Classical Algebraic Multigrid for CFD using CUDA

Simon Layton, Lorena Barba

With thanks to Justin Luitjens, Jonathan Cohen (NVIDIA)

BU College of Engineering

Problem Statement

- Solve elliptical problems with the form

$$Au = b$$

- For Instance the Pressure Poisson equation from Fluids

$$\nabla^2 \phi = \nabla \cdot u^*$$

  - This commonly takes 90%+ of total run time

- These systems arise in many other engineering applications

What is Multigrid?

- *Hierarchical*
  - Repeatedly reduce the size of the problem
  - Solve these smaller problems & transfer corrections
- *Iterative*
  - Based on a smoother (i.e. Jacobi, Gauss-Seidel)
  - Solve using recursive ‘cycles’

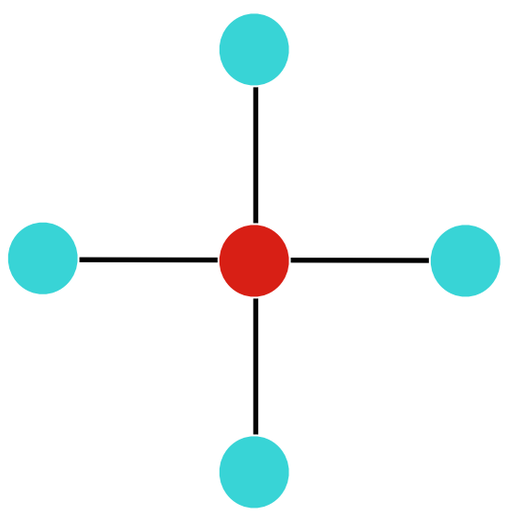
Why do we care?

- *Optimal complexity*
  - Convergence independent of size of problem
  - Consequently *O(N)* in time

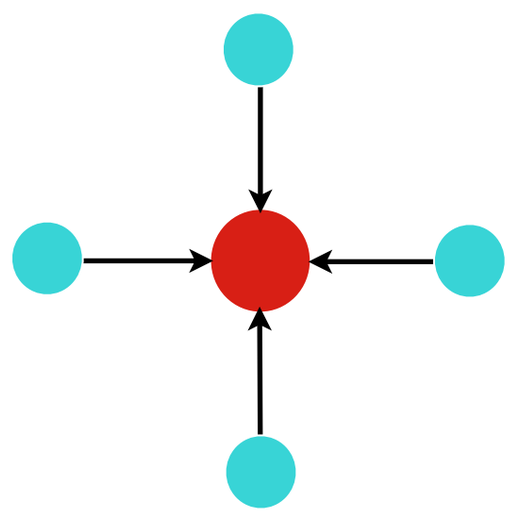
- *Parallel & scalable*
  - Hypre package from LLNL scales to 100k+ cores
  - "Scaling Hypre's multigrid solvers to 100,000 cores", AH Baker et. al.

Classical vs. Aggregative AMG

- Solve phase is the same - only difference in setup
  - Classical *selects* variables to be in the coarser level
  - Aggregation *combines* variables to form the coarser level



Classical:
Choose variable to keep on coarse level



Aggregation:
Combine variables to form entry on coarse level

AMG - Main Components

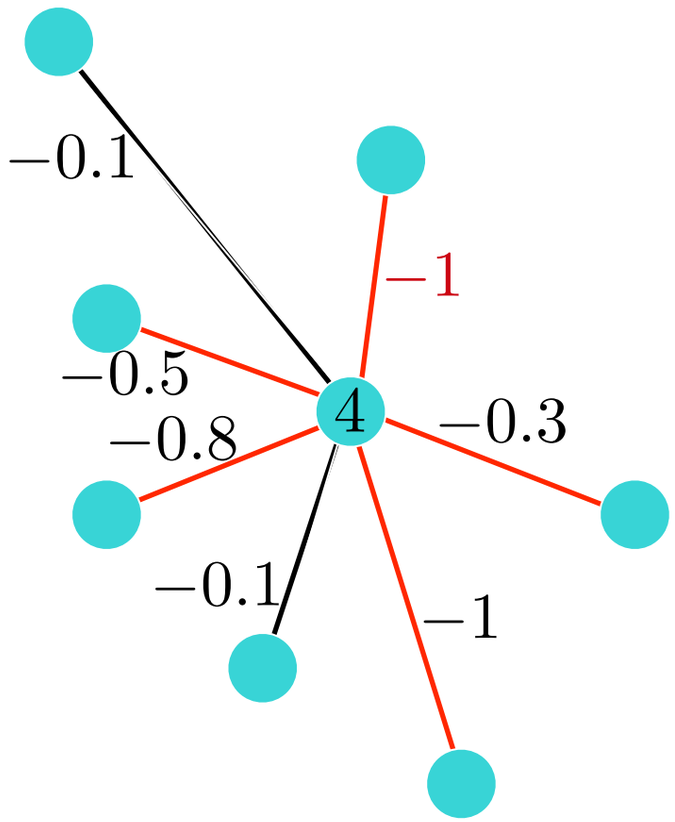
- Consists of 5 main parts

1. *Strength of Connection metric* (SoC)
   - Measure of how strongly matrix variables affect each other
2. *Selector / Aggregator*
   - Choose variables / aggregates to move to coarser level
3. *Interpolator / Restrictor*
   - To generate coarsened matrices & transfer residuals between levels
4. *Galerkin Product*
   - Generate next coarsest level: $A^{k+1} = R^k A^k P^k$
5. *Solver cycle*
   - Consists of Sparse Matrix-Vector multiplications

Strength of Connection

- Variable i *strongly depends* on variable j if:

$$-a_{ij} \geq \alpha \max_{k \neq i}(-a_{ik})$$

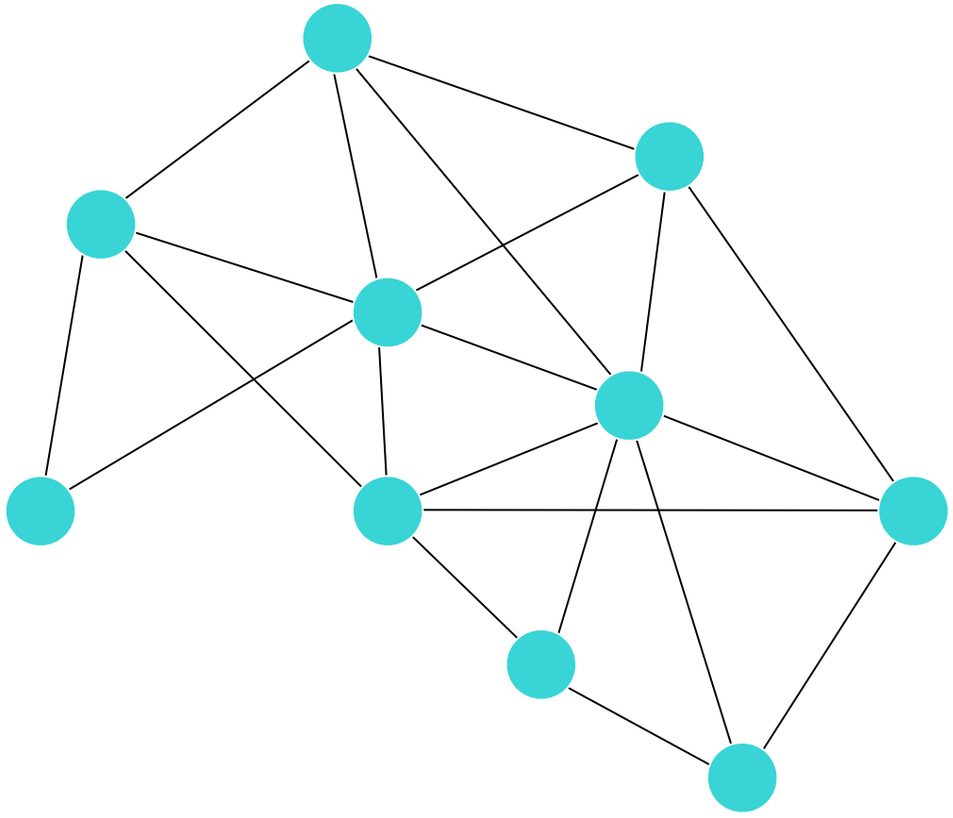


In this case:
α= 0.25
Threshold = -0.25

- After max value calculated, strength calculation for each edge can be performed in parallel

PMIS Selector

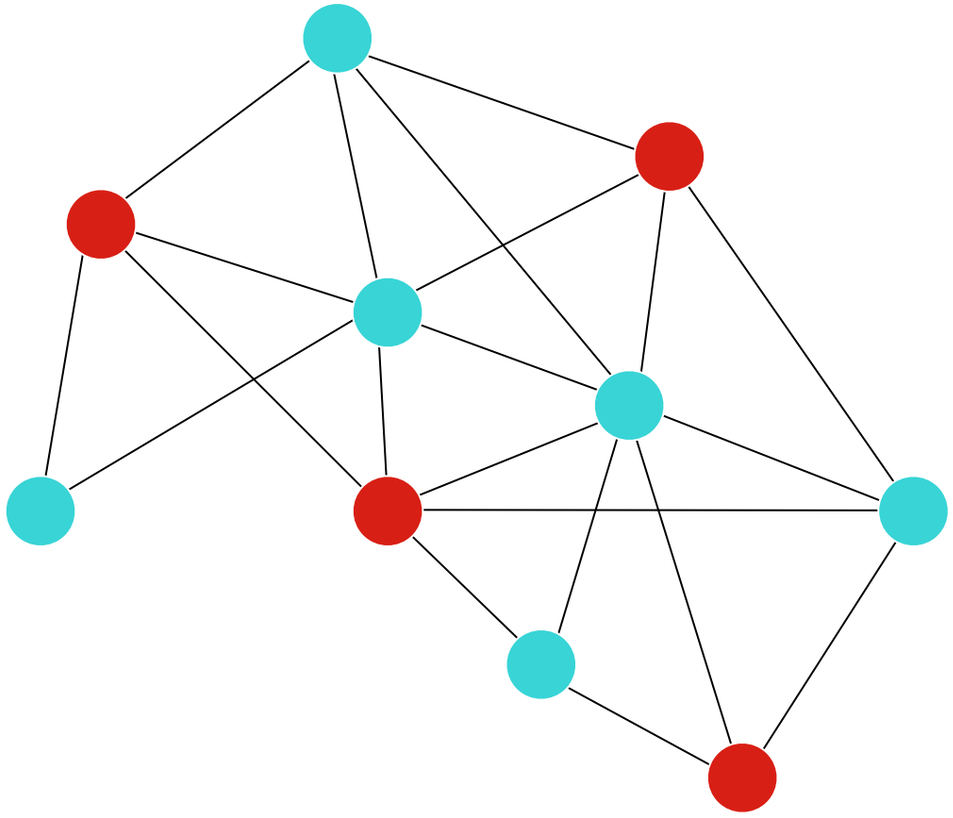
- Select some variables to be coarse
  - Use strongest connection
  - Neighbours of these variables are now fine
- Repeat until all variables are classified



Initial graph

PMIS Selector

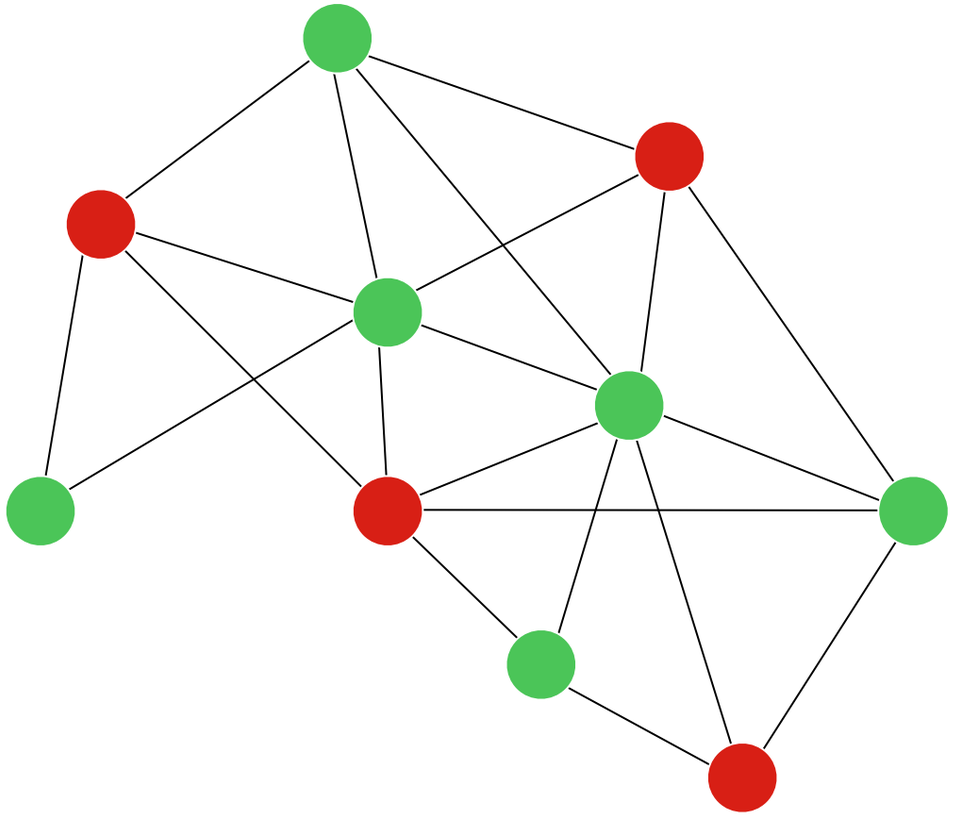
- Select some variables to be coarse
  - Use strongest connection
  - Neighbours of these variables are now fine
- Repeat until all variables are classified



Select some variables to be coarse

PMIS Selector

- Select some variables to be coarse
  - Use strongest connection
  - Neighbours of these variables are now fine
- Repeat until all variables are classified



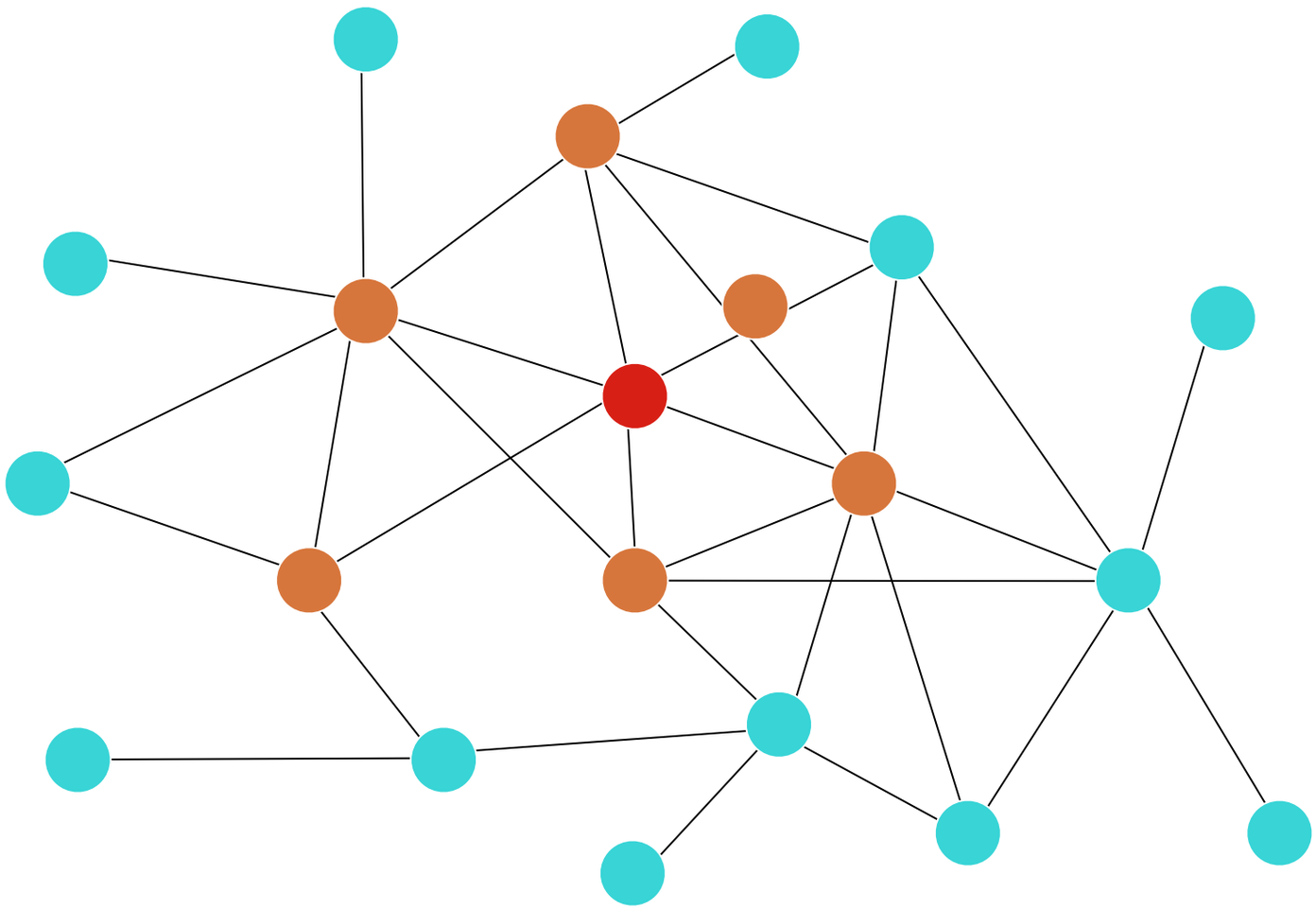
Neighbours of these coarse variables marked as fine

Interpolator

- Extended+i interpolator used

  - Needed to ensure good convergence with our selector

  - Considers neighbours of neighbours for interpolation set



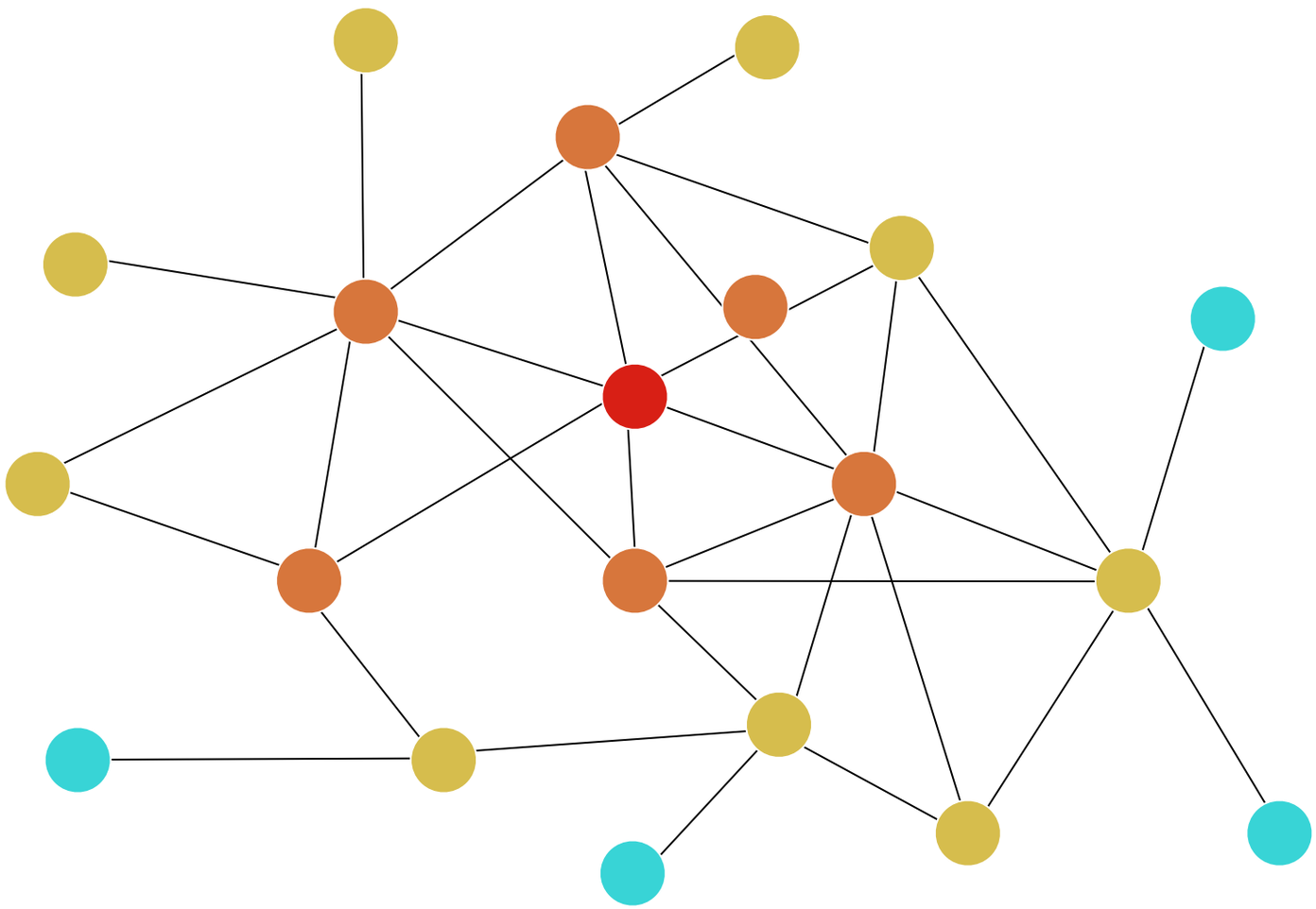
Distance 1

Interpolator

- Extended+i interpolator used

  - Needed to ensure good convergence with our selector

  - Considers neighbours of neighbours for interpolation set



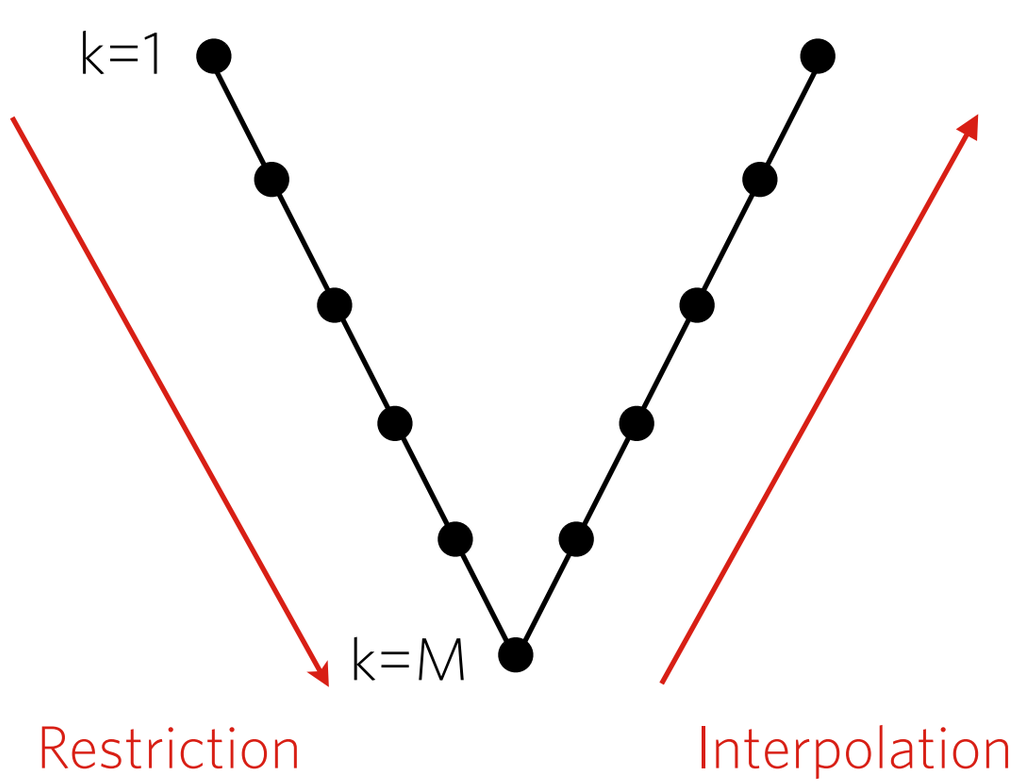
Distance 2

Solve Cycle

- Described recursively

  - Simplest is V-cycle

```
if (level = M (coarsest))
        solve A^M u^M = f^M
else
        Apply smoother n1 times to  A^k u^k = f^k
        Perform coarse grid correction
            Set r^k = f^k - A^k u^k
            Set r^(k+1) = R^k r^k
            Call cycle recursively for next level
            Interpolate e^k = P^k e^(k+1)
            Correct solution: u^k = u^k + e^k
        Apply smoother n2 times to  A^k u^k = f^k
```



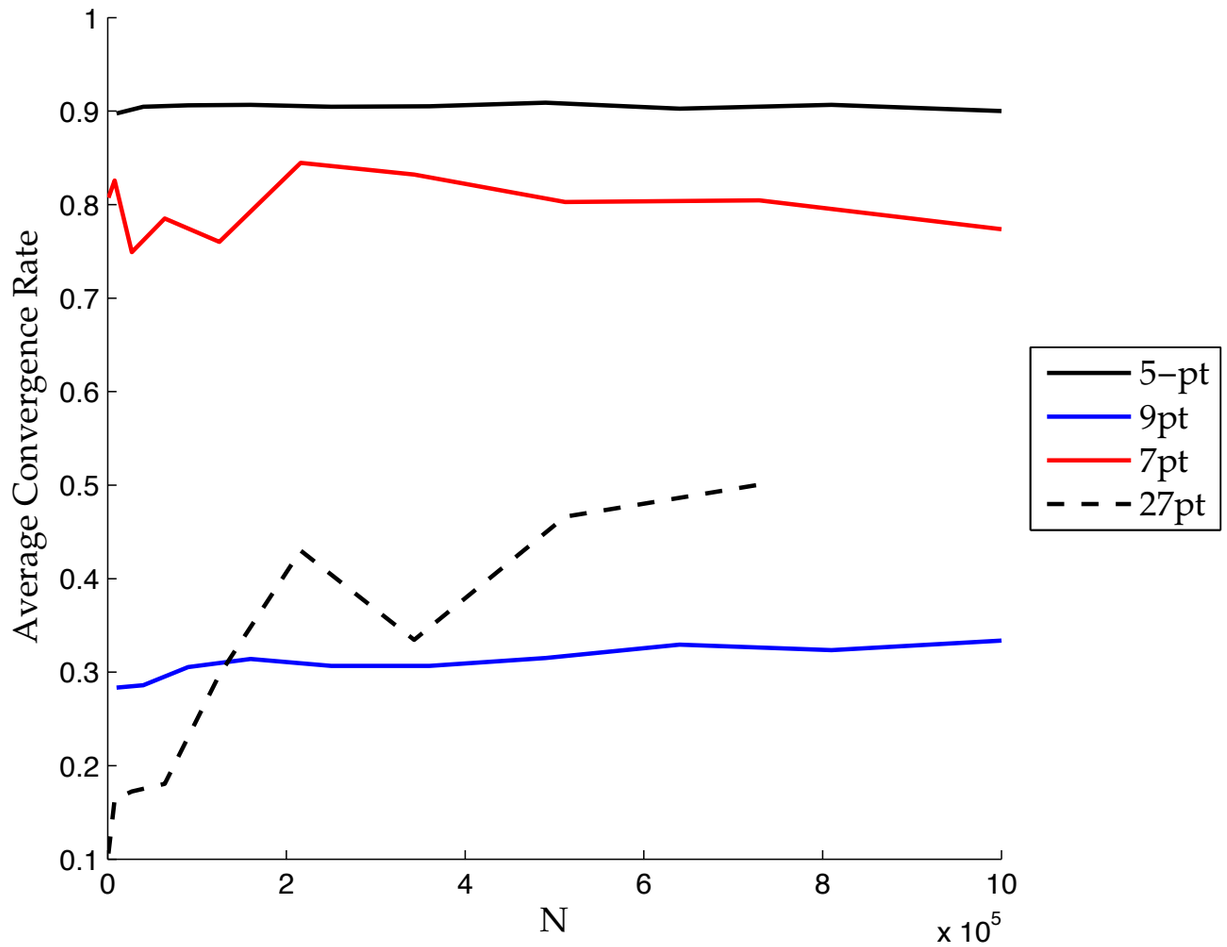
GPU Implementation - Initial Thoughts

- Most operations seem to easily expose parallelism:
  - Strength of connection
  - PMIS selector
  - Solve cycle - SpMV from cusp / cusparse
- Others less obvious or trickier:
  - Galerkin Product
  - Interpolator
- Want to ensure correctness
  - Compare against identical algorithms in Hypre

Initial Results - Convergence

- Regular Poisson grids

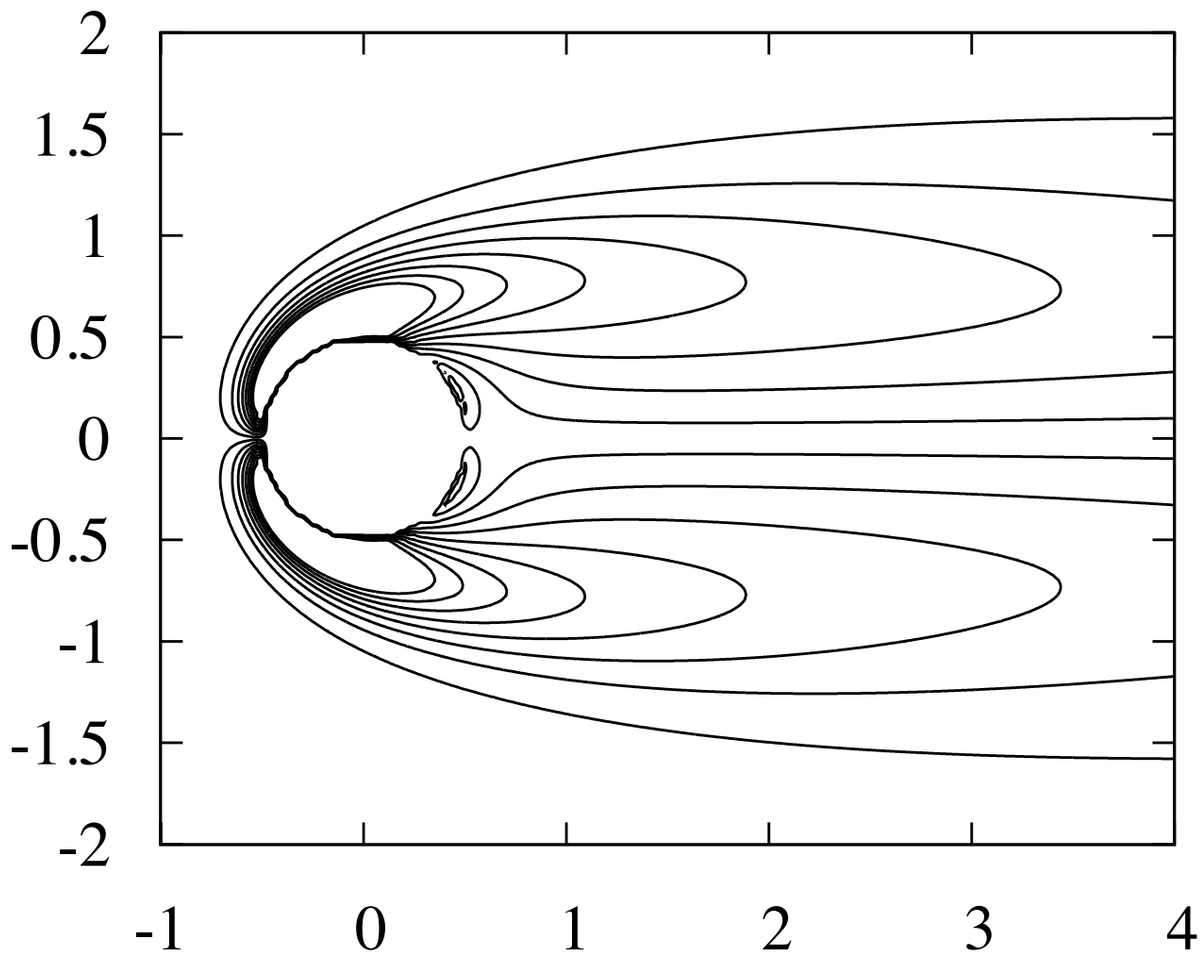
  - Demonstrates problem size invariance for convergence rate



27pt stencil runs out of memory on C2050 - we expect to get the same behaviour as other stencils

Initial Results - Immersed Boundary Method

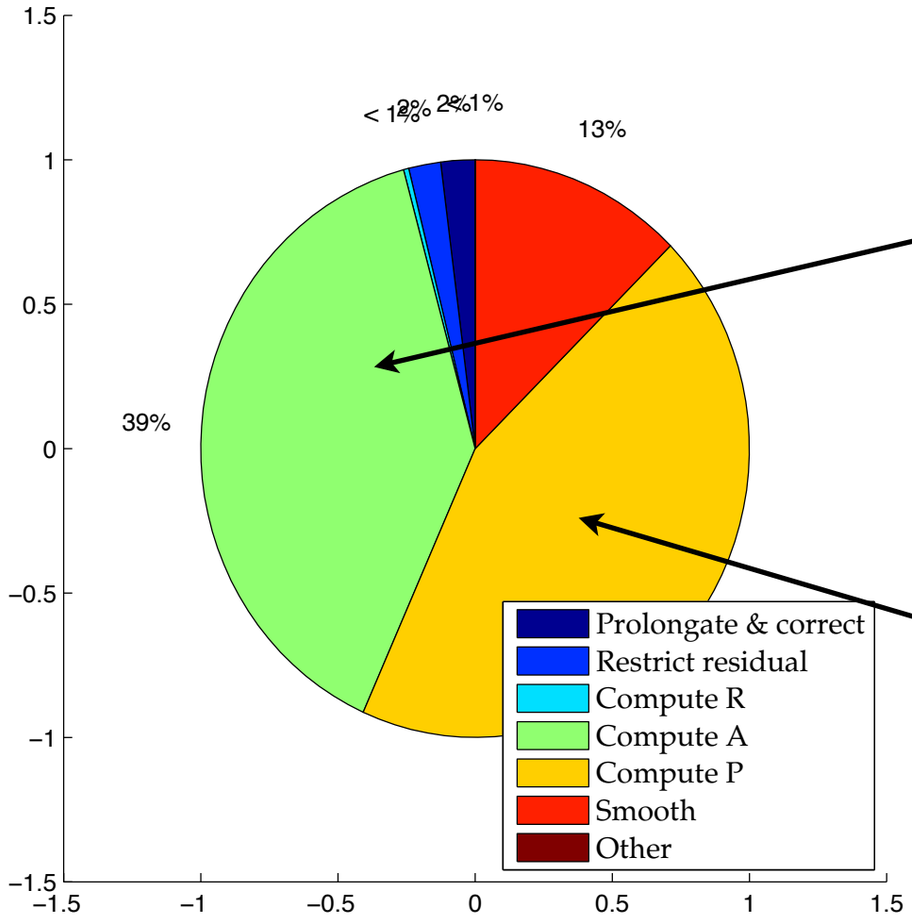
- Speed compared to 1 node using Hypre (unoptimized)



Code	*Time*	*Speedup*
CUDA	3.57s	1.87x
Hypre (1C)	6.69s	-
Hypre (2C)	5.29s	1.26x
Hypre (4C)	4.16s	1.61x
Hypre (6C)	4.00s	1.67x

Where to go from here?

- Breakdown 5pt Poisson stencil by routine (easy case)



Compute A is Galerkin product - Sparse R*A*P

Interpolation matrix: Can we form this as an existing pattern?

Compute P - Interpolation Matrix

- Interpolation weights depend on the distance 2 set:

$$\hat{C}_i = C_i^s \cup \bigcup_{j \in F_i^s} C_j^s$$

- This repeated set union turns out to be fairly tricky..
  - Can be formed as a specialized SpMM

SpMM / Repeated set union

$$\hat{C}_i = C_i^s \cup \bigcup_{j \in F_i^s} C_j^s$$

- SpMM conceptually:
  - For each output row, if influencing column already in row, accumulate to current value, else add column to row
- Set union conceptually:
  - For each set of coarse connections, if entry already added to result set, ignore, else add to set.

$$\hat{C} = (I + F^s) \cdot C^s$$

SpMM: Idea

- Similar in more detail later by Julien Demouth (S0285)

```
For each row i in matrix A:          ← Warp
  Accumulator = []
  For each entry j in row i:
    rowB = column(j)
    For each entry k on rowB of B:   ← Thread
      val = value(j)*value(k)
      Accumulator.add(column(k),val)
  Accumulator.write
```

- Implement Accumulator as hash table in shared memory

  - Work on as many rows per block as possible that fit in sMem

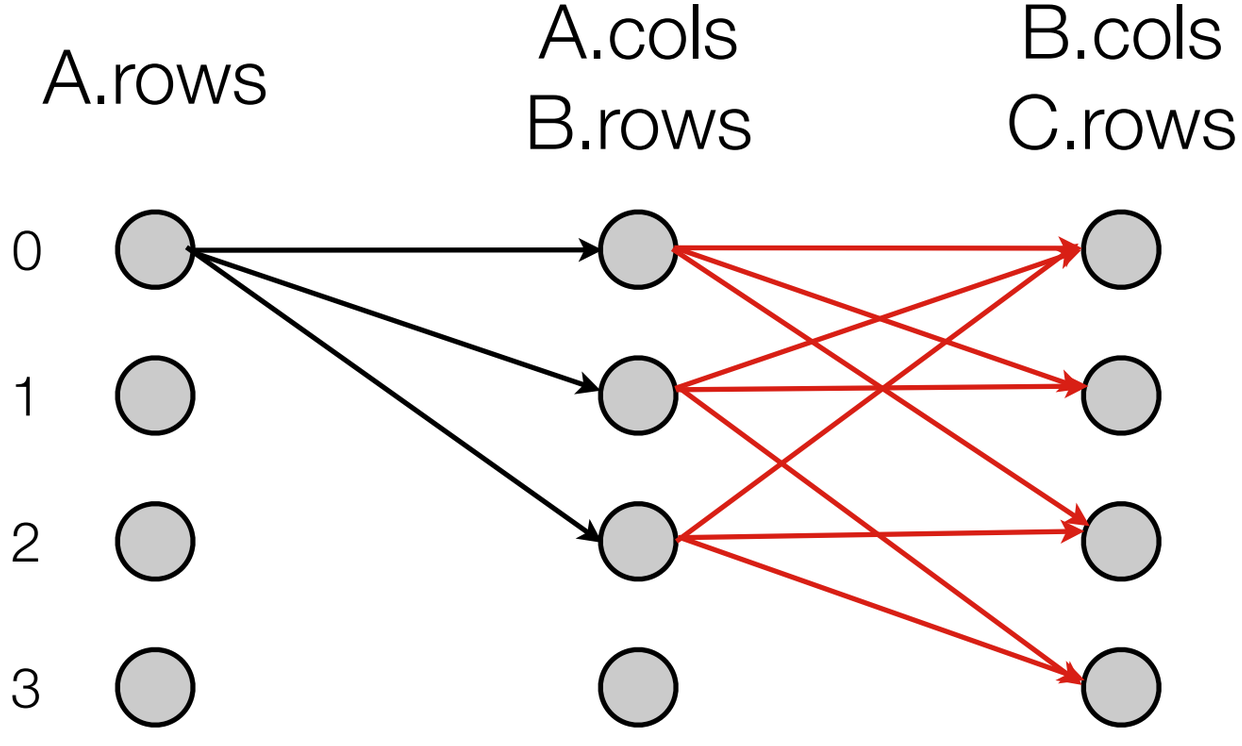
Problem!

- Performance *highly* dependent on size of hash tables
  - Not enough sMem = cannot solve problem
  - Too many values in sMem = many collisions = poor perf.
- Answer? Estimate storage needed
  - Allocate #warps per block & size of shared memory based on this estimation
  - If all else fails, pass multiplication to cusp
    - Worst case almost identical performance

Storage Estimation

- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998

- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)

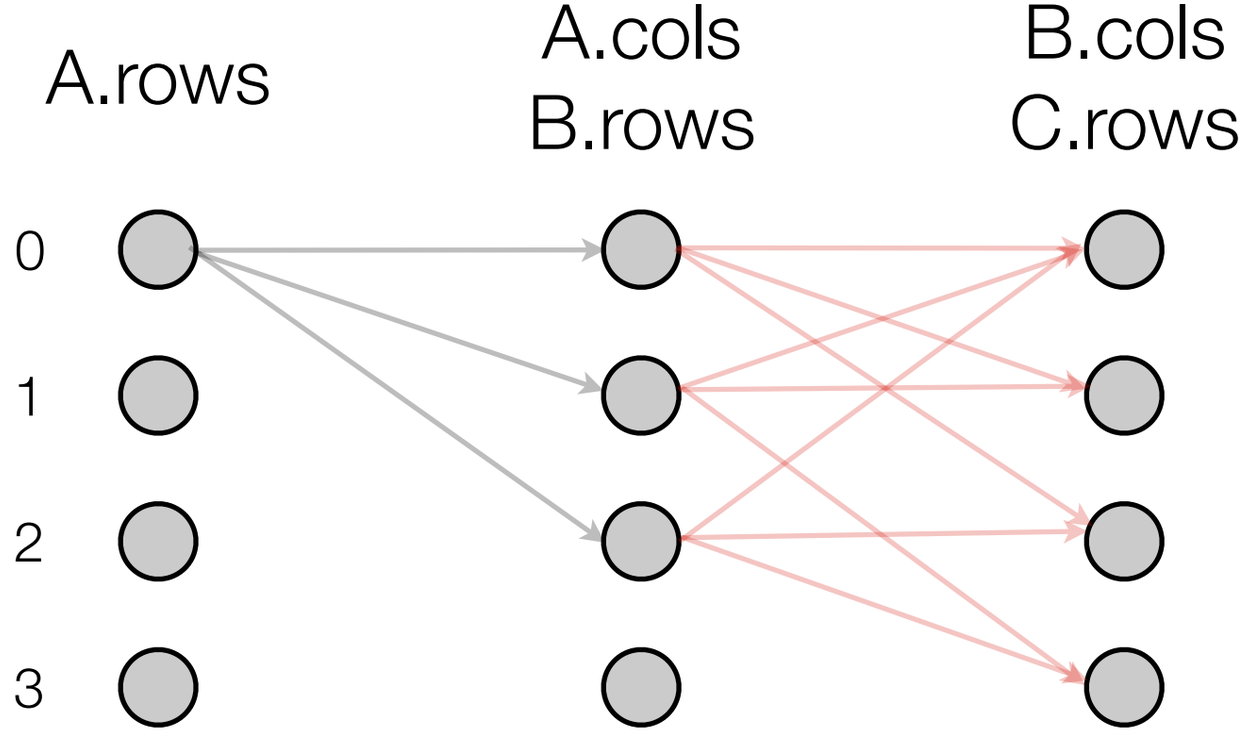


- By looking at connections, get estimation of non-zeros in C

Storage Estimation

- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998

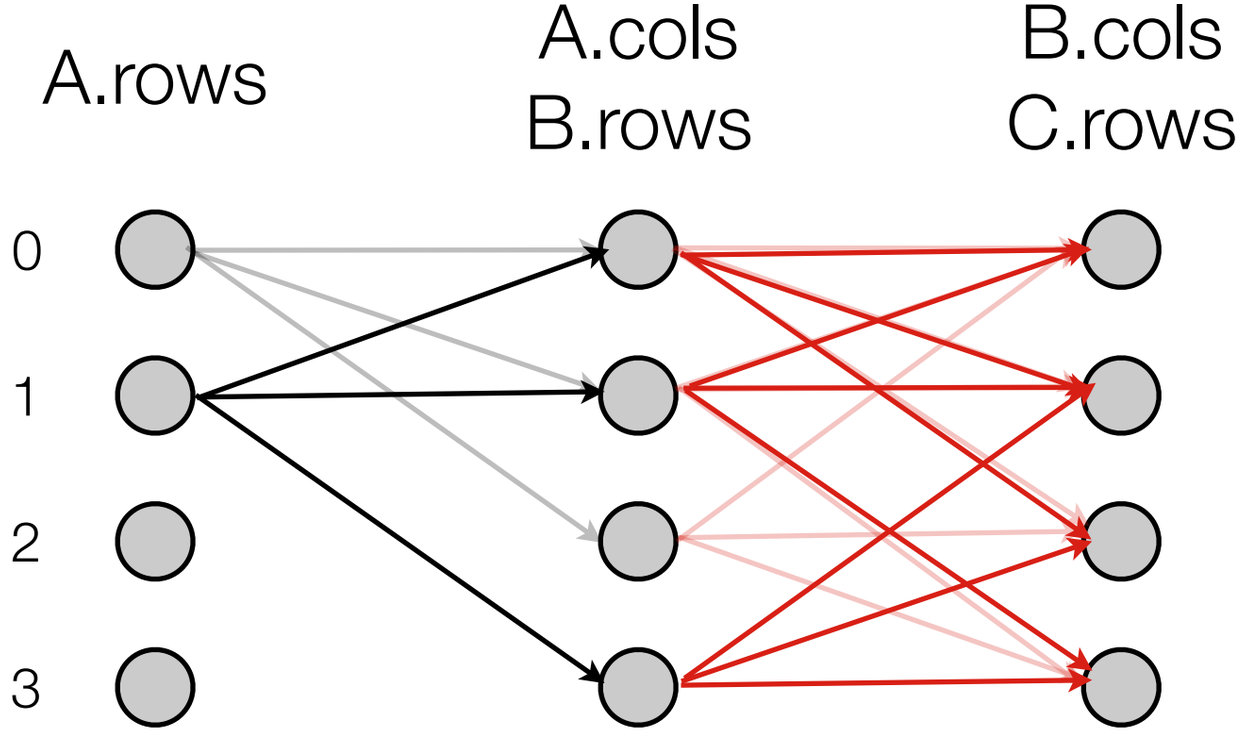
- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)



- By looking at connections, get estimation of non-zeros in C

Storage Estimation

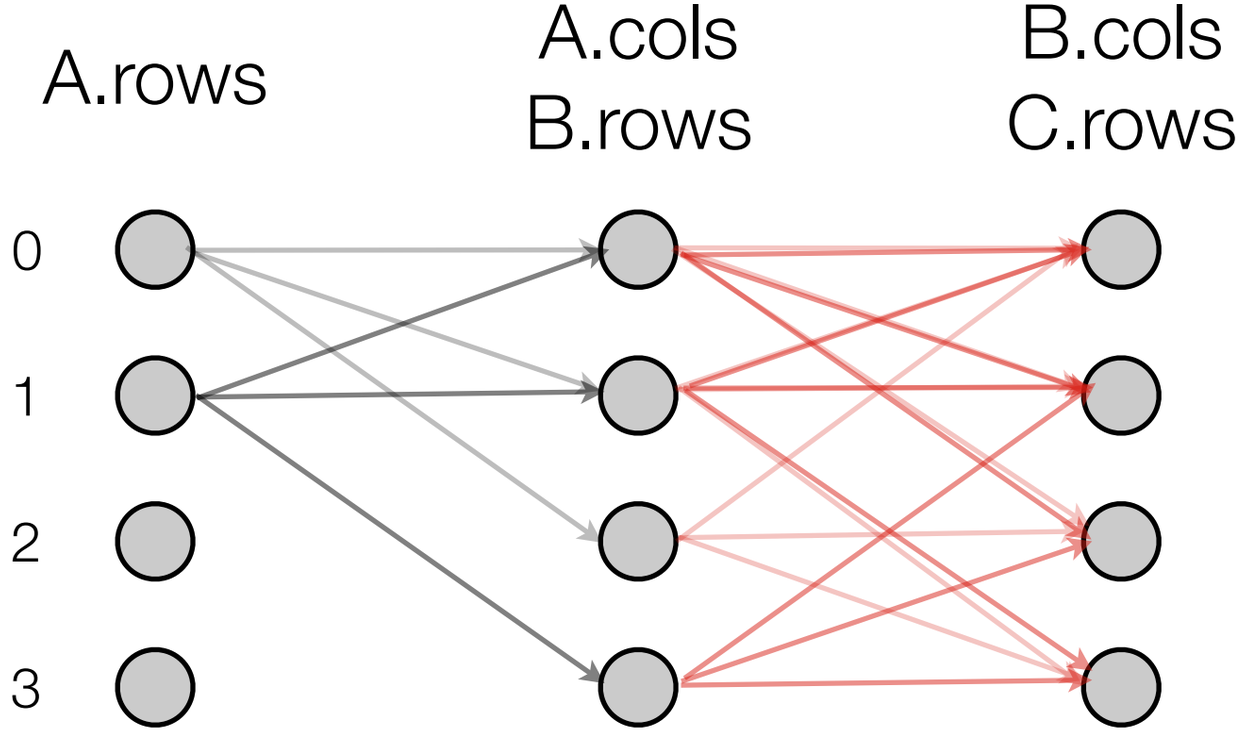
- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998
- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)



- By looking at connections, get estimation of non-zeros in C

Storage Estimation

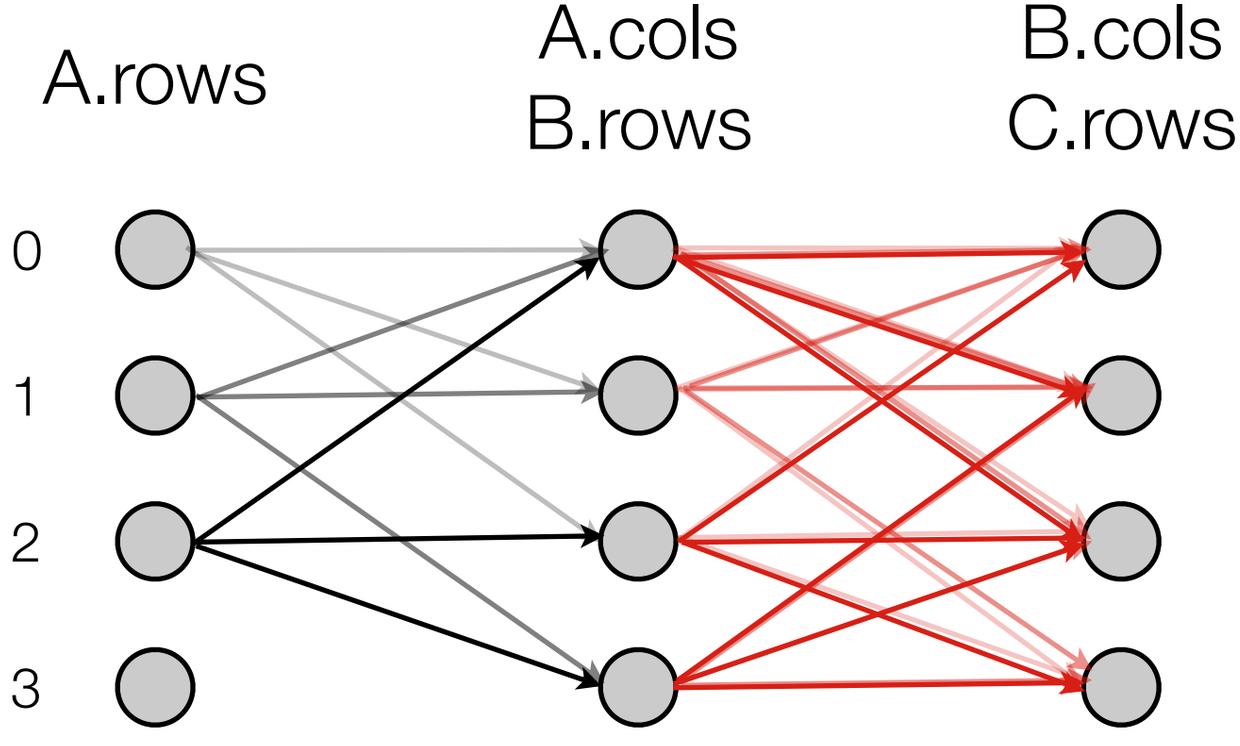
- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998
- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)



- By looking at connections, get estimation of non-zeros in C

Storage Estimation

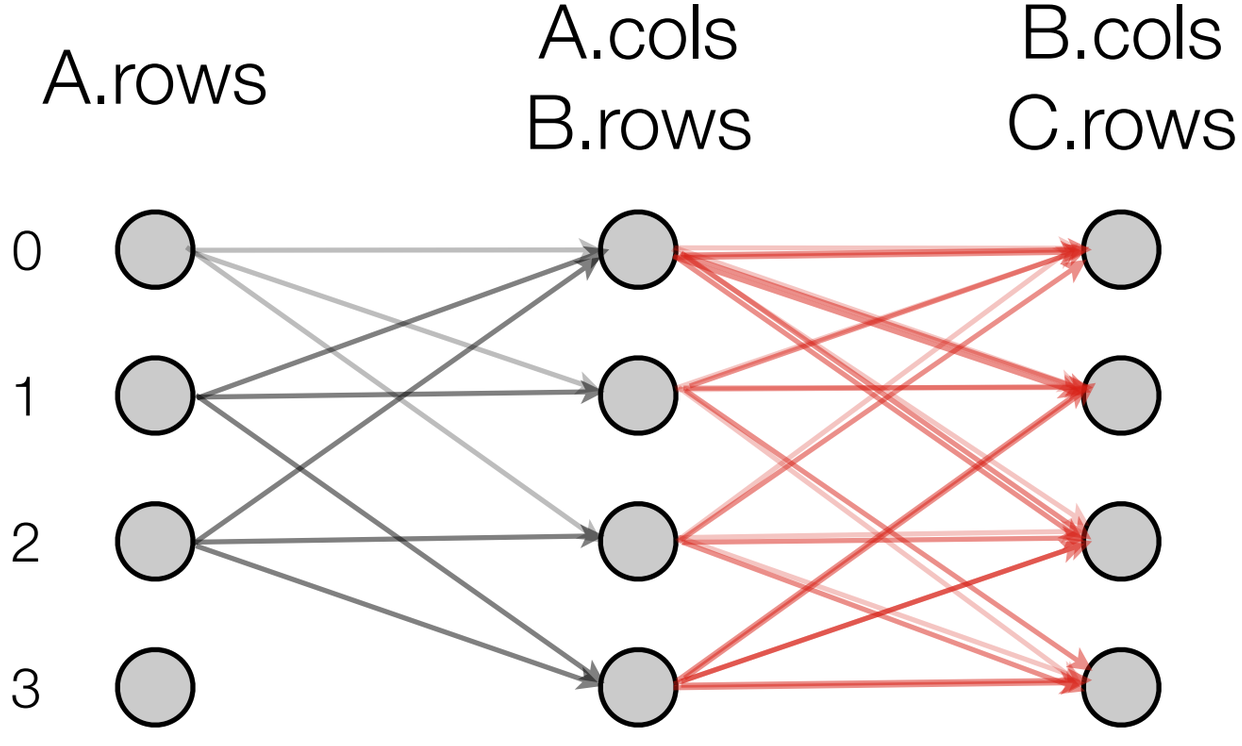
- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998
- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)



- By looking at connections, get estimation of non-zeros in C

Storage Estimation

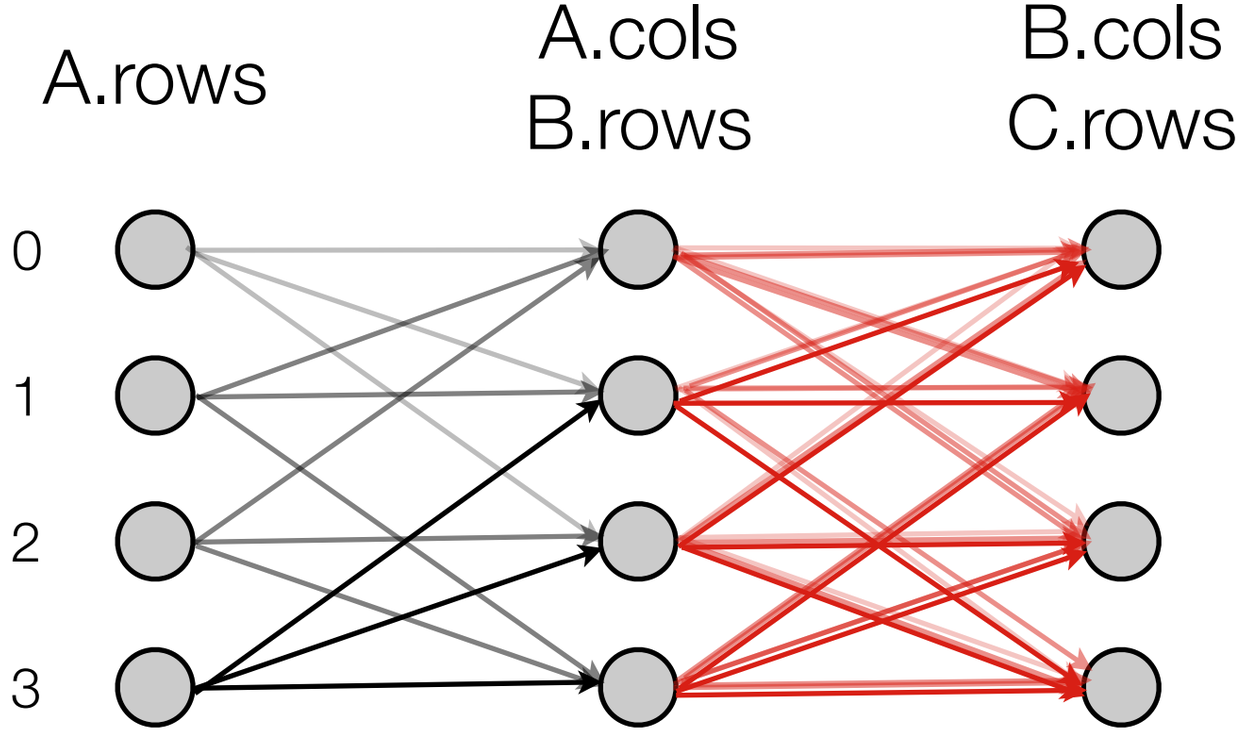
- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998
- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)



- By looking at connections, get estimation of non-zeros in C

Storage Estimation

- From Edith Cohen: '*Structure prediction and computation of sparse matrix products*' 1998
- Represent as bipartite graph (2x2 grid, 5pt Poisson stencil)



- By looking at connections, get estimation of non-zeros in C

Storage Prediction - Implementation

```
For r in [0,R)
  For each row, i:
    u[i] = rand() // exponential dist.

  For each row in A, i:
    v[i] = +infinity
    For each non-zero on i, j:
      v[i] = min(v[i],u[col[j]])
  For each row in B, i
    w[i][r] = +infinity
    For each non-zero on i, j:
      w[i][r] = min(w[i],v[col[j]])

nnz[i] = ceil((R-1)/sum(w[i][:])) // est
```

Storage Prediction - Optimizations

```
For r in [0,R)
  For each row, i:
    u[i] = rand() // exponential dist.

  For each row in A, i:
    v[i] = FLT_MAX
    For each non-zero on i, j:
      v[i] = min(v[i],hash(col[j]))
  For each row in B, i
    w[i][r] = FLT_MAX
    For each non-zero on i, j:
      w[i][r] = min(w[i],v[col[j]])

nnz[i] = ceil((R-1)/sum(w[i][:])) // estimator
```

Use pseudo-random hash

Only float precision needed

Generalized SpMVs

Conclusions

- Implemented full classical AMG algorithm in parallel
  - Validated against OSS code, Hypre
  - Tested with CFD problem

- Attempted to solve bottleneck of Galerkin product & repeated set intersection
  - Predict storage per row
  - Use this to determine launch parameters for hash-based SpMM using shared memory