

Linear Algebra on GPUs

Jens Krüger 
Technische Universität München



Why LA on GPUs?

1. Why should we care about Linear Algebra at all?

Use LA to solve PDEs

solving PDEs can increase realism for
VR, Education, Simulations, Games, ...

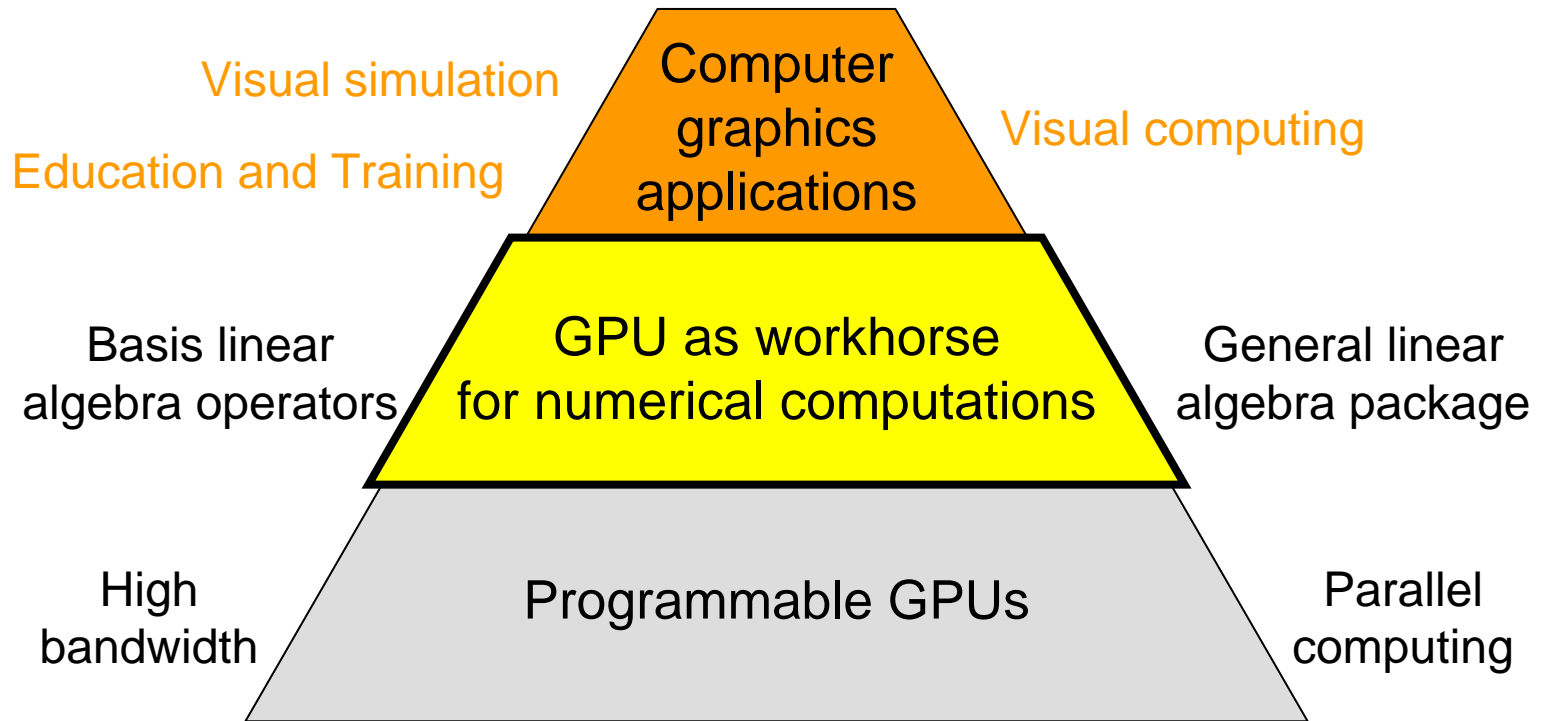


Why Linear Algebra on GPUs?

2. ... and why do it on the GPU?

- a) The GPU is a fast streaming processor
LA operations are easily “streamable”
- b) The result of the computation is already on the GPU and ready for display

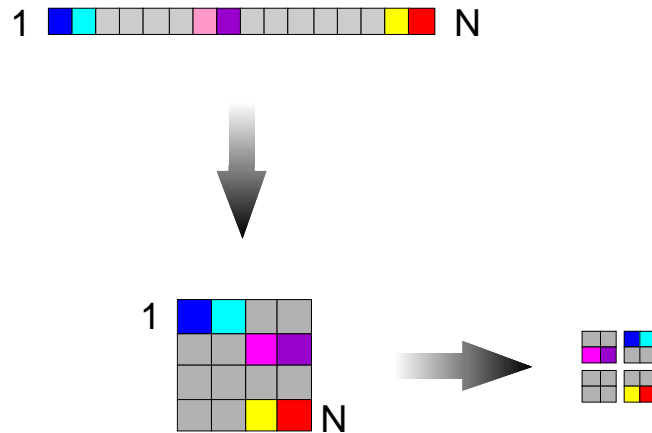
Getting started ...



Representation

Vector representation

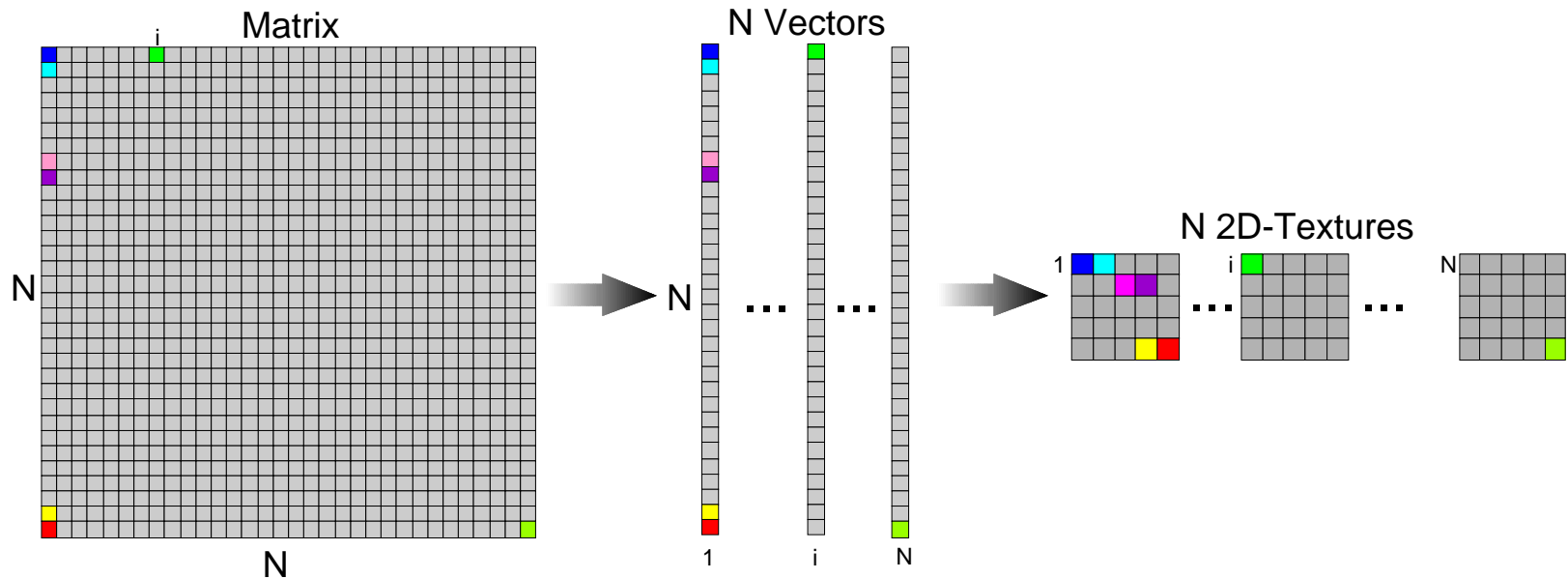
- 2D textures best we can do
 - Per-fragment vs. per-vertex operations
 - High texture memory bandwidth
 - Read-write access, dependent fetches



Representation (cont.)

Dense Matrix representation

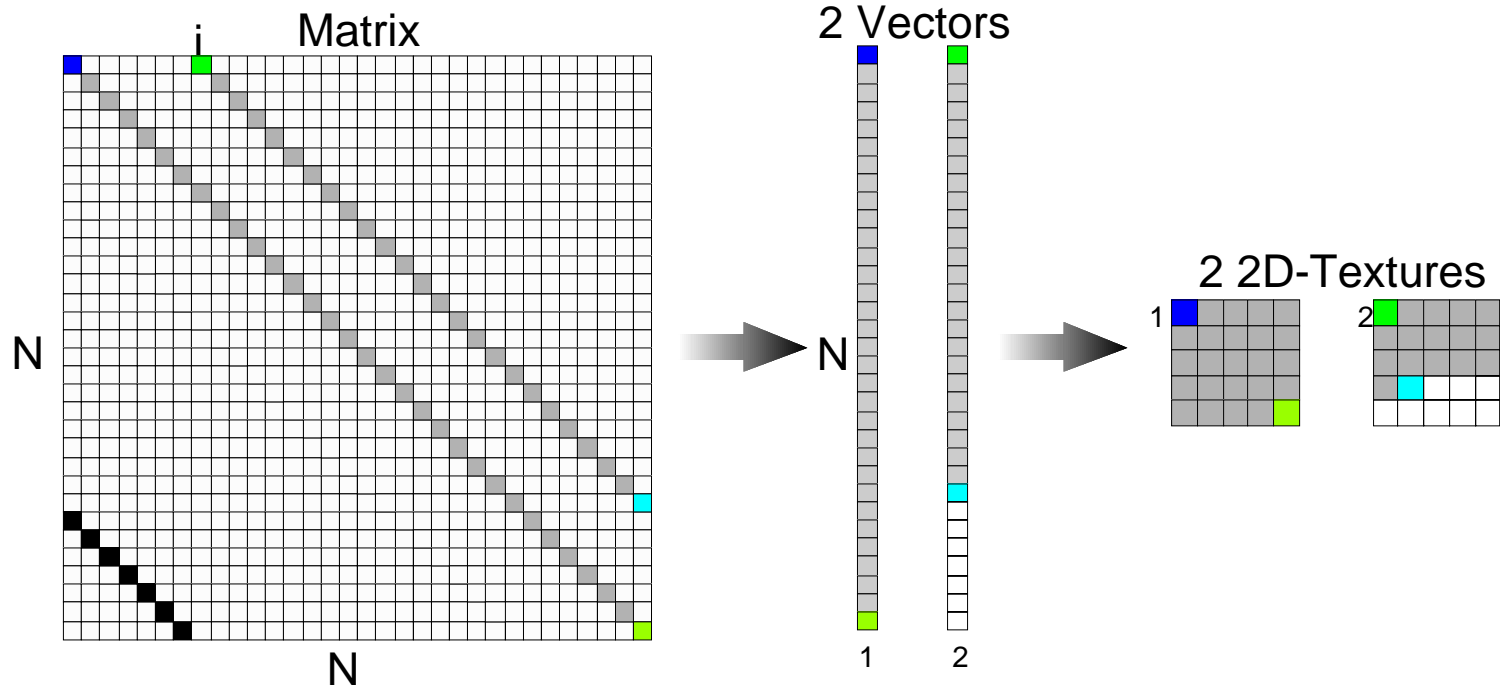
- treat a dense matrix as a set of column vectors
- again, store these vectors as 2D textures



Representation (cont.)

Banded Sparse Matrix representation

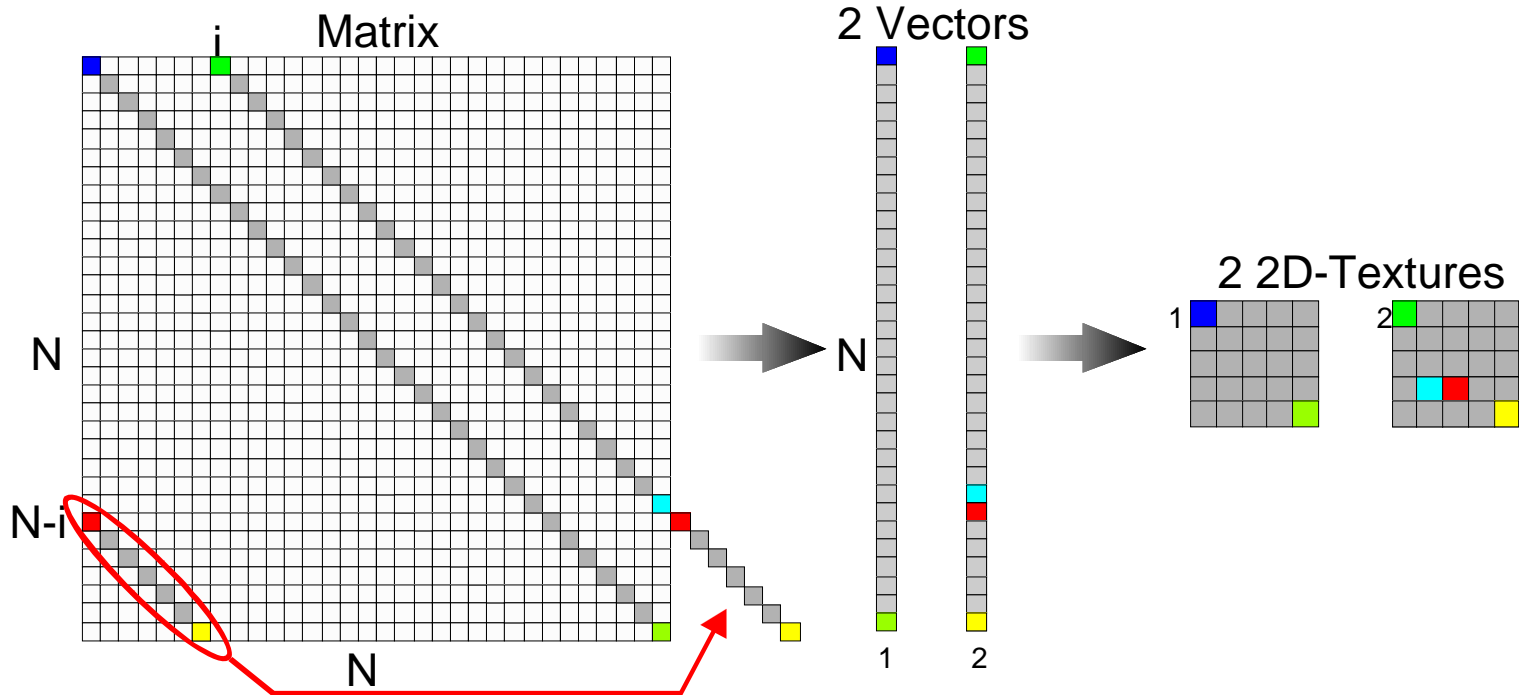
- treat a banded matrix as a set of diagonal vectors



Representation (cont.)

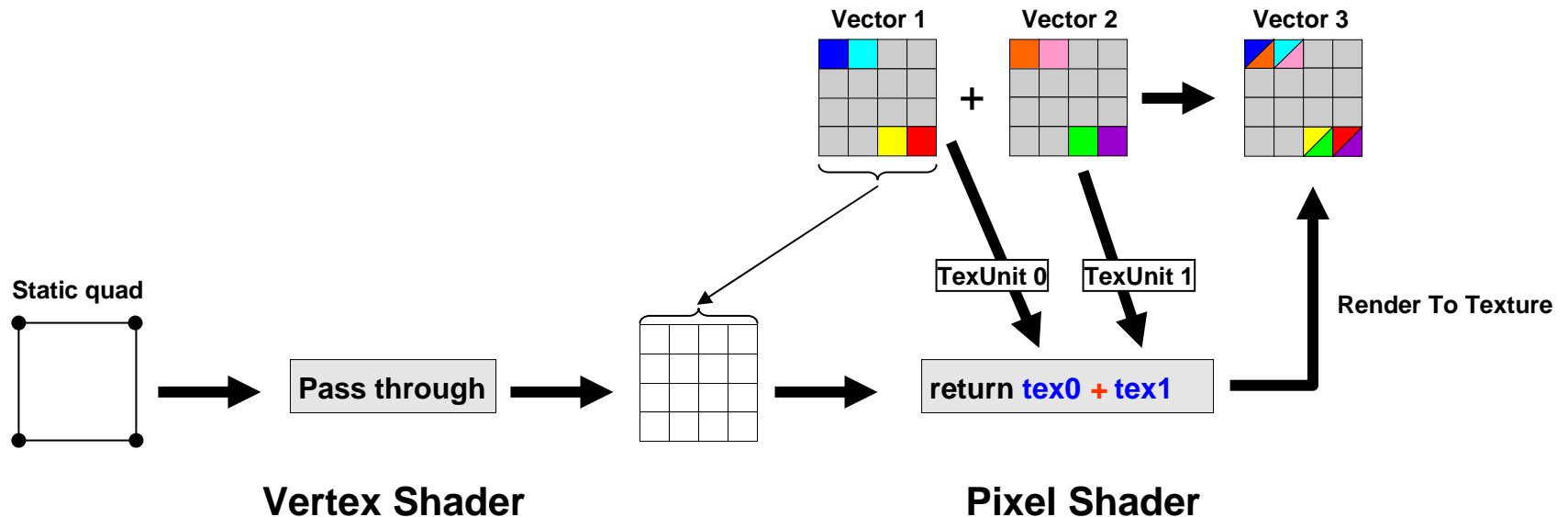
Banded Sparse Matrix representation

- combine opposing vectors to save space



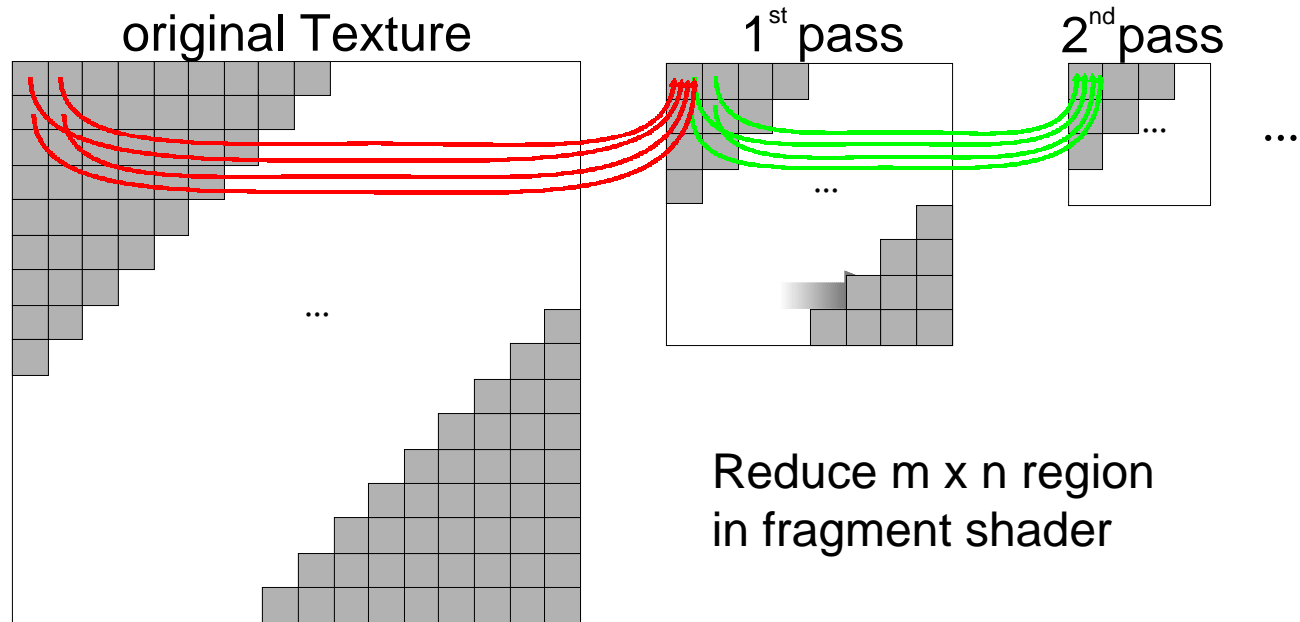
Operations

- Vector-Vector Operations
 - Reduced to 2D texture operations
 - Coded in vertex/fragment shaders
- Example: $\text{Vector1} + \text{Vector2} \rightarrow \text{Vector3}$



Operations (cont.)

- Vector-Vector Operations
 - Reduce operation for scalar products



The “single float” on GPUs

Some operations generate single float values
e.g. reduce

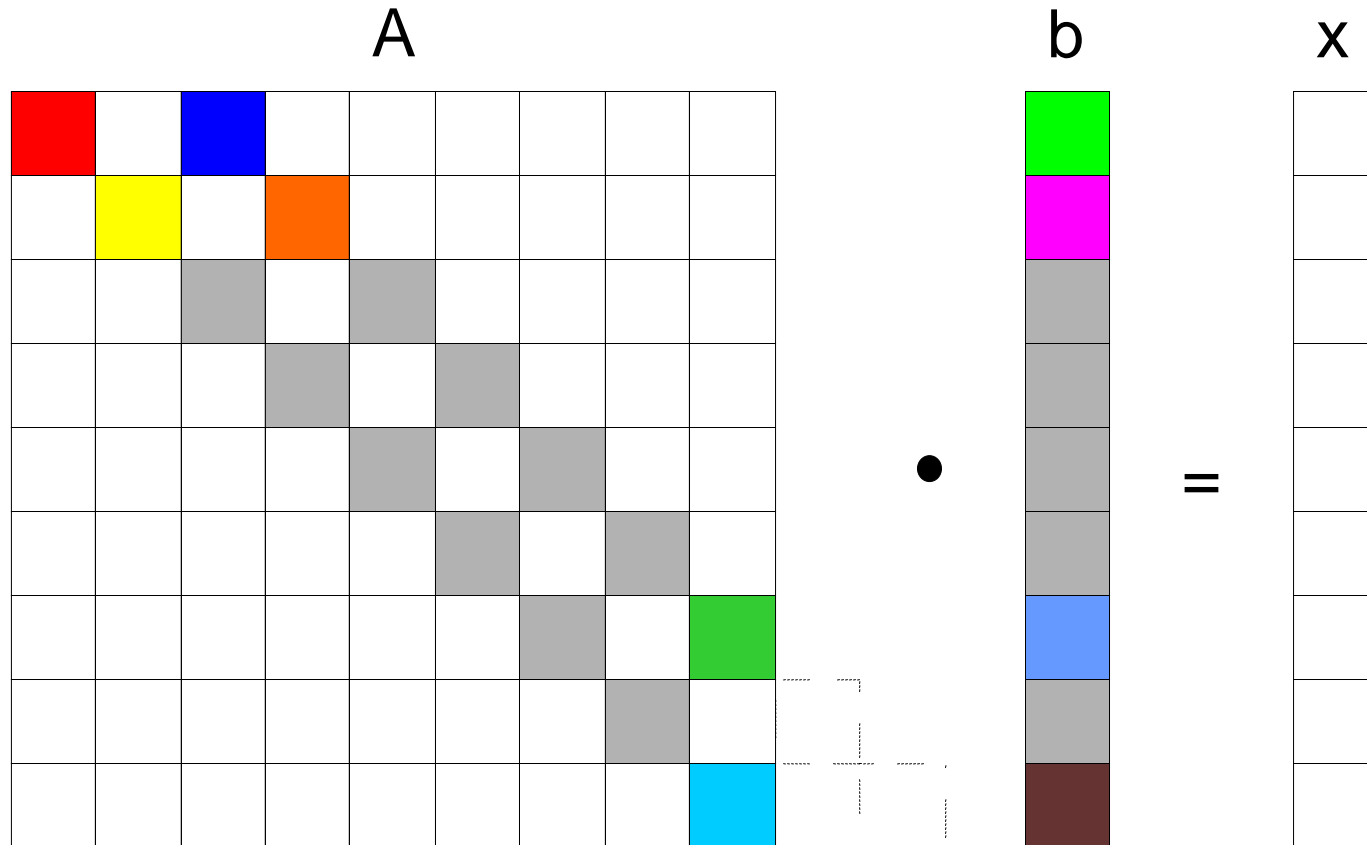


Read-back to main-mem is slow

→ Keep single floats on the GPU as 1x1 textures

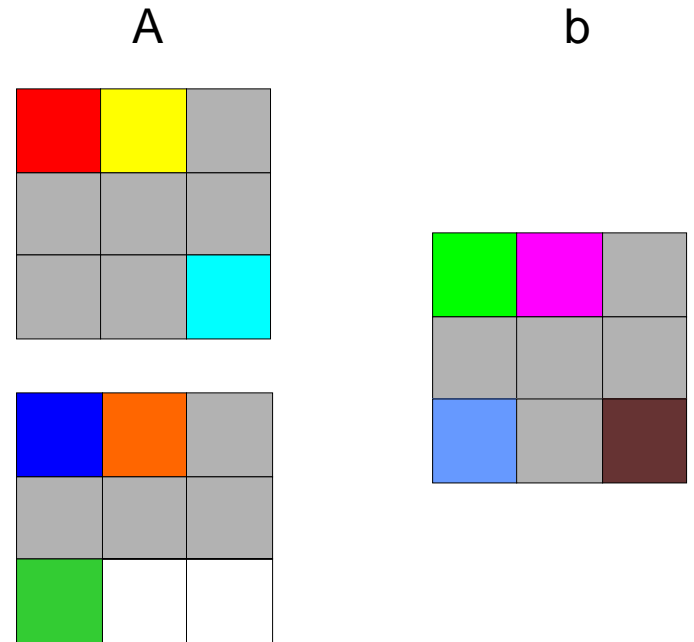
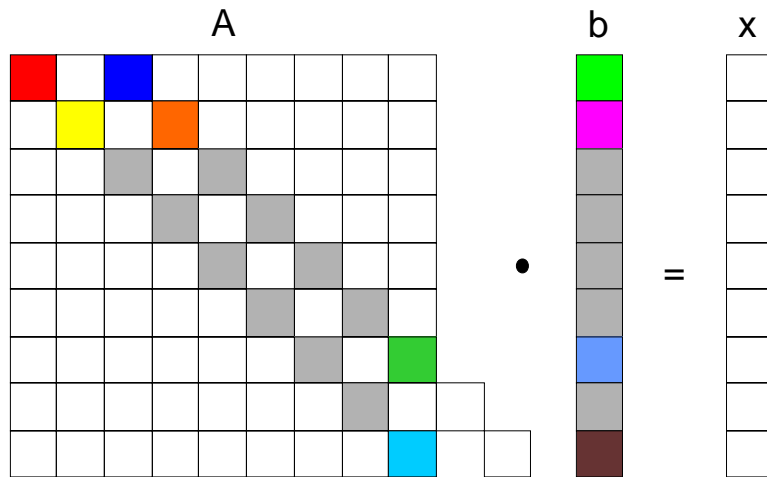
Operations (cont.)

In depth example: *Vector / Banded-Matrix Multiplication*



Example (cont.)

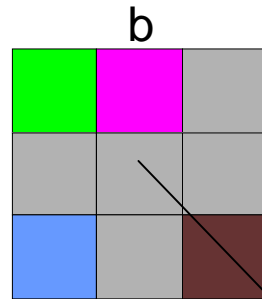
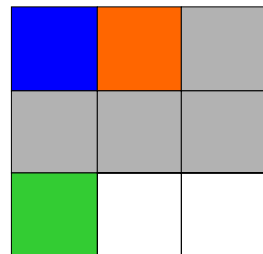
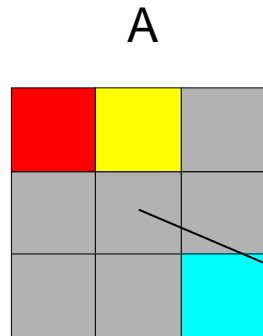
Vector / Banded-Matrix Multiplication



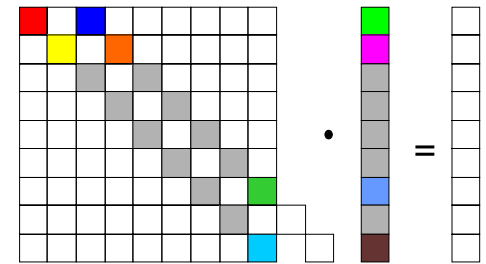
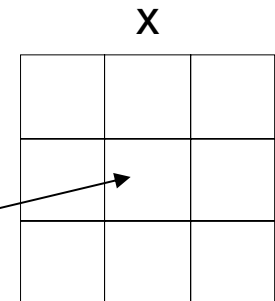
Example (cont.)

Compute the result in 2 Passes

Pass 1:



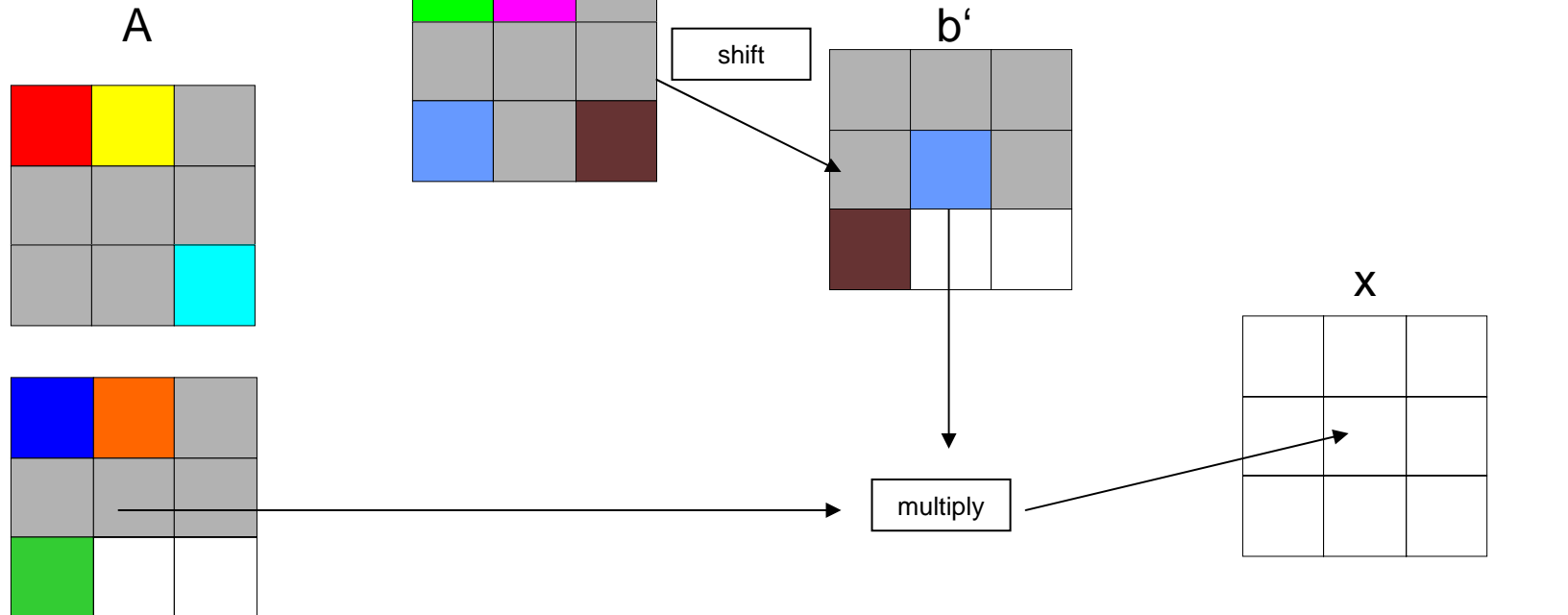
multiply



Example (cont.)

Compute the result in 2 Passes

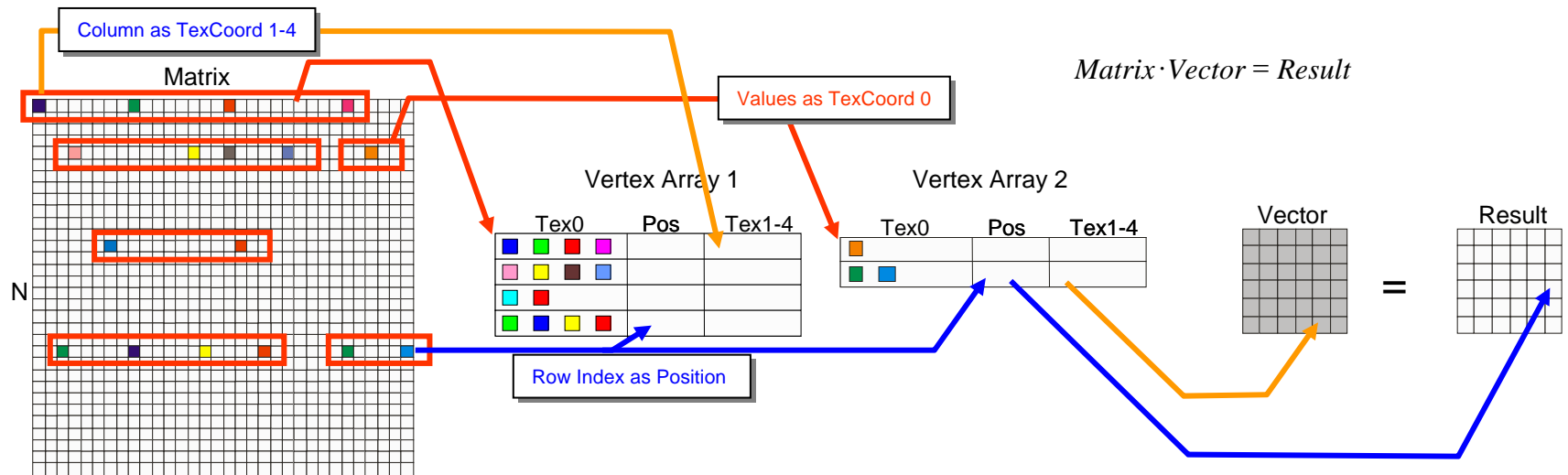
Pass 2:



Representation (cont.)

Random sparse matrix representation

- Textures do not work
 - Splitting yields highly fragmented textures
 - Difficult to find optimal partitions
- Idea: encode only non-zero entries in vertex arrays



Building a Framework

Presented so far:

- representations on the GPU for
 - single float values
 - vectors
 - matrices
 - dense
 - banded
 - random sparse
- operations on these representations
 - add, multiply, reduce, ...
 - upload, download, clear, clone, ...

Building a Framework Example: CG

Encapsulate into Classes for more complex algorithms

- Example use: Conjugate Gradient Method, complete source:

```
void clCGSolver::solveInit() {
    Matrix->matrixVectorOp(CL_SUB,X,B,R);      // R = A*x-b
    R->multiply(-1);                          // R = -R
    R->clone(P);                              // P = R
    R->reduceAdd(R, Rho);                      // rho = sum(R*R);
}

void clCGSolver::solveIteration() {
    Matrix->matrixVectorOp(CL_NULL,P,NULL,Q); // Q = Ap;
    P->reduceAdd(Q,Temp);                     // temp = sum(P*Q);
    Rho->div(Temp,Alpha);                     // alpha = rho/temp;
    X->addVector(P,X,1,Alpha);                 // X = X + alpha*P
    R->subtractVector(Q,R,1,Alpha);            // R = R - alpha*Q
    R->reduceAdd(R,NewRho);                    // newrho = sum(R*R);
    NewRho->divZ(Rho,Beta);                    // beta = newrho/rho
    R->addVector(P,P,1,Beta);                  // P = R+beta*P;
    clFloat *temp; temp=NewRho;
    NewRho=Rho; Rho=temp;                    // swap rho and newrho pointers
}

void clCGSolver::solve(int maxI) {
    solveInit();
    for (int i = 0;i< maxI;i++) solveIteration();
}

int clCGSolver::solve(float rhoTresh, int maxI) {
    solveInit(); Rho->clone(NewRho);
    for (int i = 0;i< maxI && NewRho.getData() > rhoTresh;i++) solveIteration();
    return i;
}
```

Building a Framework

Example: Jacobi

The Jacobi method for a problem

$$A \cdot x = b \quad \text{where} \quad A = L + D + U$$

can be expressed with matrices as follows:

$$x^{(k+1)} = D^{-1} \cdot \left(b - (L + U) \cdot x^{(k)} \right)$$

If the Matrix is stored in the diagonal form the matrix D^{-1} can be computed on the fly by inverting the diagonal elements during the vector-vector product. So one Jacobi step becomes one matrix-vector product, one vector-vector product and one vector subtract.

Example 1: 2D Waves (explicit)

$$\frac{\partial^2 \mathbf{x}}{\partial t^2} = c^2 \left(\frac{\partial^2 \mathbf{x}}{\partial u^2} + \frac{\partial^2 \mathbf{x}}{\partial v^2} \right)$$

Finite difference discretization:

$$\mathbf{x}_{i,j}^{t+1} = \beta \cdot (\mathbf{x}_{i-1,j}^t + \mathbf{x}_{i,j-1}^t + \mathbf{x}_{i+1,j}^t + \mathbf{x}_{i,j+1}^t) + (2 - 4\beta) \cdot \mathbf{x}_{i,j}^t - \mathbf{x}_{i,j}^{t-1}$$

Instead of writing a custom shader for this filter think about this as a matrix-vector operation

2-4β	β		β							x_1^t	x_1^{t-1}	x_1^{t+1}
β	2-4β	β		β						x_2^t	x_2^{t-1}	x_2^{t+1}
	β	2-4β			β					x_3^t	x_3^{t-1}	x_3^{t+1}
β			2-4β	β		β				x_4^t	x_4^{t-1}	x_4^{t+1}
	β		β	2-4β	β		β			x_5^t	x_5^{t-1}	x_5^{t+1}
		β		β	2-4β			β		x_6^t	x_6^{t-1}	x_6^{t+1}
			β			2-4β	β			x_7^t	x_7^{t-1}	x_7^{t+1}
				β		β	2-4β	β		x_8^t	x_8^{t-1}	x_8^{t+1}
					β		β	2-4β		x_9^t	x_9^{t-1}	x_9^{t+1}

Example 2: 2D Waves (implicit)

2D wave equation

- Finite difference discretization
- Implicit Crank-Nicholson scheme

Key Idea: Rewrite as Matrix-Vector Product

Example 2: 2D Waves (implicit)

$4\alpha+1$	$-\alpha$		$-\alpha$					
$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$				
	$-\alpha$	$4\alpha+1$			$-\alpha$			
$-\alpha$			$4\alpha+1$	$-\alpha$		$-\alpha$		
	$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$		$-\alpha$	
		$-\alpha$		$-\alpha$	$4\alpha+1$			$-\alpha$
			$-\alpha$			$4\alpha+1$	$-\alpha$	
				$-\alpha$		$-\alpha$	$4\alpha+1$	$-\alpha$
					$-\alpha$		$-\alpha$	$4\alpha+1$

•

x_1^{t+1}
x_2^{t+1}
x_3^{t+1}
x_4^{t+1}
x_5^{t+1}
x_6^{t+1}
x_7^{t+1}
x_8^{t+1}
x_9^{t+1}

=

c_1^t
c_2^t
c_3^t
c_4^t
c_5^t
c_6^t
c_7^t
c_7^t
c_9^t

$$c_i^t = \alpha \cdot (x_{i-1,j}^t + x_{i,j-1}^t + x_{i+1,j}^t + x_{i,j+1}^t) + (2 - 4\alpha) \cdot x_{i,j}^t - x_{i,j}^{t-1}$$

$$\text{where } \alpha = \frac{\Delta t^2 \cdot c^2}{2 \cdot \Delta h^2}$$

Navier-Stokes on GPUs

The Equations

The Navier-Stokes Equations for 2D:

$$\frac{\delta u}{\delta t} = \frac{1}{\text{Re}} \nabla^2 u - V \cdot \nabla u + f_x - \frac{\delta p}{\delta x} \quad \text{Advection}$$

$$\frac{\delta v}{\delta t} = \frac{1}{\text{Re}} \nabla^2 v - V \cdot \nabla v + f_y - \frac{\delta p}{\delta y}$$

Diffusion

$$\text{div}(V) = 0$$

Zero Divergence

External Forces

Pressure Gradient

NSE Discretization

$$\frac{\partial v}{\partial t} = \frac{1}{\text{Re}} \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right) - \frac{\partial(uv)}{\partial x} - \frac{\partial(v^2)}{\partial y} + f_y - \frac{\partial p}{\partial y}$$

Diffusion

$$\left[\frac{\partial^2 v}{\partial x^2} \right]_{i,j} = \frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{(\delta x)^2}$$

$$\left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} = \frac{v_{i,j+1} - 2v_{i,j} + v_{i,j-1}}{(\delta y)^2}$$

Advection

$$\left[\frac{\partial(uv)}{\partial x} \right]_{i,j} = \frac{1}{\delta x} \left(\frac{(u_{i,j} + u_{i,j+1})(v_{i,j} + v_{i+1,j})}{2} - \frac{(u_{i-1,j} + u_{i-1,j+1})(v_{i-1,j} + v_{i,j})}{2} \right) + \alpha \frac{1}{\delta x} \left(\frac{|u_{i,j} + u_{i,j+1}|(v_{i,j} - v_{i+1,j})}{2} - \frac{(u_{i-1,j} + u_{i-1,j+1})(v_{i-1,j} - v_{i,j})}{2} \right)$$

Pressure

$$\left[\frac{\partial p}{\partial y} \right]_{i,j} = \frac{p_{i,j+1} - p_{i,j}}{\delta y}$$

Navier-Stokes Equations (cont.)

Rewrite the Navier Stokes Equations

$$u_{i,j}^{(t+1)} = F_{i,j}^{(t)} - \frac{\delta t}{\delta x} (p_{i+1,j}^{(t+1)} - p_{i,j}^{(t+1)}) \quad v_{i,j}^{(t+1)} = G_{i,j}^{(t)} - \frac{\delta t}{\delta y} (p_{i,j+1}^{(t+1)} - p_{i,j}^{(t+1)})$$

where

$$F_{i,j} = u_{i,j} + \delta t \left(\frac{1}{\text{Re}} \left(\left[\frac{\partial^2 u}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 u}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial(u^2)}{\partial x} \right]_{i,j} - \left[\frac{\partial(uv)}{\partial y} \right]_{i,j} + f_x \right);$$

$$G_{i,j} = v_{i,j} + \delta t \left(\frac{1}{\text{Re}} \left(\left[\frac{\partial^2 v}{\partial x^2} \right]_{i,j} + \left[\frac{\partial^2 v}{\partial y^2} \right]_{i,j} \right) - \left[\frac{\partial(uv)}{\partial x} \right]_{i,j} - \left[\frac{\partial(v^2)}{\partial y} \right]_{i,j} + f_y \right);$$

now F and G can be computed

Navier-Stokes Equations (cont.)

Problem: Pressure is still unknown!

$$u_{i,j}^{(t+1)} = F_{i,j}^{(t)} - \frac{\delta t}{\delta x} (p_{i+1,j}^{(t+1)} - p_{i,j}^{(t+1)}) \quad v_{i,j}^{(t+1)} = G_{i,j}^{(t)} - \frac{\delta t}{\delta y} (p_{i,j+1}^{(t+1)} - p_{i,j}^{(t+1)})$$

From $\text{div}(V) = 0$ derive:

$$0 = \frac{\partial u^{t+1}}{\partial x} + \frac{\partial v^{t+1}}{\partial y} = \frac{\partial G^t}{\partial x} - \delta t \frac{\partial^2 p^{t+1}}{\partial x^2} + \frac{\partial F^t}{\partial y} - \delta t \frac{\partial^2 p^{t+1}}{\partial y^2}$$

...to get this Poisson Equation:

$$\frac{p_{i+1,j}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i-1,j}^{(n+1)}}{(\delta x)^2} + \frac{p_{i,j+1}^{(n+1)} - 2p_{i,j}^{(n+1)} + p_{i,j-1}^{(n+1)}}{(\delta y)^2} = \frac{1}{\delta t} \left(\frac{F_{i,j}^{(n)} - F_{i-1,j}^{(n)}}{\delta x} + \frac{G_{i,j}^{(n)} - G_{i,j-1}^{(n)}}{\delta y} \right)$$

Navier-Stokes Equations (cont.)

The basic algorithm:

1. Compute F and G

1. add external forces
2. advect
3. diffuse

easy 😊

semi-lagrange [Stam 1999]

explicit

2. solve the Poisson equation

use the CG solver

3. update velocities

subtract pressure
gradient

Multigrid on GPUs

Multigrid “in English”

1. do a few Jacobi/Gauss-Seidel iterations on the fine grid
 - Jacobi/G-S eliminate high frequencies in the error
 - conjugate gradient does not have this property !!!
2. compute the residuum of the last approximation
3. propagate this residuum to the next coarser grid
 - can be done by the means of a matrix multiplication
4. solve the coarser grid for the absolute error
 - for the solution you can use another multigrid step
5. backpropagate the error to the finer grid
 - can be done by another matrix multiplication (transposed matrix from 3.)
6. use the error to correct the first approximation
7. do another few Jacobi/Gauss-Seidel iterations to remove noise introduced by the propagation steps

Multigrid “in Greek”

$$A^h \cdot x^h = b^h$$

$$A^h \cdot (x'^h + e^h) = b^h$$

$$A^h \cdot e^h = \underbrace{b^h - A^h \cdot x'^h}_{r^h}$$

$$I_h^{2h} \cdot A^h \cdot \underbrace{e^h}_{(I_h^{2h})^T \cdot e^{2h}} = I_h^{2h} \cdot r^h$$

$$\left(I_h^{2h} \cdot A^h \cdot (I_h^{2h})^T \right) \cdot e^{2h} = r^{2h}$$

$$A^{2h} \cdot e^{2h} = r^{2h}$$

- Consider this problem: x is the solution vector of a set of linear equations
- this equation holds for current approximation x' with the error e
- rearranging leads to the residual equation with residuum r
- now multiply both sides with a non quadratic interpolation matrix and replace the error with an error times the transposed interpolation matrix
- Let A^{2h} be the product of the Interpolation Matrix, A and the transposed Interpolation matrix.
- Finally we end up with a new set of linear equations with only half the size in every dimension of the old one. We solve this set for the error at this grid level. Propagating the error the above steps we can derive the error for the large system and use it to correct out approximation

Multigrid (cont.)

- “fine grid” , “coarse grid” only makes sense if the problem to solve corresponds to a grid ☹
 - this is the case in the finite difference methods described before ☺
 - need to find an “interpolation matrix” for the propagation step to generate the coarser grid (for instance simple linear interpolation)
 - need an “extrapolation matrix” to move from the coarse to the fine grid
- the coarse grid matrices can be pre-computed

Multigrid on GPUs

Observation:

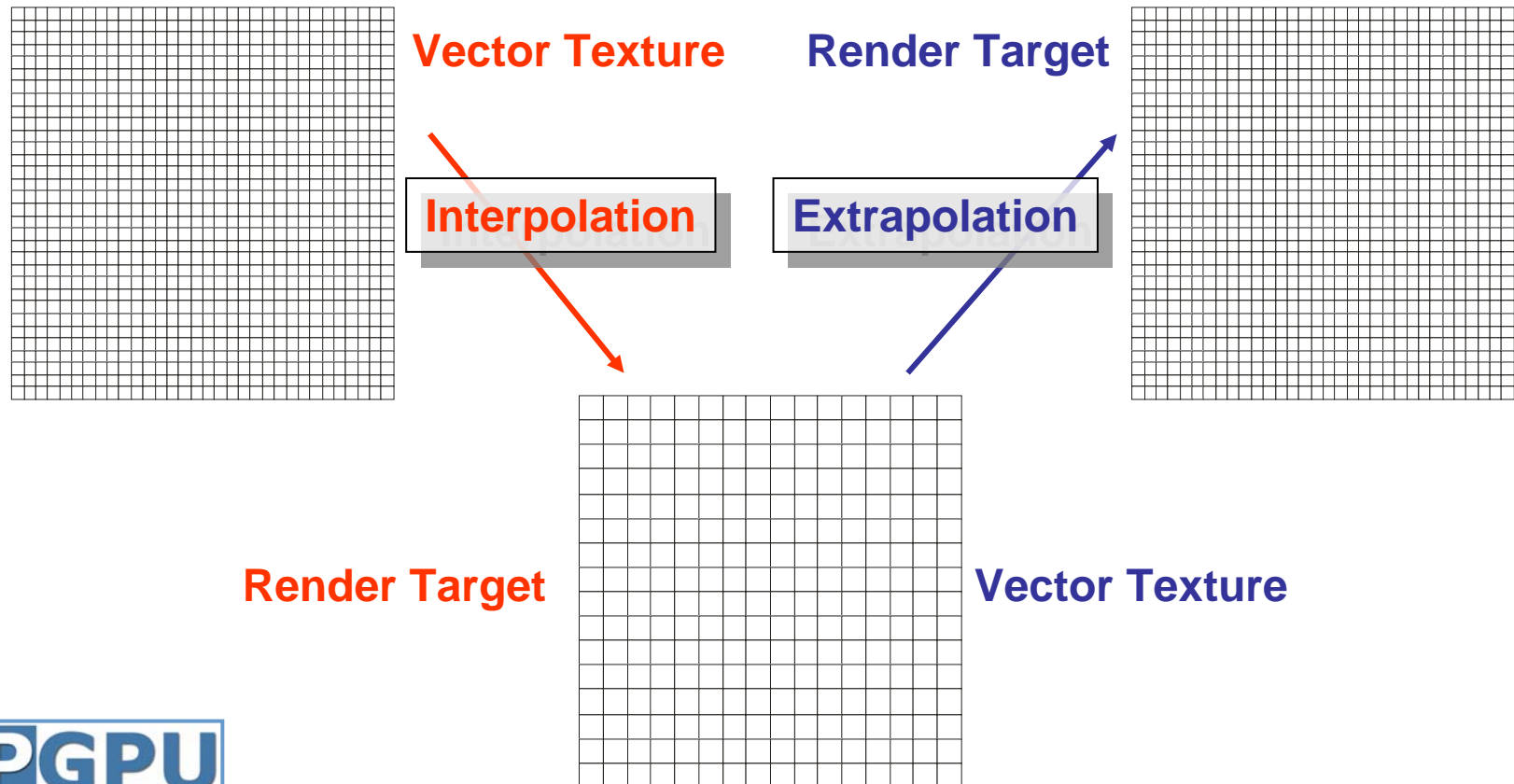
- you only need matrix-vector operations and a “Jacobi smoother” to do multigrid
- to put it on the GPU simply use the matrix-vector operations from the framework

Improvement:

- to do the interpolation and extrapolation steps we can use the fast bilinear interpolation hardware of the GPU instead of a vector-matrix multiplication

Multigrid on GPUs (cont.)

We can embed the new “rescale” easily into the vector representation by simply rendering one textured quad.



Selected References

- Chorin, A.J., Marsden, J.E. *A Mathematical Introduction to Fluid Mechanics*. 3rd ed. Springer. New York, 1993
- Briggs, Henson, McCormick *A Multigrid Tutorial*, 2nd ed. siam, ISBN 0-89871-462-1
- Acton *Numerical Methods that Work*, The Mathematical Association of America ISBN 0-88385-450-3
- Krüger, J. Westermann, R. Linear algebra operators for GPU implementation of numerical algorithms, In Proceedings of SIGGRAPH 2003, ACM Press / ACM SIGGRAPH, <http://wwwwcg.in.tum.de/Research/Publications>
- Bolz, J., Farmer, I., Grinspun, E., Schröder, P. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid, In Proceedings of SIGGRAPH 2003, ACM Press / ACM SIGGRAPH, <http://www.multires.caltech.edu/pubs/>
- Hillesland, K. E. Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware, In Proceedings of SIGGRAPH 2003, ACM Press / ACM SIGGRAPH, <http://www.cs.unc.edu/~khillesl/nlopt/>
- Fedkiw, R., Stam, J. and Jensen, H.W. Visual Simulation of Smoke. In Proceedings of SIGGRAPH 2001, ACM Press / ACM SIGGRAPH. 2001, <http://graphics.ucsd.edu/~henrik/papers/smoke/>
- Stam, J. Stable Fluids. In Proceedings of SIGGRAPH 1999, ACM Press / ACM SIGGRAPH, 121-128. 1999, <http://www.dgp.toronto.edu/people/stam/reality/Research/pub.html>
- Harris, M., Coombe, G., Scheuermann, T., and Lastra, A. Physically-Based Visual Simulation on Graphics Hardware.. Proc. 2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware 2002, <http://www.markmark.net/cml/>