Linear algebra - Solutions to linear systems

Thomas Sangild
GPGPU lecture
University of Aarhus
25/11-2004

References

- Handouts
  - Krüger J and Westermann R. Linear algebra Operators for GPU implementation of Numerical Algorithms. Siggraph 2003.
  - Bolz J, Farmer I, Grinspun E, and Schröder P. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. Siggraph 2003.

Function headers

```
enum CL_enum_op = { CL_ADD, CL_MULT, CL_SUB };

void clVecOp(
    CL_enum_op op, float α, float β, clVec x, clVec y, clVec res
);

void clMatVec (
    CL_enum_op op, clMat A, clVec x, clVec y, clVec res
);

enum CL_enum_cmb = { CL_ADD, CL_MULT, CL_MAX, CL_MIN };

float clVecReduce( CL_enum_cmb cmb, clVec x, clVec y );
void clVecReduce( CL_enum_cmb cmb, clVec x, clVec y, clMat res );
```

1x1 matrix

We seek representations of

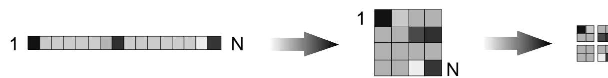
- Vectors
- Matrices
  - Dense
  - Sparse
    - Unstructured
    - Structured

Representation

Vector representation

Textures best we can do

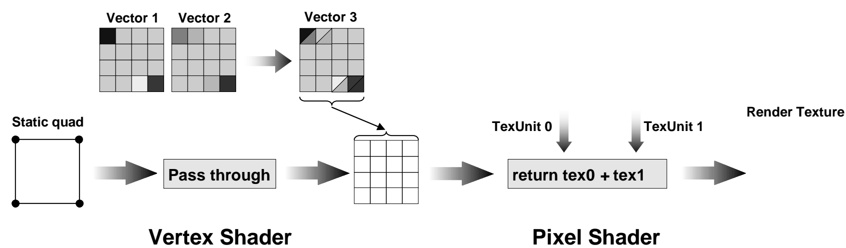
- 2D RGBA textures really rock
  - Per-pixel vs. per-vertex operations
    - 6 gigapixels/second vs. 0.7 gigavertices/second
    - Efficient texture memory cache
    - Texture read-write access



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Operations

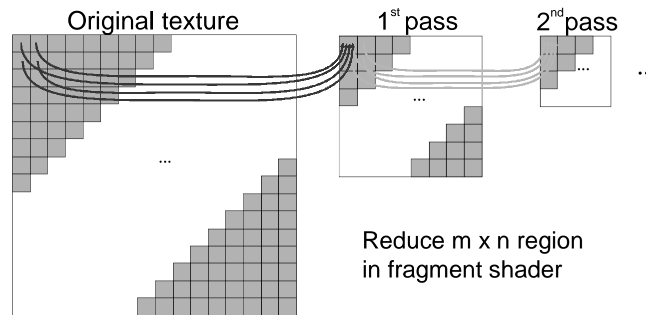
- **Vector-Vector** Operations
  - Reduced to 2D texture operations
  - Coded in pixel shaders
- Example: *Vector1 + Vector2 → Vector3*



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Operations (reduce)

Reduce operation for scalar products



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

The "single float" on GPUs

Some operations generate single float values
e.g. reduce

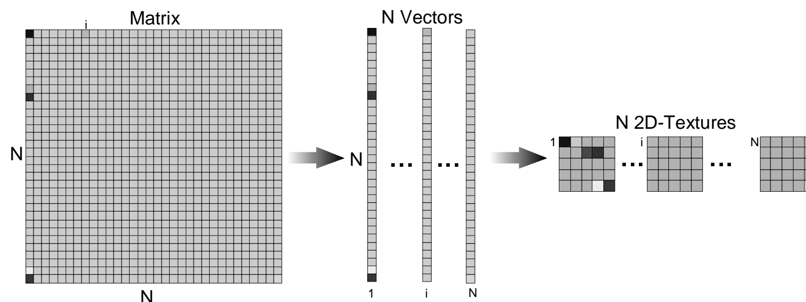
Read-back to main memory is slow
→ Keep single floats on the GPU as 1x1 textures

Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Representation (cont.)

Dense Matrix representation

- Treat a dense matrix as a set of column vectors
- Again, store these vectors as 2D textures

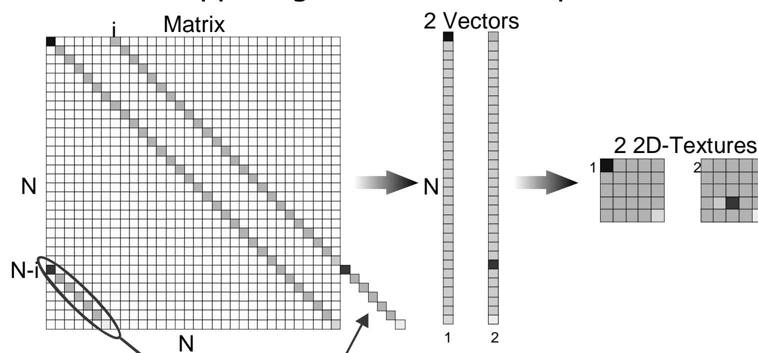


Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Representation (cont.)

Banded Sparse Matrix representation

- Treat a banded matrix as a set of **diagonal** vectors
- Combine opposing vectors to save space

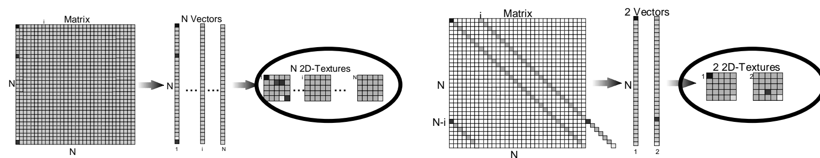


Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Operations (cont.)

Matrix-Vector Operations

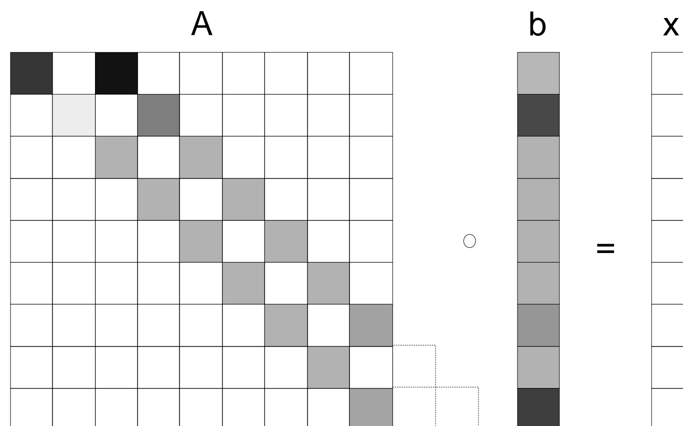
- Split it up into Vector – Vector operations



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Operations

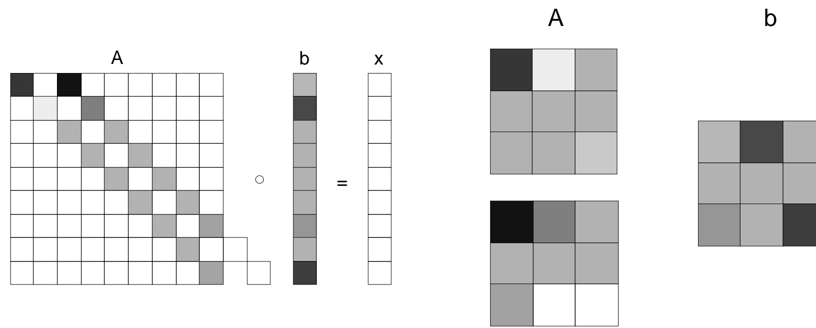
In depth example: *Vector / Banded-Matrix Multiplication*



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Example (cont.)

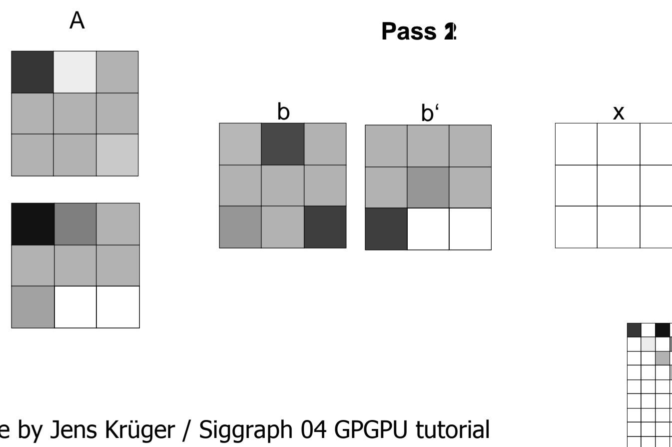
Vector / Banded-Matrix Multiplication



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Example (cont.)

Compute the result in 2 Passes:



Slide by Jens Krüger / Siggraph 04 GPGPU tutorial

Representation (cont.)

Unstructured Sparse Matrix representation

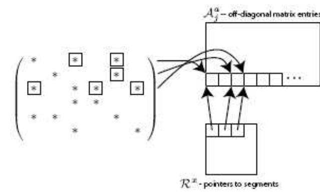


Figure 1: *Off-diagonal elements of each row are compacted into segments which are tightly packed into $\mathcal{A}^a_j$. A pointer to the beginning of each segment is stored in $\mathcal{R}^x$.*

$$j = \mathcal{R}^x[i]$$
$$\mathcal{Y}^x[i] = \mathcal{A}^x_i[i] * \mathcal{X}^x[i] + \sum_{c=0}^{k_i-1} \mathcal{A}^a_j[j+c] * \mathcal{X}^x[\mathcal{C}^a[j+c]],$$

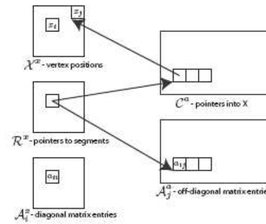


Figure 2: *When the fragment program executes on the pixel corresponding to row i, the window position is used as a texture coordinate to fetch x_i in $\mathcal{X}^x$ and a_{ii} in $\mathcal{A}^x_i$. The window position also identifies the segment pointer in $\mathcal{R}^x$, which points to the location of non-zero elements a_{ij} in $\mathcal{A}^a_j$ corresponding to row i. Finally, using the segment pointer from $\mathcal{R}^x$ we can access entries in $\mathcal{C}^a$ which reveal the addresses of x_j in $\mathcal{X}^x$ corresponding to non-zero a_{ij}.*

From: Bolz et al. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.

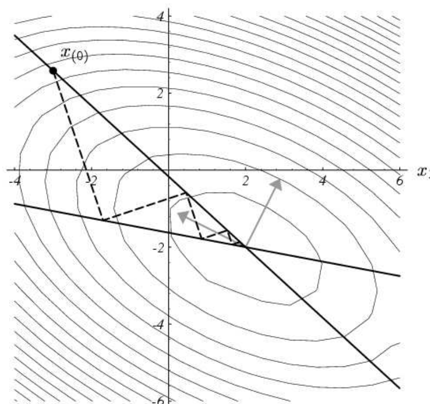
Solving linear systems

- Large, sparse linear systems of equations **Ax=b** arise in many practical applications
  - FEM of linear elasticity
  - Fluid dynamics
  - ...
- And many approaches can be used to find **x**
  - Invert matrix A
  - Gaussian elimination
  - **Conjugate gradient** (Square, symmetric and positive definite A)
  - **Jacobi iteration**
  - **Multigrid**
  - ...

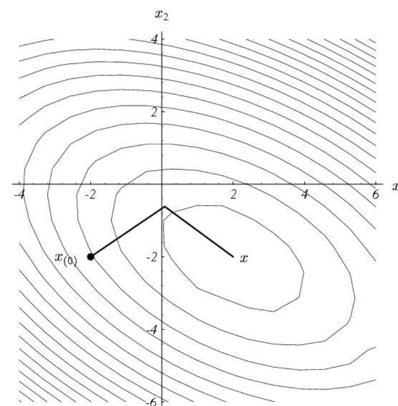
Conjugate gradient

- A quadratic form is a scalar, quadratic function of a vector:
  - $f(x)=½x^TAx-b^Tx+c$
- If A is symmetric and positive definit f(x) is minimized by the solution to Ax=b.
  - Since $f'(x)=Ax-b$ then $f'(x)=0 \Rightarrow Ax=b$

Conjugate gradient



Steepest descent



Conjugate gradient

Image source: Shewchuk JR:
An introduction to the conjugate gradient method without the agonizing pain

Conjugate gradient

Unpreconditioned CG

1. $p^{(0)} = r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
2. **for** $i \leftarrow 0$ **to** $\#itr$
3. $\rho_i = r^{(i)^T} r^{(i)}$
4. $q^{(i)} = Ap^{(i)}$
5. $\alpha_i = \rho_i / p^{(i)^T} q^{(i)}$
6. $x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
7. $r^{(i+1)} = r^{(i)} - \alpha_i q^{(i)}$
8. $\beta_i = r^{(i+1)^T} r^{(i+1)} / \rho_i$
9. $p^{(i+1)} = r^{(i+1)} + \beta_i p^{(i)}$
10. convergence check

Unpreconditioned GPU-based CG

1. **clMatVec**($CL_SUB, A, x^{(0)}, b, r^{(0)}$) initial guess $x^{(0)}$
2. **clVecOp**($CL_ADD, -1, 0, r^{(0)}, NULL, r^{(0)}$)
3. **clVecOp**($CL_ADD, 1, 0, r^{(0)}, NULL, p^{(0)}$)
4. **for** $i \leftarrow 0$ **to** $\#itr$
5. $\rho_i =$ **clVecReduce**($CL_ADD, r^{(i)}, r^{(i)}$)
6. **clMatVec**($CL_ADD, A, p^{(i)}, NULL, q^{(i)}$)
7. $\alpha_i =$ **clVecReduce**($CL_ADD, p^{(i)}, q^{(i)}$)
8. $\alpha_i = \rho_i / \alpha_i$
9. **clVecOp**($CL_ADD, 1, \alpha_i, x^{(i)}, p^{(i)}, x^{(i+1)}$)
10. **clVecOp**($CL_SUB, 1, \alpha_i, r^{(i)}, q^{(i)}, r^{(i+1)}$)
11. $\beta_i =$ **clVecReduce**($CL_ADD, r^{(i+1)}, r^{(i+1)}$)
12. $\beta_i = \beta_i / \rho_i$
13. **clVecOp**($CL_ADD, 1, \beta_i, r^{(i+1)}, p^{(i)}, p^{(i+1)}$)
14. convergence check

Image source: Krüger and Westermann:
Linear Algebra Operators for GPU implementation of Numerical Algorithms

Jacobi iteration

- Choose initial guess:
  - $X^{(0)}=(2,1,4)^T$
- Solve each row of the linear system assuming the other variables are correct.
  - $X^{(1)}=(0.57,2.5,2.8)^T$
- Converges to the solution $x=(1,2,3)^T$?

$$\begin{bmatrix} 7 & -1 & 2 \\ 3 & -8 & 1 \\ 1 & 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 11 \\ -10 \\ 16 \end{bmatrix}$$

$$\begin{aligned} 7x_1^{(1)} - 1(1) + 2(4) &= 11 \\ 3(2) - 8x_2^{(1)} + 1(4) &= -10 \\ 1(2) + 0(1) + 5x_3^{(1)} &= 16 \end{aligned}$$

Jacobi iteration

We say that A is *diagonally dominant* if

$$|a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}| \qquad \text{for} \qquad i = 1, \ldots, n$$

If A is diagonally dominant the Jacobi iteration will converge to the solution of Ax=b

Jacobi iteration

At the $(i+1)$st iteration, the vector $\mathbf{x}^{(i+1)}$ is calculated by

$$x_j^{(i+1)} = \frac{1}{a_{jj}} \left(- \sum_{\substack{k=1 \\ k \neq j}}^{n} a_{jk} x_k^{(i)} + b_j \right) \qquad j = 1, \ldots, n \tag{3}$$

We will have a concrete use of this in our fluid solver in the next lecture.

Multigrid

- Faster convergence on coarser scale?
  - Relaxation / solve
    - Jacobi iteration
  - Projection operator P
    - Full weighing (as in an averaging reduce operation)
  - Interpolation operator
    - Bi-/tri-linear interpolation

Multigrid

$\mathbf{u}_h \leftarrow$ V-Cycle$(\eta_1, \eta_2, \mathbf{v}_h, \mathbf{b}_h)$
  If (CoarsestQ (h))
    Return $\mathbf{u}_h \leftarrow$ Solve$(\mathbf{A}_h, \mathbf{v}_h, \mathbf{b}_h)$
  Else
    $\mathbf{v}_h \leftarrow$ Relax$(\eta_1, \mathbf{v}_h, \mathbf{b}_h)$ // pre-smooth
    $\mathbf{b}_{2h} \leftarrow \mathbf{P}(\mathbf{b}_h - \mathbf{A}_h\mathbf{v}_h)$ // project residual
    $\mathbf{v}_{2h} \leftarrow$ V-Cycle$(\eta_1, \eta_2, \mathbf{0}_{2h}, \mathbf{b}_{2h})$ // recurse
    $\mathbf{v}_h \leftarrow \mathbf{v}_h + \mathbf{S}\mathbf{v}_{2h}$ // interpolate & correct
    Return $\mathbf{u}_h \leftarrow$ Relax$(\eta_2, \mathbf{v}_h, \mathbf{b}_h)$ // post-smooth

$\mathbf{u}_h \leftarrow$ Relax$(\eta, \mathbf{v}_h, \mathbf{b}_h)$
  For ($i = 0; i < \eta; i = i + 1$)
    $\mathbf{r} \leftarrow \mathbf{b}_h - \mathbf{A}_h\mathbf{v}_h$
    $\mathbf{v}_h \leftarrow \mathbf{v}_h + \omega(\mathbf{A}_h)_{ii}^{-1}\mathbf{r}$
  Return $\mathbf{u}_h \leftarrow \mathbf{v}_h$

From: Bolz et al. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.