Stam 2000]:
  - **Add forces**:

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

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Add Forces

Explicit Euler integration

$$\mathbf{w}_1 = \mathbf{u}(\mathbf{x},t) + \mathbf{f}(\mathbf{x},t)\Delta t$$

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - **Advect**:

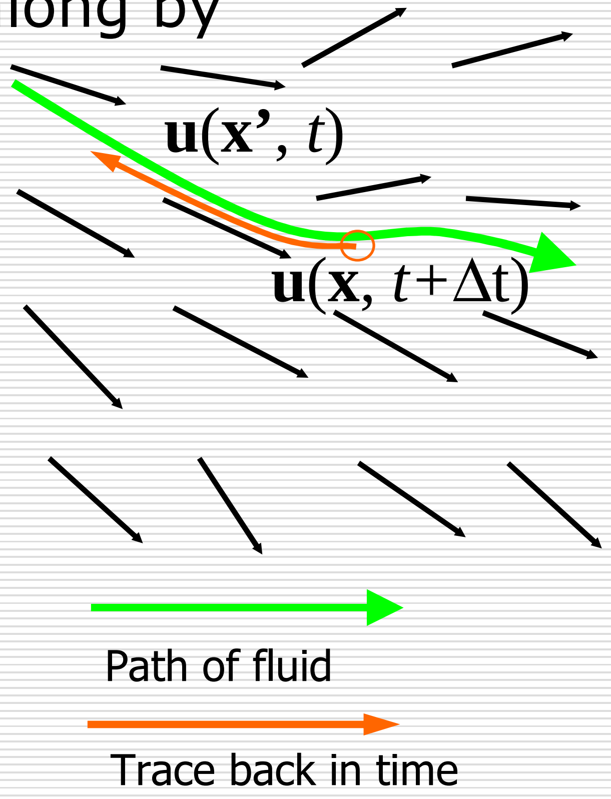
$$\frac{\partial \mathbf{u}}{\partial t} = \underbrace{-(\mathbf{u} \cdot \nabla)\mathbf{u}}_{\text{Advection}} - \frac{1}{\rho}\nabla p - \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

Advection

- Advection:
  - quantities in a fluid are carried along by its velocity

$$\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{x} - \mathbf{w}_1 \Delta t)$$

- Want velocity at position x at new time t + Δt
- Follow velocity field back in time from x : $(x - w_1\Delta t)$
  - Like tracing particles!
  - Simple in a fragment program



$\mathbf{u}(\mathbf{x}', t)$
$\mathbf{u}(\mathbf{x}, t+\Delta t)$
Path of fluid
Trace back in time

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - **Diffuse**:

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

Diffusion (viscosity)

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Numerical integration

- Implicit
  - Stable for large timesteps

$$\frac{dw_2}{dt} = \frac{w_2(x,t+h) - w_2(x,t)}{h} = \nu\nabla^2 w_2(x,t+h) \Rightarrow$$

$$w_2(x,t+h) = w_2(x,t) + h\nu\nabla^2 w_2(x,t+h) \Rightarrow$$

$$(I - h\nu\nabla^2)\underbrace{w_2(x,t+h)}_{\mathbf{w}_3(\mathbf{x})} = w_2(x,t)$$

Viscous Diffusion

$$\left(\mathbf{I} - \nu \Delta t \nabla^2\right)\mathbf{w}_3 = \mathbf{w}_2$$

- Solution by Jacobi iteration

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - Diffuse:
  - **Solve for pressure**: $\nabla^2 p = \nabla \cdot \mathbf{w}_3$

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Poisson-Pressure Solution

- Poisson Equation
  - Jacobi, Gauss-Seidel, Multigrid, etc.
    - Jacobi easy on GPU, the rest are trickier

$$\nabla^2 p = \nabla \cdot \mathbf{w}_3$$

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Algorithm

- Break it down [Stam 2000]:
  - Add forces:
  - Advect:
  - Diffuse:
  - Solve for pressure:
  - **Subtract pressure gradient**: $\mathbf{u}(\mathbf{x}, t+\Delta t) = \mathbf{w}_3 - \nabla p$

Subtract Pressure Gradient

$$\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{w}_3 - \nabla p$$

- Last computation of the time step
  - u is now a divergence-free velocity field

Source: Mark Harris / GPGPU tutorial@Siggraph 04

Implementation

Cg code.

But we “convert” to Cuda on the fly…

Pseudocode of timestep

```
// Apply the first 3 operators in Equation 12.
u = advect(u);
u = diffuse(u);
u = addForces(u);
// Now apply the projection operator to the result.
p = computePressure(u);
u = subtractPressureGradient(u, p);
```

Textures

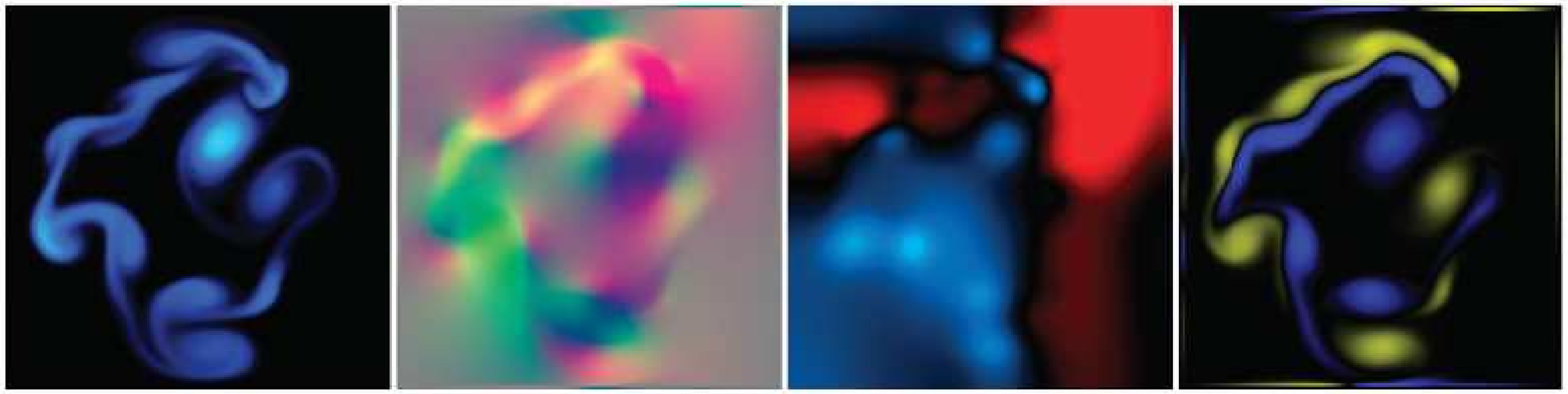
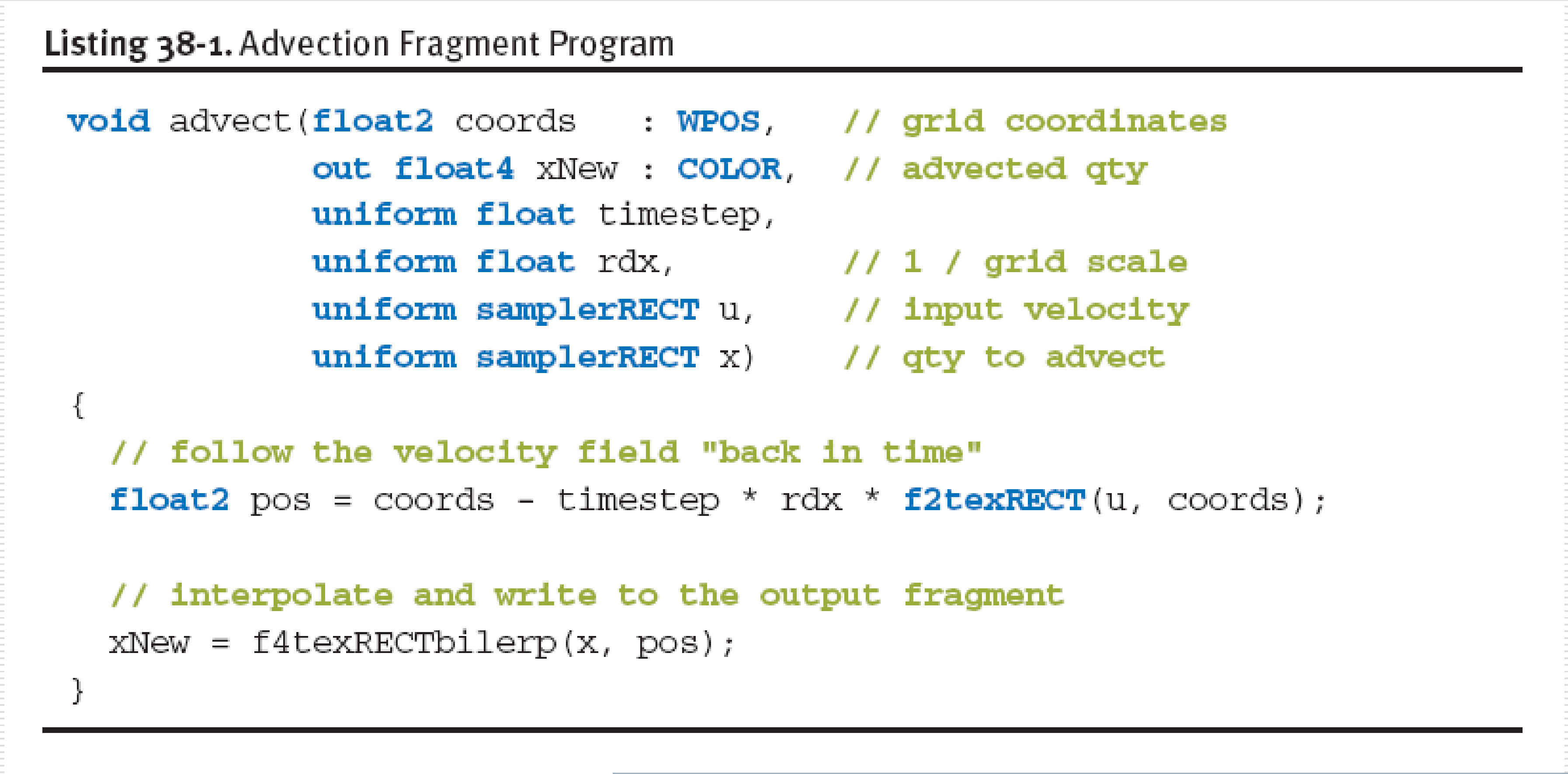


Figure 38-4. The State Fields of a Fluid Simulation, Stored in Textures
From left to right, the fields are "ink," velocity (scaled and biased into the range [0, 1], so zero velocity is gray), pressure (blue represents low pressure, red represents high pressure), and vorticity (yellow represents counter-clockwise rotation, blue represents clockwise rotation).

Advection



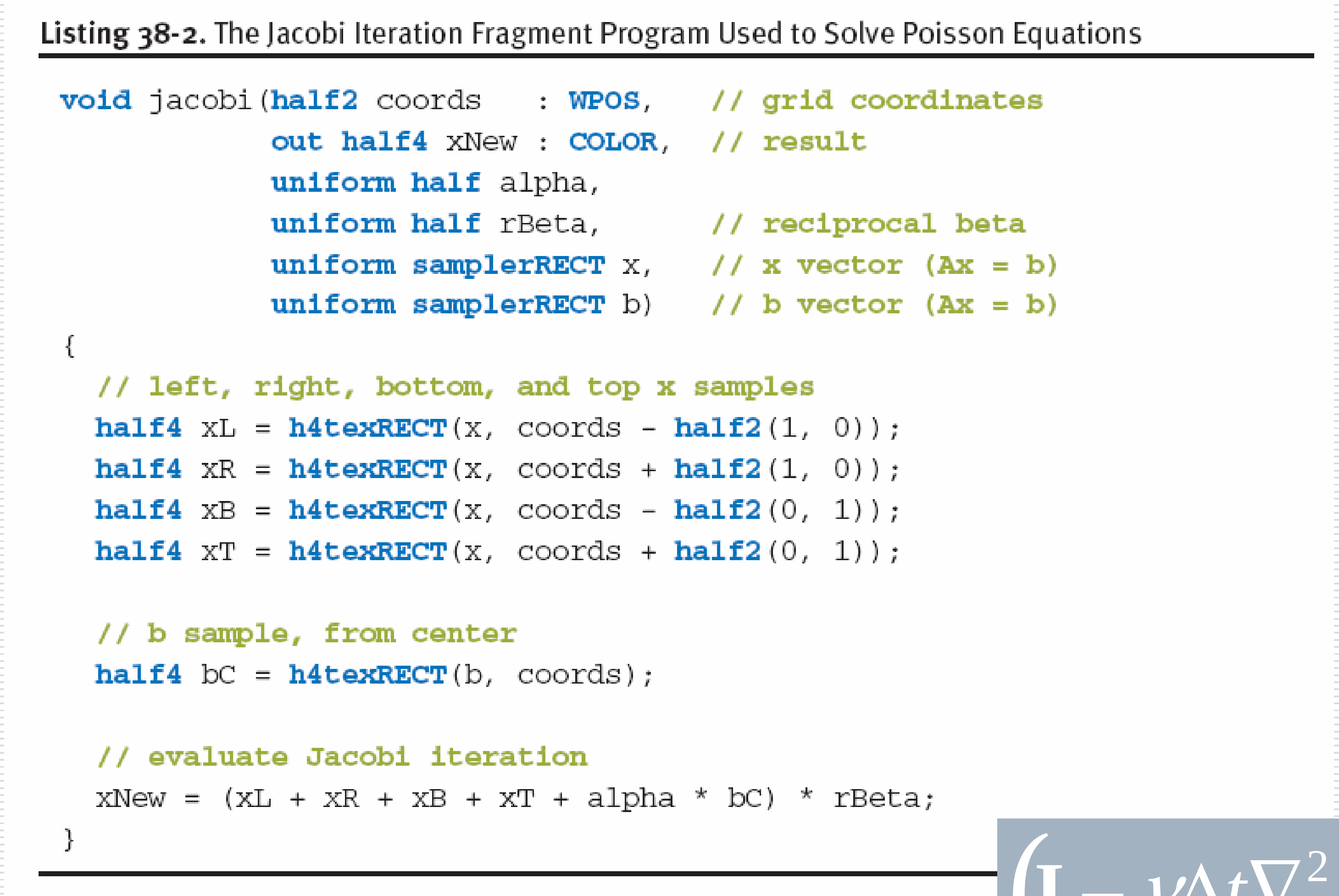
Listing 38-1. Advection Fragment Program

```
void advect(float2 coords   : WPOS,    // grid coordinates
            out float4 xNew : COLOR,   // advected qty
            uniform float timestep,
            uniform float rdx,         // 1 / grid scale
            uniform samplerRECT u,     // input velocity
            uniform samplerRECT x)     // qty to advect
{
  // follow the velocity field "back in time"
  float2 pos = coords - timestep * rdx * f2texRECT(u, coords);

  // interpolate and write to the output fragment
  xNew = f4texRECTbilerp(x, pos);
}
```

$$\mathbf{w}_2(\mathbf{x}) = \mathbf{w}_1(\mathbf{x} - \mathbf{w}_1 \Delta t)$$

Diffusion



Listing 38-2. The Jacobi Iteration Fragment Program Used to Solve Poisson Equations

```
void jacobi(half2 coords   : WPOS,   // grid coordinates
            out half4 xNew : COLOR,  // result
            uniform half alpha,
            uniform half rBeta,      // reciprocal beta
            uniform samplerRECT x,   // x vector (Ax = b)
            uniform samplerRECT b)   // b vector (Ax = b)
{
  // left, right, bottom, and top x samples
  half4 xL = h4texRECT(x, coords - half2(1, 0));
  half4 xR = h4texRECT(x, coords + half2(1, 0));
  half4 xB = h4texRECT(x, coords - half2(0, 1));
  half4 xT = h4texRECT(x, coords + half2(0, 1));

  // b sample, from center
  half4 bC = h4texRECT(b, coords);

  // evaluate Jacobi iteration
  xNew = (xL + xR + xB + xT + alpha * bC) * rBeta;
}
```

$$\left(\mathbf{I} - \nu \Delta t \nabla^2\right)\mathbf{w}_3 = \mathbf{w}_2$$

Divergence

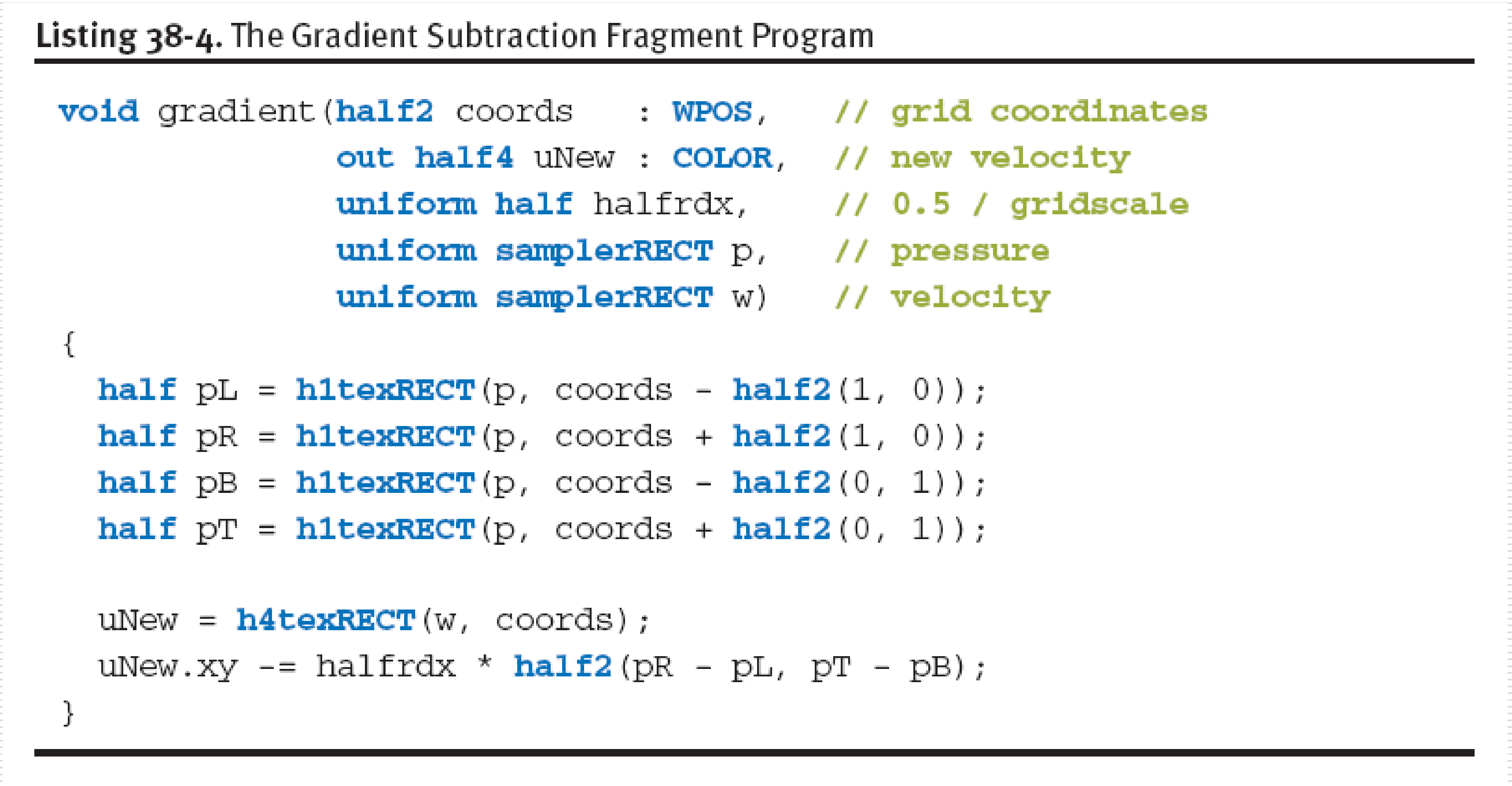
Listing 38-3. The Divergence Fragment Program

```
void divergence(half2 coords  : WPOS,    // grid coordinates
                out half4 div : COLOR,   // divergence
                uniform half halfrdx,    // 0.5 / gridscale
                uniform samplerRECT w)   // vector field
{
  half4 wL = h4texRECT(w, coords - half2(1, 0));
  half4 wR = h4texRECT(w, coords + half2(1, 0));
  half4 wB = h4texRECT(w, coords - half2(0, 1));
  half4 wT = h4texRECT(w, coords + half2(0, 1));

  div = halfrdx * ((wR.x - wL.x) + (wT.y - wB.y));
}
```

$$\nabla^2 p = \nabla \cdot \mathrm{w}, \qquad (10)$$

Gradient subtraction



Listing 38-4. The Gradient Subtraction Fragment Program

```
void gradient(half2 coords   : WPOS,    // grid coordinates
              out half4 uNew : COLOR,   // new velocity
              uniform half halfrdx,     // 0.5 / gridscale
              uniform samplerRECT p,    // pressure
              uniform samplerRECT w)    // velocity
{
  half pL = h1texRECT(p, coords - half2(1, 0));
  half pR = h1texRECT(p, coords + half2(1, 0));
  half pB = h1texRECT(p, coords - half2(0, 1));
  half pT = h1texRECT(p, coords + half2(0, 1));

  uNew = h4texRECT(w, coords);
  uNew.xy -= halfrdx * half2(pR - pL, pT - pB);
}
```

$$\mathbf{u} = \mathbf{w} - \nabla p. \tag{8}$$

Partial differential equations

- Finite difference discretizations lead to “lattice formulations”
  - Like the Jacobi iteration program
  - Similar in implementation to cellular automata

Lattice Boltzmann Models

Speculations…

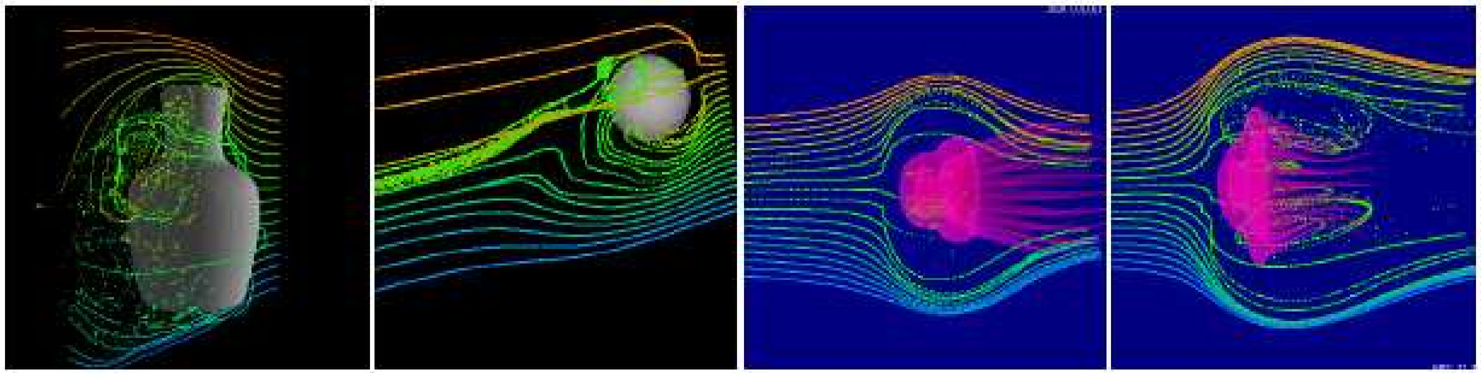
Implementing Lattice Boltzmann Computation on Graphics Hardware

Wei Li, Xiaoming Wei, and Arie Kaufman.

The Visual Computer 19(7-8) **2003**.

Hardware-near paper

Extrapolate…



movie

LBM

- The LBM consists of a regular grid and a set of packet distribution values.
- Each packet distribution f_{qi} corresponds to a velocity direction vector $\mathbf{e}_{\mathbf{qi}}$ shooting from a node to its neighbor.

$$\rho = \sum_{qi} f_{qi} \quad (15)$$

$$\overrightarrow{v} = \frac{1}{\rho} \sum_{qi} f_{qi} \overrightarrow{e_{qi}} \quad (16)$$

$$f_{qi}^{new}(\overrightarrow{x}, t) - f_{qi}(\overrightarrow{x}, t) = -\frac{1}{\tau}(f_{qi}(\overrightarrow{x}, t) - f_{qi}^{eq}(\rho, \overrightarrow{v})) \quad (17)$$

$$\begin{aligned} f_{qi}^{eq}(\rho, \overrightarrow{v}) = \rho(A_q + B_q < \overrightarrow{e_{qi}}, \overrightarrow{v} > + \\ C_q < \overrightarrow{e_{qi}}, \overrightarrow{v} >^2 + D_q < \overrightarrow{v}, \overrightarrow{v} >) \end{aligned} \quad (18)$$

LBM in Cuda

- Every step is easy to program
  - Initially read f_i from global memory but **store in shared memory**
  - Iterate and write out densities, velocities…
  - Write out only for visualization or external boundary update

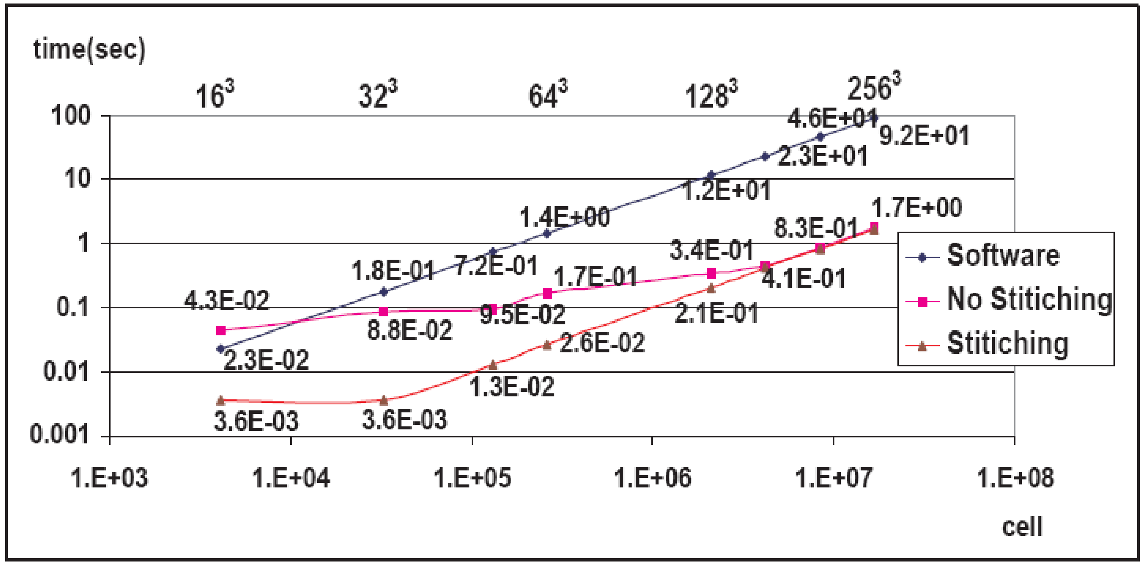
$$\rho = \sum_{qi} f_{qi} \quad (15)$$

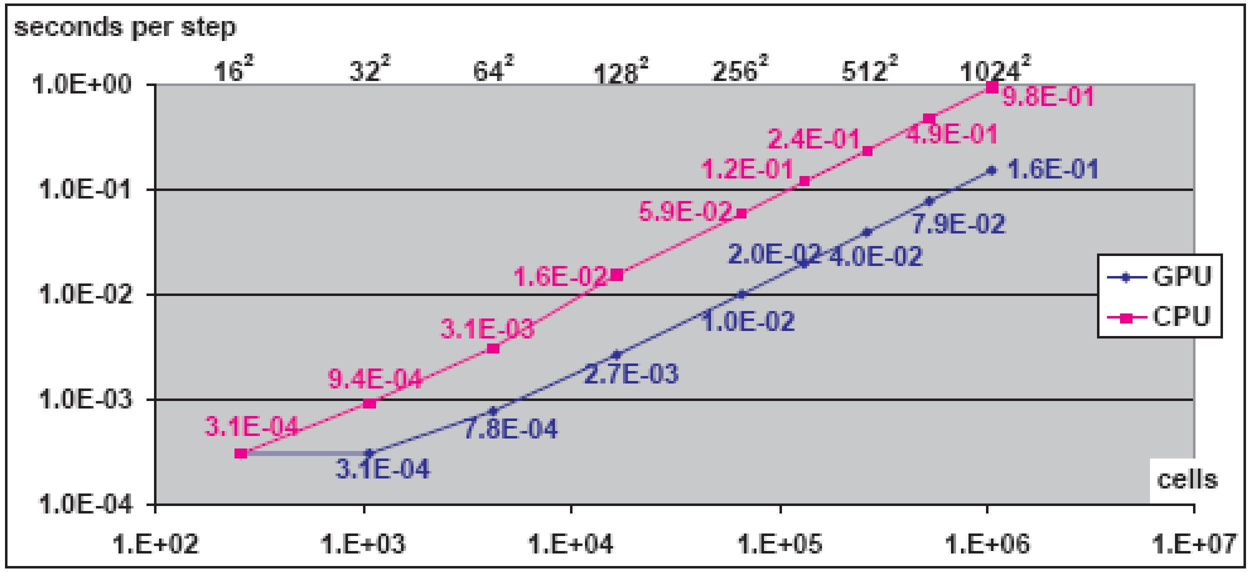
$$\overrightarrow{v} = \frac{1}{\rho} \sum_{qi} f_{qi} \overrightarrow{e_{qi}} \quad (16)$$

$$f_{qi}^{new}(\overrightarrow{x}, t) - f_{qi}(\overrightarrow{x}, t) = -\frac{1}{\tau}(f_{qi}(\overrightarrow{x}, t) - f_{qi}^{eq}(\rho, \overrightarrow{v})) \quad (17)$$

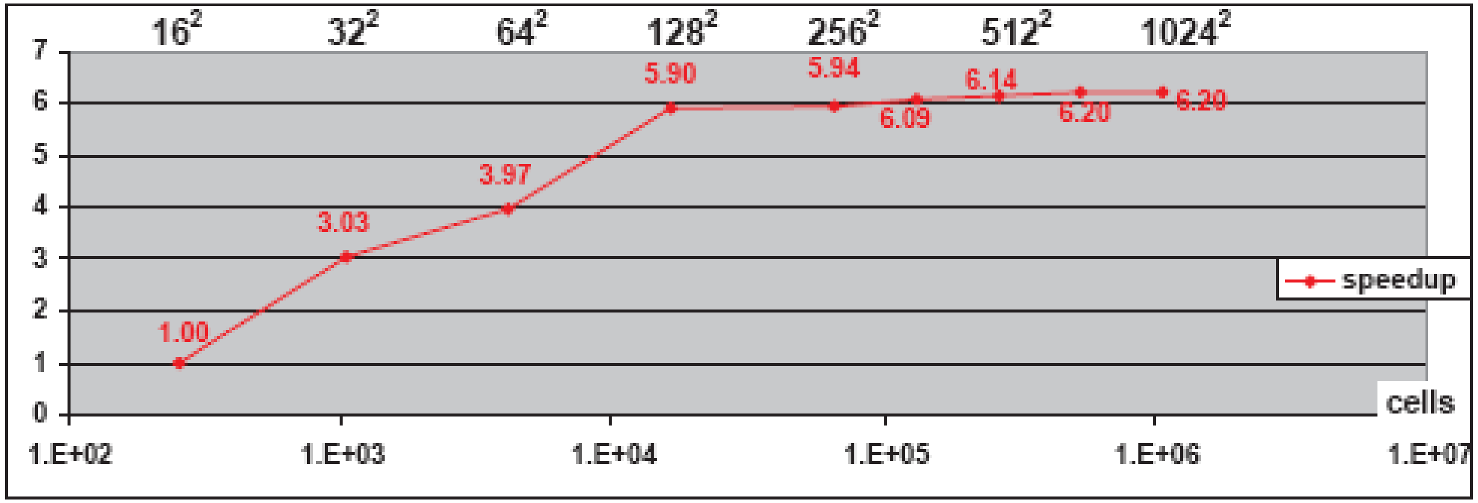
$$\begin{aligned} f_{qi}^{eq}(\rho, \overrightarrow{v}) = \rho(A_q + B_q < \overrightarrow{e_{qi}}, \overrightarrow{v} > + \\ C_q < \overrightarrow{e_{qi}}, \overrightarrow{v} >^2 + D_q < \overrightarrow{v}, \overrightarrow{v} >) \end{aligned} \quad (18)$$

Two papers from the same authors!

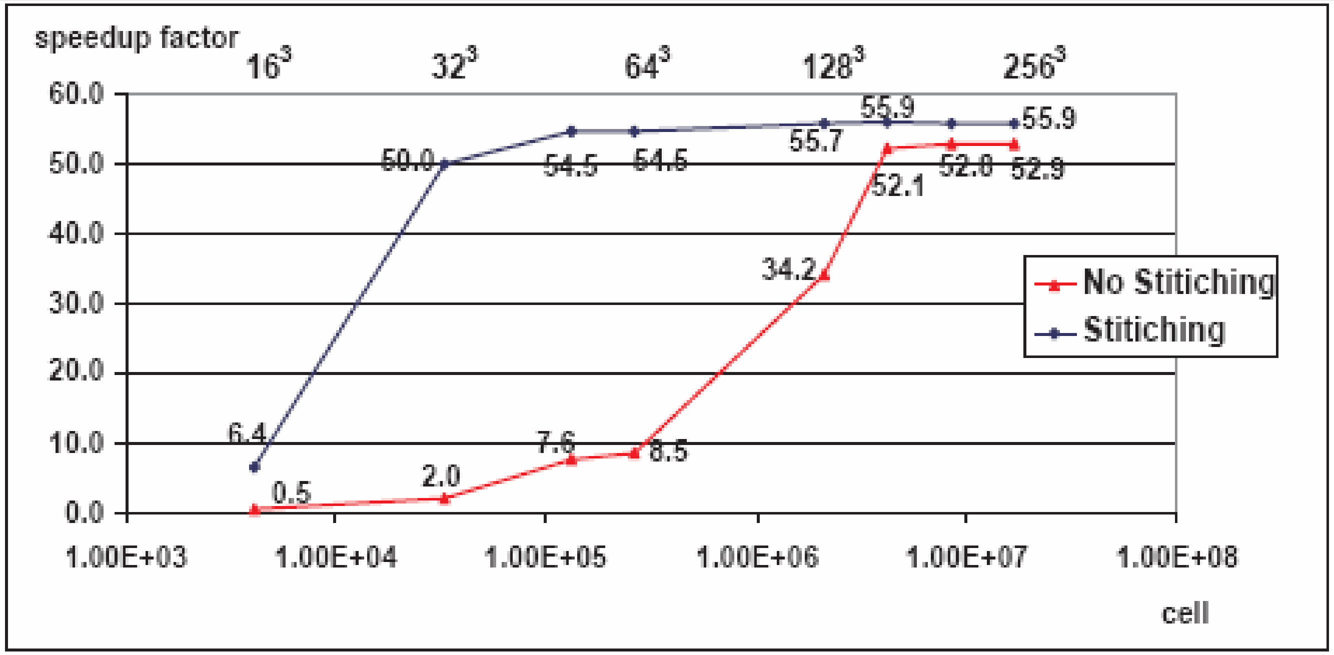




Speedup



16^2
32^2
64^2
128^2
256^2
512^2
1024^2
7
6
5
4
3
2
1
0
1.00
3.03
3.97
5.90
5.94
6.09
6.14
6.20
6.20
speedup
cells
1.E+02
1.E+03
1.E+04
1.E+05
1.E+06
1.E+07



speedup factor
16^3
32^3
64^3
128^3
256^3
60.0
50.0
40.0
30.0
20.0
10.0
0.0
6.4
50.0
54.5
54.5
55.7
55.9
55.9
0.5
2.0
7.6
8.5
34.2
52.1
52.8
52.9
No Stitiching
Stitiching
1.00E+03
1.00E+04
1.00E+05
1.00E+06
1.00E+07
1.00E+08
cell

Cellular automata

Informal definition

Cellular automaton

- A regular grid of cells, each in one of a finite number of states.
- The grid can be in any finite number of dimensions.
- Time is also discrete
  - The state of a cell at time t is a function of the states of a finite number of cells (called its *neighborhood*) at time $t - 1$.

Adapted from wikipedia

Rich behaviour from simple functions

- Example, exclusive or

$$1 \oplus 1 = 0 \oplus 0 = 0 \text{ and } 1 \oplus 0 = 0 \oplus 1 = 1$$

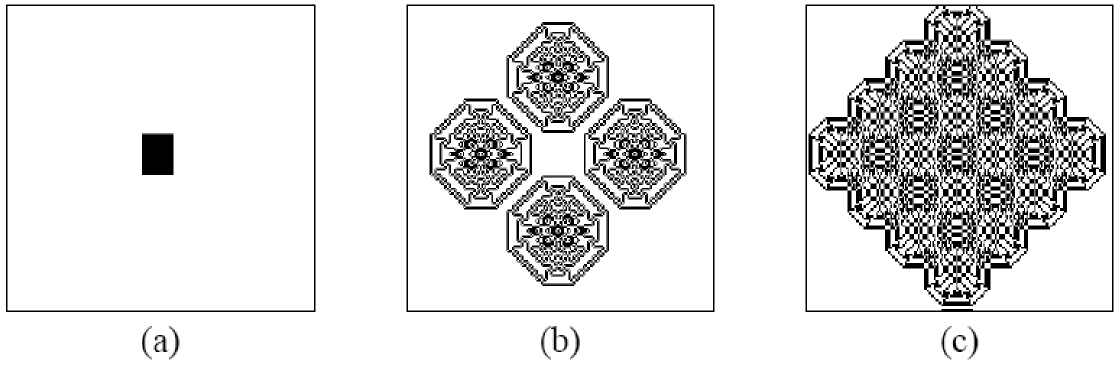


Figure 2: The $\oplus$ rule on a 256×256 periodic lattice. (a) initial configuration. (b) and (c) configurations after $t_b = 93$ and $t_c = 110$ iterations, respectively.

Courtesy Prof. Bastien Chopard

CA in CUDA: format

- If the number of possible states per cell corresponds to 2^{32}
  - integer
- No bool type on GPUs (at present)
  - Chars available
  - Represent multiple cell per primitive for best performance

Exclusive OR

Implementation considerations

CA in CUDA: global memory

- Read from and write to global memory after each iteration
- Simple and easy
  - Inefficient with respect to memory bandwidth
- **Kernel**
  - Read neighbours' states
  - Compute four-way exclusive or
  - Write result at node

CA in CUDA: textures

- Read states from a texture and write to global memory after each iteration
  - Cache hits reduces memory bandwidth
- **BUT**
  - Currently no write-to-texture support
  - Write to global memory and copy to CUDA array
  - For simple kernels the cop outweighs the cache gain

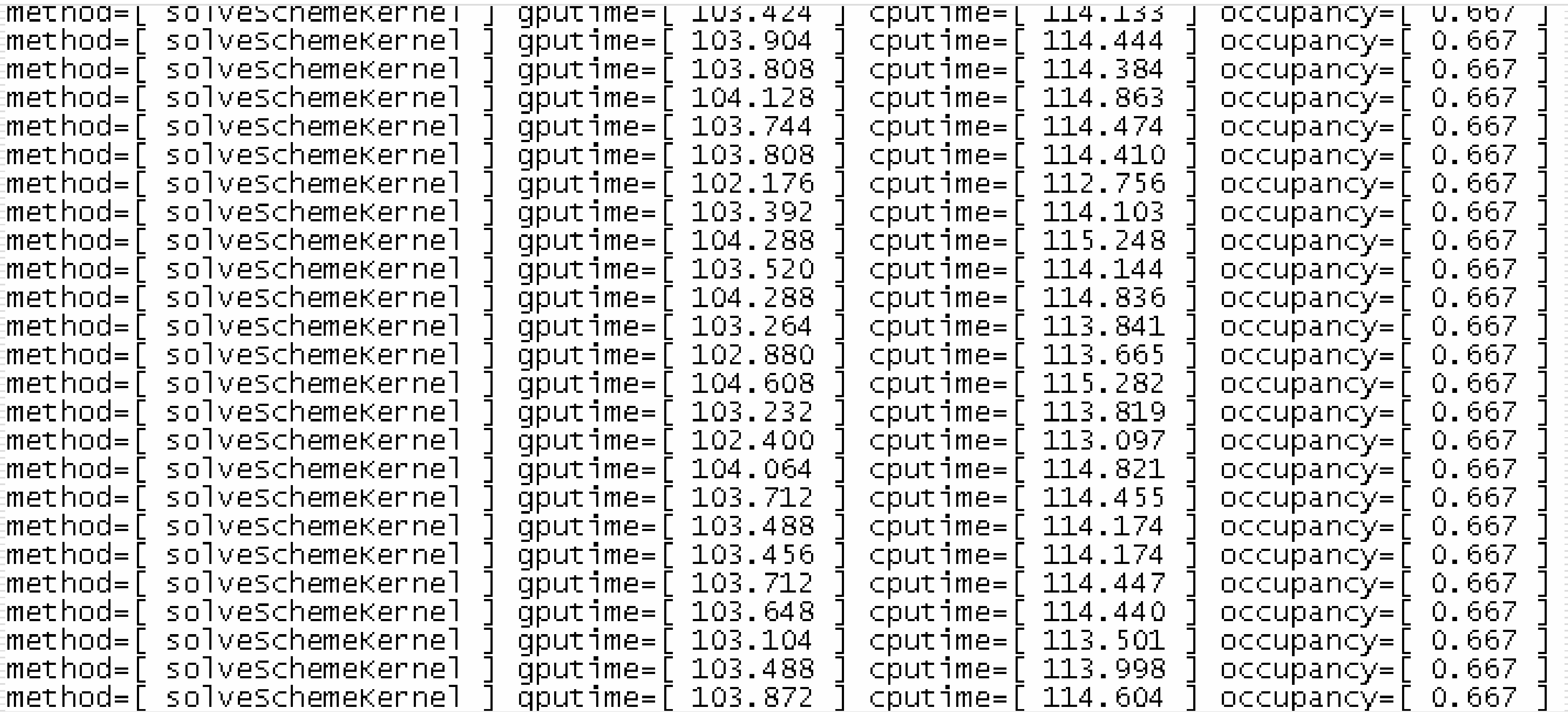
CA in CUDA: shared memory

- Read states from global memory *once* into shared memory and write to global memory after each iteration
  - Reduces bandwidth
- **Kernel**
  - Read current cell state into shared memory
    - Block border cells read also border cell state
    - Watch out for bank conflicts!
  - Synchronize threads
  - Compute four-way exclusive or from shared mem.
  - Write result at node (from register)

In my experience…

- For 256x256 grid
  - Shared memory version ~5 times faster as global memory version in a specific 2D registration problem

Cuda profiler



```
method=[ solveSchemeKernel ] gputime=[ 103.424 ] cputime=[ 114.133 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.904 ] cputime=[ 114.444 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.808 ] cputime=[ 114.384 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 104.128 ] cputime=[ 114.863 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.744 ] cputime=[ 114.474 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.808 ] cputime=[ 114.410 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 102.176 ] cputime=[ 112.756 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.392 ] cputime=[ 114.103 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 104.288 ] cputime=[ 115.248 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.520 ] cputime=[ 114.144 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 104.288 ] cputime=[ 114.836 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.264 ] cputime=[ 113.841 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 102.880 ] cputime=[ 113.665 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 104.608 ] cputime=[ 115.282 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.232 ] cputime=[ 113.819 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 102.400 ] cputime=[ 113.097 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 104.064 ] cputime=[ 114.821 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.712 ] cputime=[ 114.455 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.488 ] cputime=[ 114.174 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.456 ] cputime=[ 114.174 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.712 ] cputime=[ 114.447 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.648 ] cputime=[ 114.440 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.104 ] cputime=[ 113.501 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.488 ] cputime=[ 113.998 ] occupancy=[ 0.667 ]
method=[ solveSchemeKernel ] gputime=[ 103.872 ] cputime=[ 114.604 ] occupancy=[ 0.667 ]
```

Iterate in kernel

- If you are only interested in the result after 'n' iterations
- Iterate in kernel
  - to remove CPU overhead
  - Only border cells need to read/write global memory in each iteration
    - Communication between blocks
    - Rest of shared mem. is already known

That was it

Thank you for coming