Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid

Jeffrey Bolz Ian Farmer

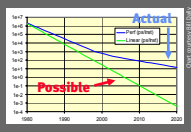
Eitan Grinspun Peter Schröder

Caltech

Why Use the GPU?

Semiconductor trends

- cost
- wires vs. compute
- Stanford streaming supercomputer



Parallelism

- many functional units
- graphics is prime example

New Hardware Features

Latest generation graphics HW

- floating point throughout
- programmability
- high resource limits
  - dependent texturing
  - many registers
  - many instructions

How to Exploit?

Harvesting this power

- what application suitable?
- what abstractions useful?

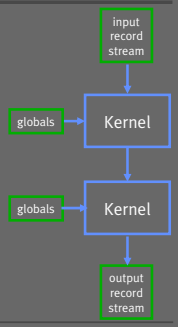
History

- massively parallel SIMD machines
- media processing

Streaming Model

What is the right abstraction?

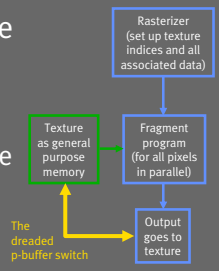
- Purcell, et al. 2002
- data structures ⇢ streams ⇢ textures
- algorithms ⇢ kernels ⇢ fragment programs



Mapping the Program

How does a program execute?

- "render" a rectangle
- memory as texture
- fragment program
  - many pixels provide parallelism
- write to texture
  - close the loop



Our Program

Kernels for scientific computing

- sparse matrix solvers
  - not high arithmetic intensity…
  - but… we need them everywhere
- unstructured: conjugate gradients
- structured: multigrid

Lessons to learn here…

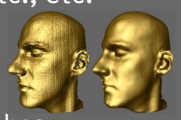
Sparse Matrices

Ubiquitous in numerical computing

- discretization of PDEs: animation
  - finite elements, difference, volumes
- optimization, editing, etc., etc.

Example here:

- processing of surfaces
  - 2-manifold triangle meshes



Geometric Flow

Canonical non-linear problem

- mean curvature flow

$$\partial_t x_i(t) = -\lambda_i(t) H_i(t) \vec{n}_i(t)$$

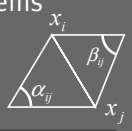
Velocity opposite mean curvature normal

- implicit time discretization

$$Ax^{(i+1)} = x^{(i)}$$

- solve sequence of SPD systems

$$a_{ij} = -\lambda \Delta t(\cot(\alpha_{ij}) + \cot(\beta_{ij}))$$
$$a_{ii} = 4A_i - \sum_{j \in N(i)} a_{ij}$$



Non-Linear Problems

Basic structure

- solver for SPD systems
  - conjugate gradients
  - other variants if not SPD
- recompute matrix entries on GPU
  - minimize CPU to GPU traffic
- control structure on CPU

Conjugate Gradients

High level code

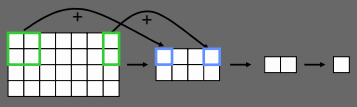
- inner loop
- matrix-vector multiply
- sum-reduction
- scalar-vector MAD

$$\begin{aligned}
&while(\delta_{new} < \varepsilon^2 \delta_0) \\
&\quad q = Ad \\
&\quad \alpha = \delta_{new} / d^T q \\
&\quad x = x + \alpha d \\
&\quad r = r - \alpha q \\
&\quad \delta_{old} = \delta_{new} \\
&\quad \delta_{new} = r^T r \\
&\quad \beta = \delta_{new} / \delta_{old} \\
&\quad d = r + \beta d
\end{aligned}$$

Vector Inner Products

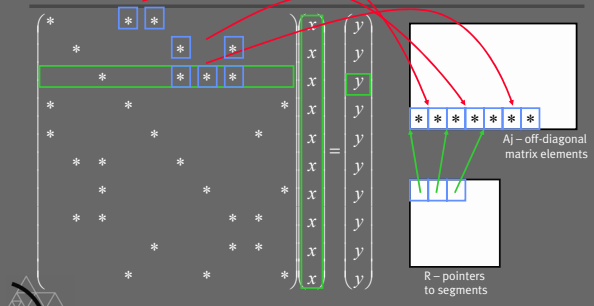
Decompose

- fragment-wise multiply
- followed by sum-reduction

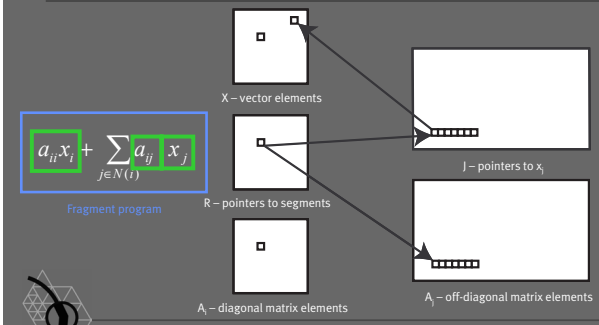


- odd dimensions can be handled
- useful for other operations as well

Ax=y



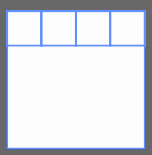
Matrix Vector Product



Apply to All Pixels

Two extremes

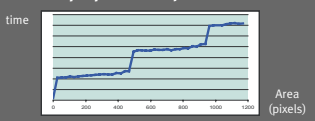
- one row at a time
  - setup overhead
- all rows at once
  - limited by worst row
- middle ground
  - organize “batches” of work
  - what size? how to organize?



What Size Batches?

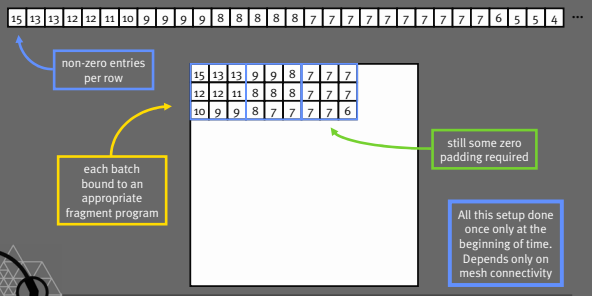
We choose fixed size rectangles

- fragment pipe is quantized



- simple experiments reveal best size
  - performance model
    - area, diagonal, wasted frags

Packing (Greedy)

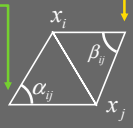


Recomputing Matrix

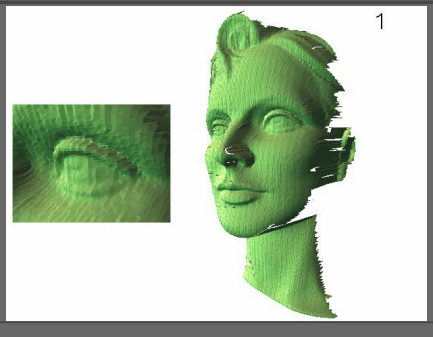
Matrix entries depend on surface

- must "render" into matrix
- two additional indirection textures
  - previous and next

$$a_{ij} = -\lambda \Delta t(\cot(\alpha_{ij}) + \cot(\beta_{ij}))$$
$$a_{ii} = 4A_i - \sum_{j \in N(i)} a_{ij}$$

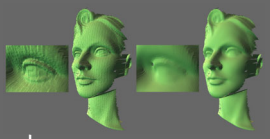


Surface Smoothing



Results (NV30@500MHz)

37k elements



- matrix multiply
  - 33 instructions
  - 1/100th of a second
- reduction
  - 7 instructions/fragment/pass
  - 1/1900th of a second
- CG solve in 1/20th of a second

Results

How efficient?

- lots of indirection
  - 33 instructions for 13 (average) flops
- bandwidth limited
  - fat texture fetches are slow
- not the ideal example for GPU…
  - small op/fetch ratio

So Far…

Unstructured matrices

- irregular triangle meshes
  - also tet meshes
- main issue is layout of matrix

Structured matrices

- much nicer layout
- example: fluids, image processing

Regular Grids

PDEs again

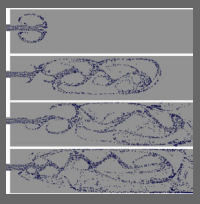
- this time variables on “pixel grid”
  - e.g.: Navier-Stokes

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

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

after discretization: solve Poisson eq. at each time step

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

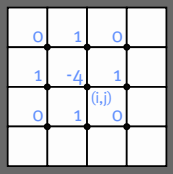


Poisson Equation

Appears all over the place

- easy to discretize on regular grid
- matrix multiply is stencil application
- FD Laplace stencil:

$$\nabla^2 X_{i,j} = X_{i-1,j} + X_{i+1,j} + X_{i,j-1} + X_{i,j+1} - 4X_{i,j}$$



Solver

Use iterative matrix solver

- just need application of stencil
  - easy: just like filtering
  - incorporate geometry (Jacobian)
  - variable coefficients

But..., very ill-conditioned

- use multigrid to counteract

Multigrid (basic)

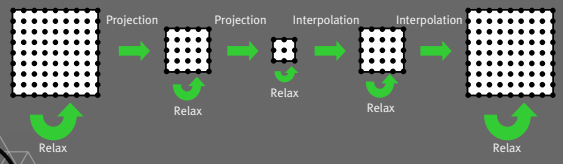
Principles

- lower frequency error resolved on coarser grid
- implementation needs:
  - interpolation (coarse⇢fine)
  - projection (fine⇢coarse)
  - relaxation
    - Jacobi iterations

V-Cycle

Fine to coarse to fine cycle

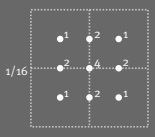
- project residual, relax, interpolate correction vector



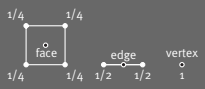
Computations

Lots of stencil applications

- matrix multiply: 3x3 stencil
- projection: 3x3 stencil
- interpolation: 2x2(!)
  - floor op in indexing...



$$\mathbf{v}_h[i] = 1/4 \sum_{d\in\{0,1\}^2} \mathbf{v}_{2h}\lfloor (i+d)/2 \rfloor$$



Details

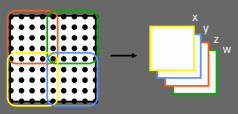
Boundaries

- Dirichlet boundaries
  - boundary variables are fixed
- Neumann boundaries
  - out of bounds access clamped to 0
- gradient and divergence operators
  - one sided differences at boundary

Texture Layout

Storage for matrices and DOFs

- variables in one texture
- matrices in 9(=3x3) textures
- all textures packed
  - exploit 4 channels
  - domain decomp.
  - padded boundary



Coarser Matrices

Operator at coarser level

- needed for relaxation at all levels
- triple matrix product

$A^c = P := {}^1/_4 S^T \; A^f \; S$

Effectively:
Stencil composition

Stencil Composition

Triple matrix product...

- work out terms and map to stencils
  - exploit local support of stencils
  - straightforward but t-e-d-i-o-u-s

$$A_{2h}^d[i] = 1/4 \sum_{e,g\in\{-1,0,1\}^2} S^e S^{e+g-2d} A_h^g[2i+e]$$
$$= 1/4 \sum_{e,g\in\{-1,0,1\}^2} S'[e] S'[e+g-2d] A_h^g[2i+e]$$

  - store S in texture S' with a 0 boundary

More Details

What is variable?

- only matrix entries (stencils) vary
- all other operators hard wired
  - still general solution!
- debugging
  - oh, joy...
- obstacles resolved at coarsest level

Flow Example

Putting it all together

- here: fixed velocity in and out
- tracer particles for visualization
  - simple advection in evolving flow

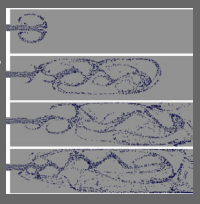
Results (NV30@500MHz)

257x257 grid

- matrix multiply - 27 instructions
  - 1/800th of a second
- interpolation 10 instr.
- projection 19 instr.

Overall performance

- 257x257 at 90 fps!



Problems

Bleeding edge...

- PBuffer overhead is killing us
- managing layout in a buffer by hand... OUCH
- scalar versus vector problems
- give us rectangular texture border

Enhancements

Small/large changes?

- global registers for reductions
- texture fetch with offset
- scatter (displacement mapping?!)
  - rasterization order undefined
    - scatter w/ undefined order still useful
- scientific computing compiler

Conclusion

Where are we now?

- performance not up to expectations
- still a good streaming processor
- most kernels run at 1-2GFlops/s
  - SSE on P4 still very competitive
- should beat CPUs in about a year
- better languages! Brook? C*?