Classical Algebraic Multigrid for Engineering Applications

Simon Layton, Lorena Barba (BU)

With thanks to Justin Luitjens, Jonathan Cohen (NVIDIA)

BU College of Engineering

nsf

Problem Statement

- Solve sparse linear systems from engineering problems

$$Au = b$$

- Pressure Poisson equation in fluids

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

  - 90%+ of total run time!

What is Multigrid and why do we care?

- *Hierarchical*

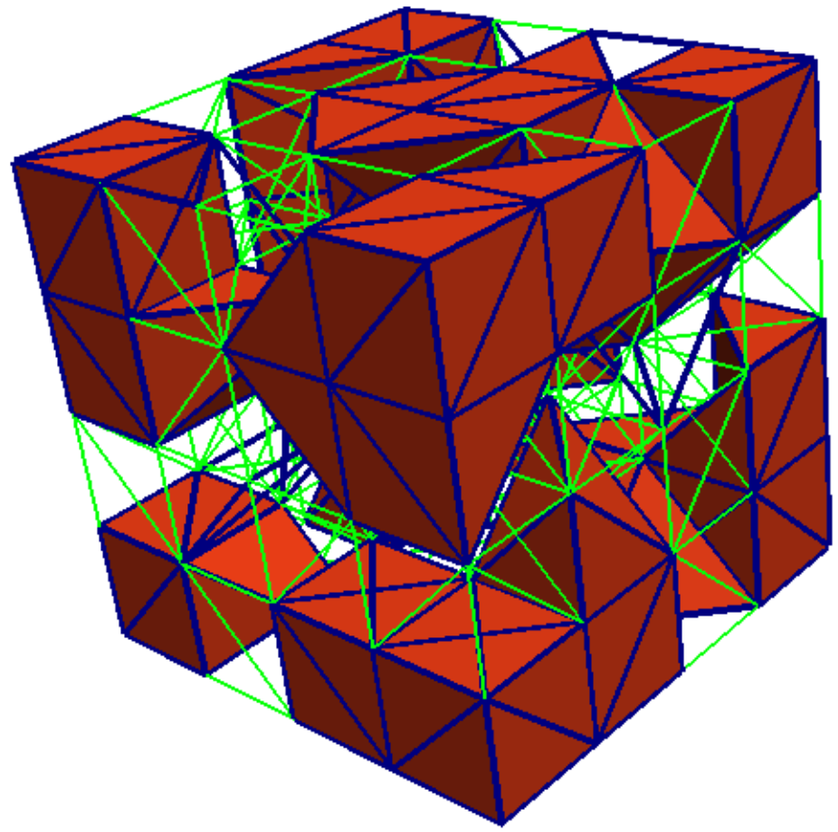
  - Repeatedly reduce size of the problem

- *Optimal Complexity*

  - O(N)

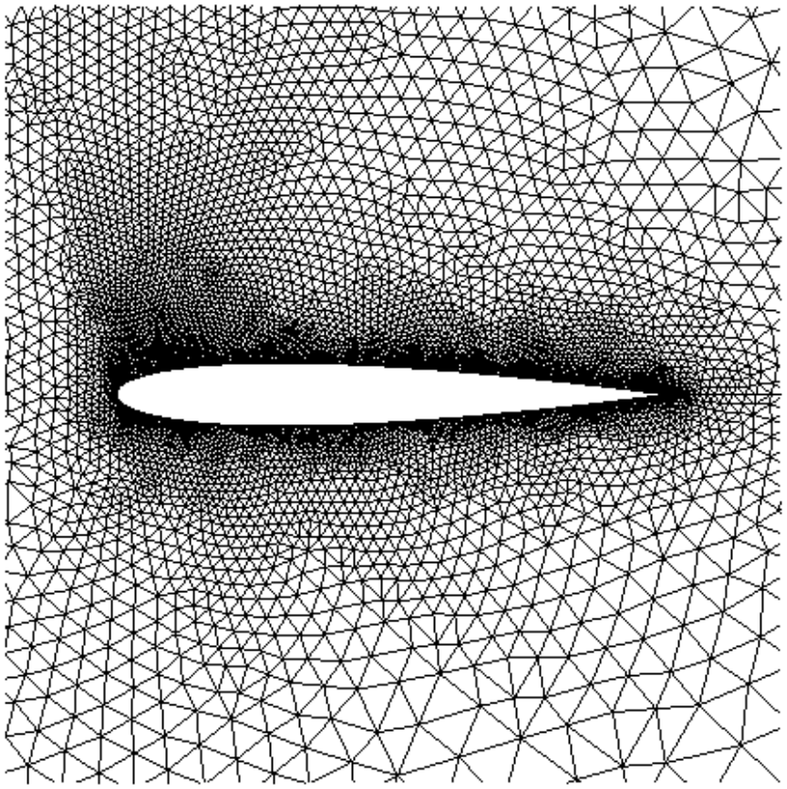
- Parallel & scalable

  - 100k+ cores (Hypre)

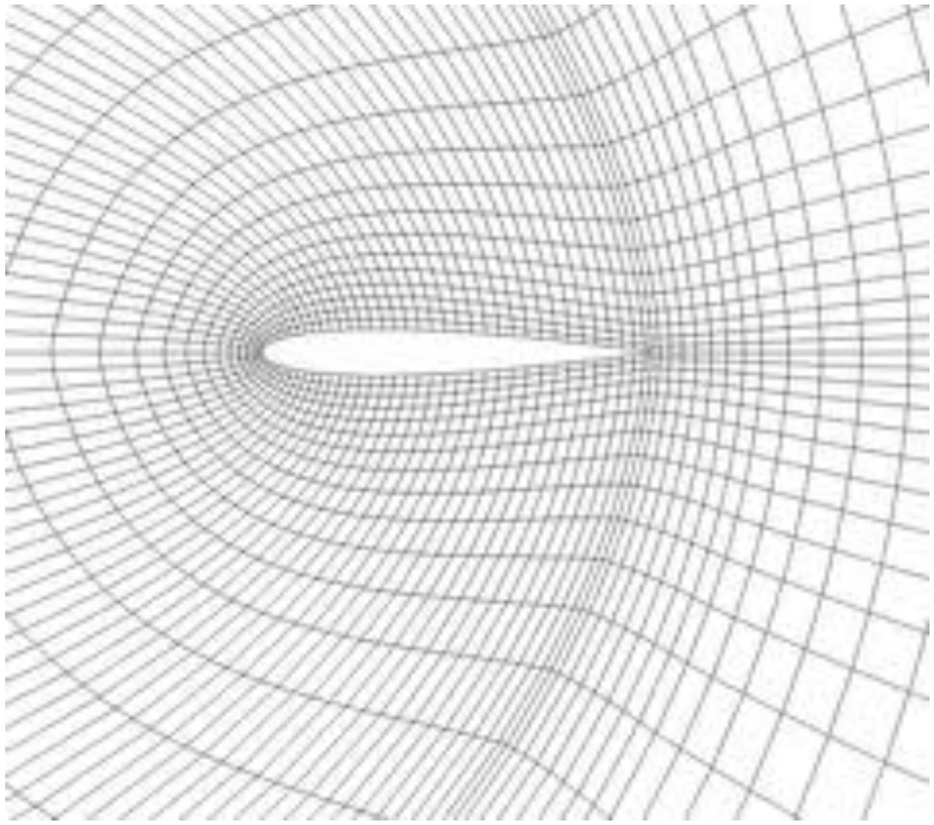


Algebraic vs. Geometric

- Coarsen from *matrix entries*
  - Not restricted to structured grids



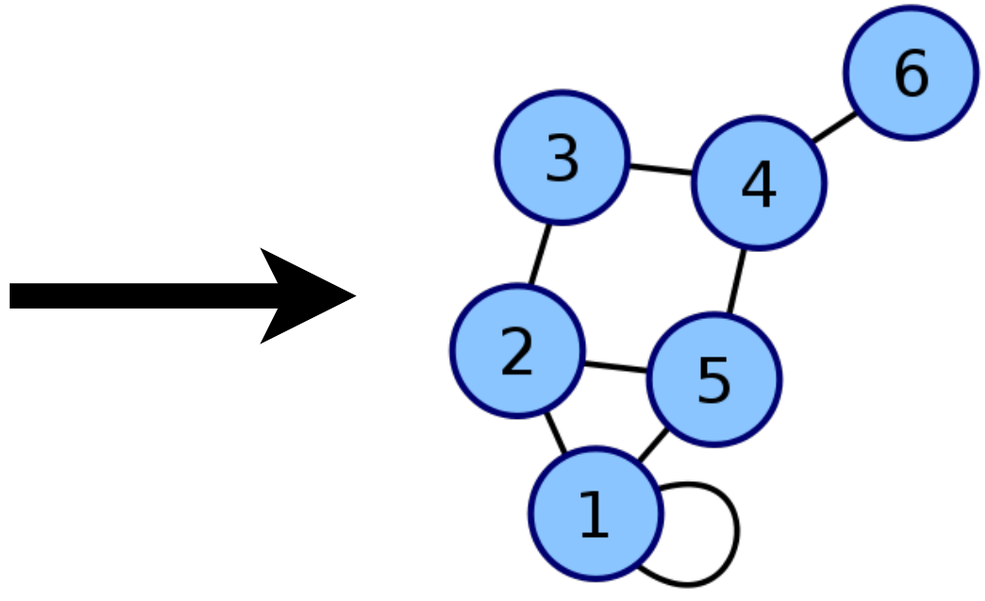
VS



Matrix as a graph

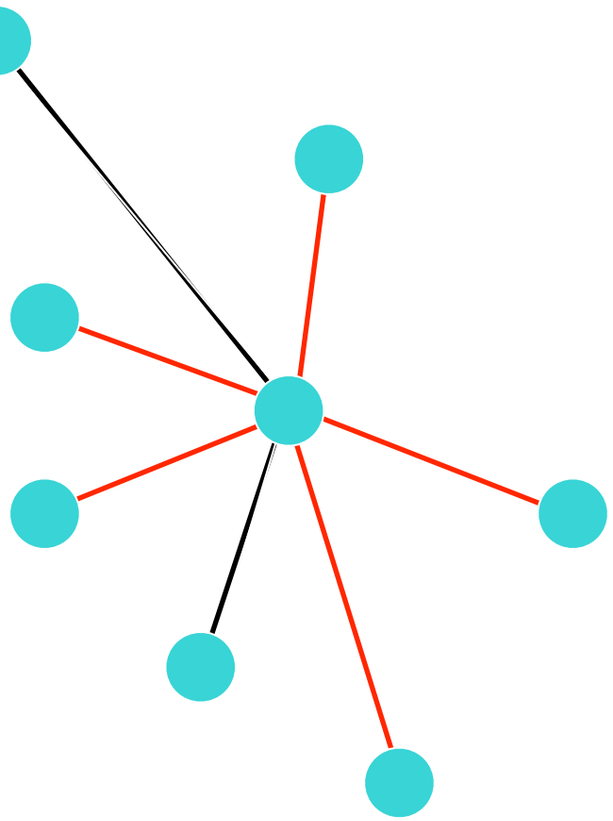
- Variables as vertices
- Non-zeros as edges

$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



Component (1) - Strength of Connection

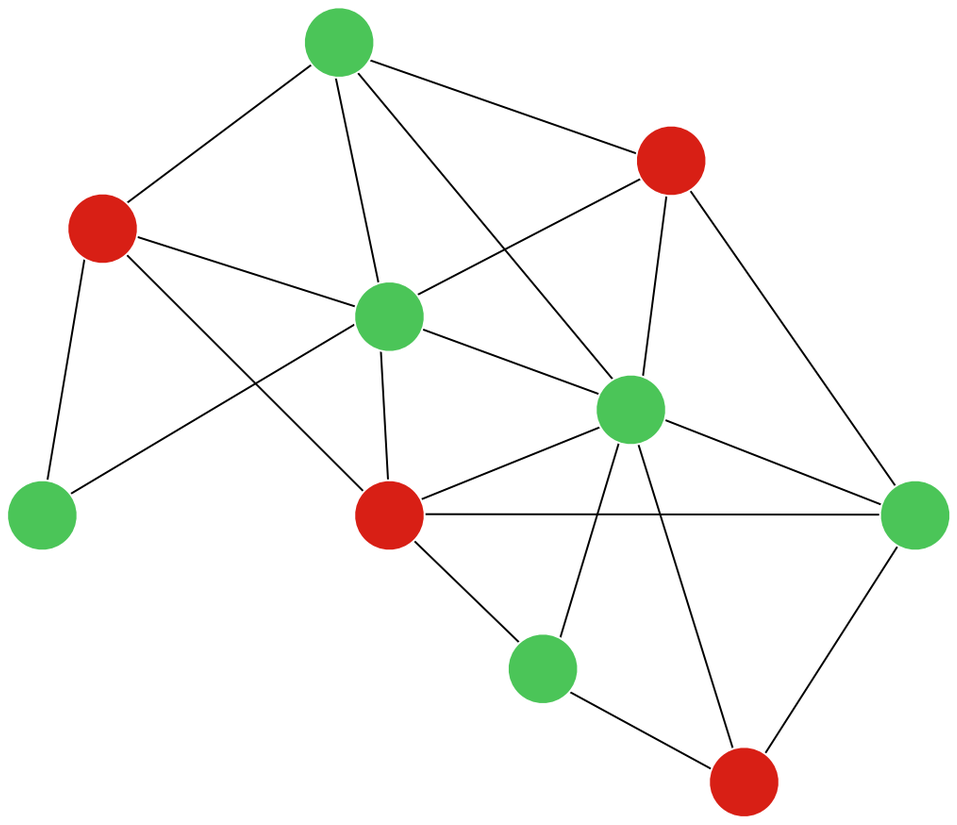
- Measure of how strongly vertices depend on each other



Each edge must
either *Strong* or *Weak*

Component (2) - Selector

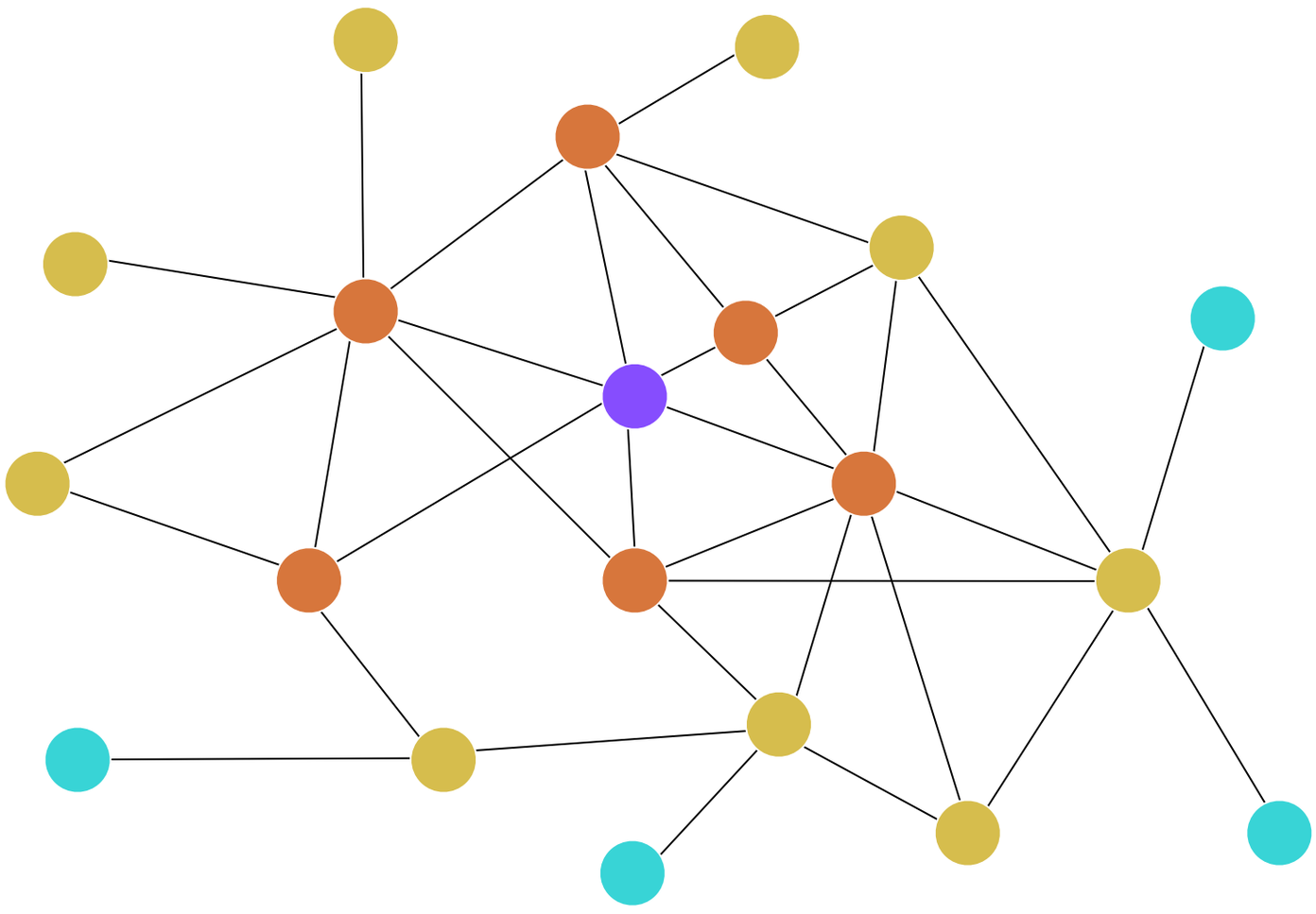
- Choose vertices with highest weights
- Weighting is # of strong edges to vertex



Component (3) - Interpolator / Restrictor

- Transfer residuals between levels

  - Construct next level



Distance 2
- Looks at neighbours of neighbours

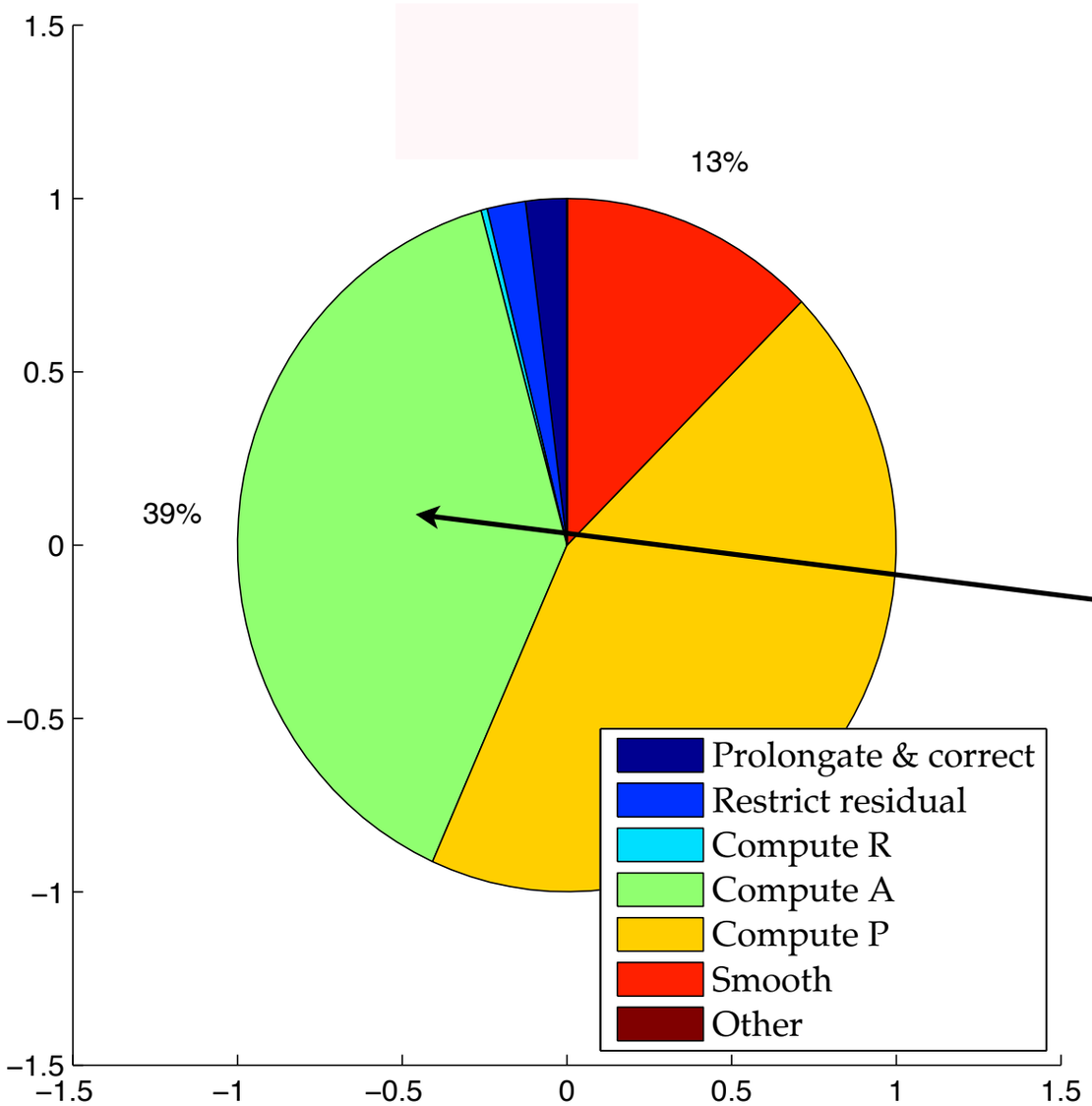
Component (4) - Galerkin Product

- Generate next level in hierarchy

$$A^{k+1} = R^k A^k P^k$$

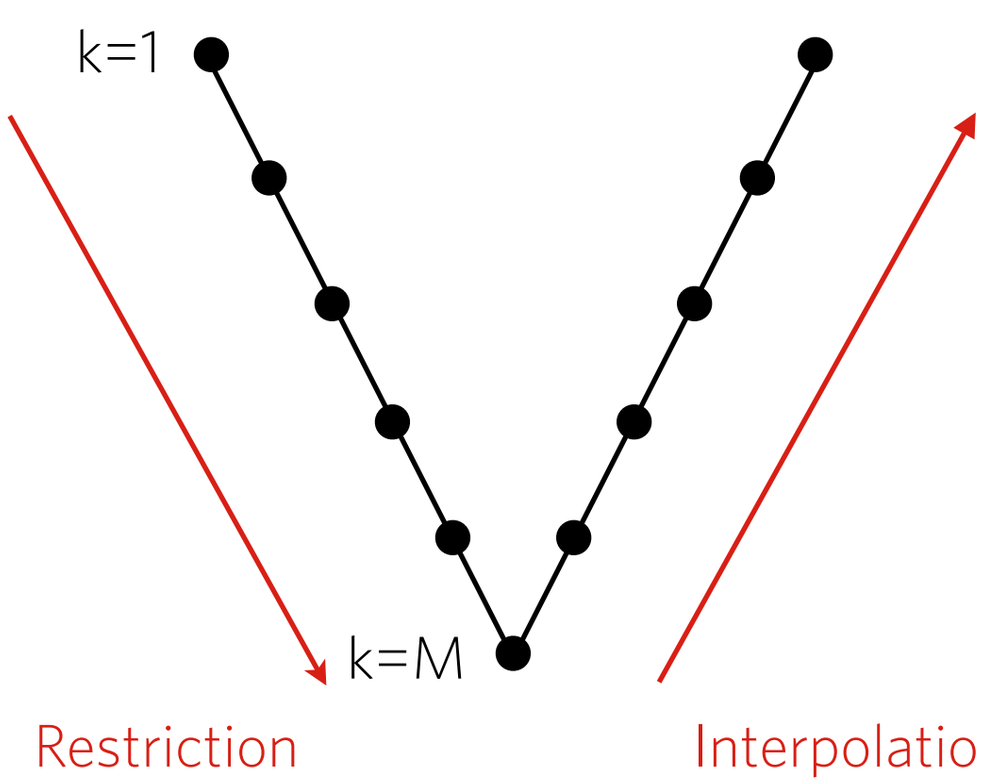
Triple matrix product

Bottleneck!



Component (5) - Solver Cycle

- Smooth errors on all levels



V-Cycle
- Simplest option
- lots of SpMV!

GPU Implementation - Justification

- Algorithm entirely parallel!
  - Fine-grained parallelism available

- If we get ~2x+ speedup, massive savings in runtime
  - Bigger runs!
  - Less time to solution!

GPU Implementation - First Thoughts

- Most operations easily expose parallelism

- Interpolator is bottleneck

- Ensure correctness

  - Compare against Hypre

  - Produce identical results

Interpolator (again)

- Interpolation weights:

$$P_{ij} = \frac{-1}{\tilde{a}_{ii}} \left(a_{ij} + \sum_{k \in F_i^s} \frac{a_{ik} \bar{a}_{ki}}{\sum_{l \in \hat{C}_i \cup \{i\}} \bar{a}_{kl}} \right), j \in \hat{C}_i$$

- Where:

$$\tilde{a}_{ii} = a_{ii} + \sum_{n \in N_i^w \setminus \hat{C}_i} a_{in} + \sum_{k \in F_i^s} a_{ik} \frac{\bar{a}_{ki}}{\sum_{l \in \hat{C}_i \cup \{i\}} \bar{a}_{kl}}$$

- And:

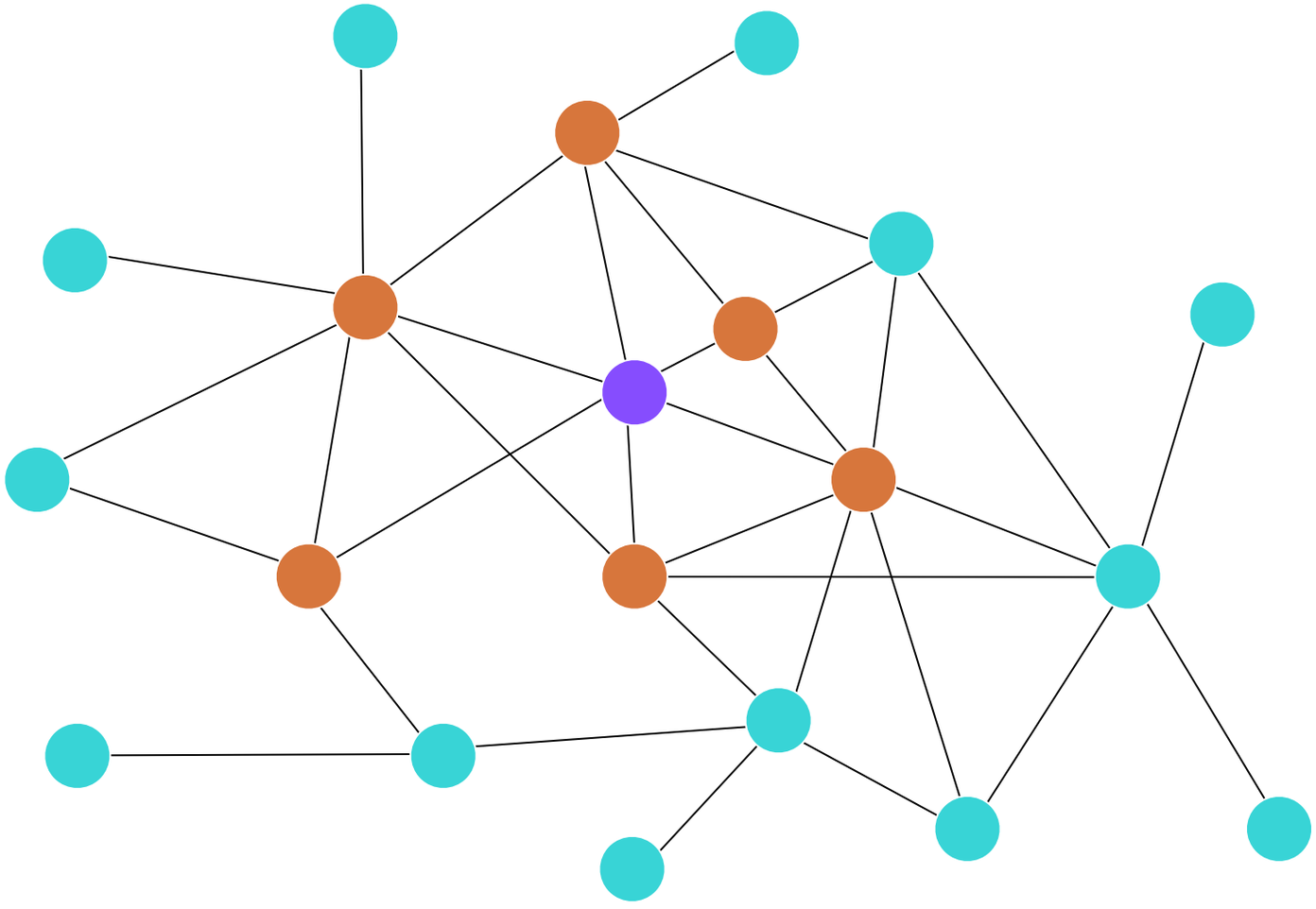
$$\hat{C}_i = C_i^s \cup \bigcup_{j \in F_i^s} C_j^s$$

- Repeated set union is tricky..

Repeated Set Union

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

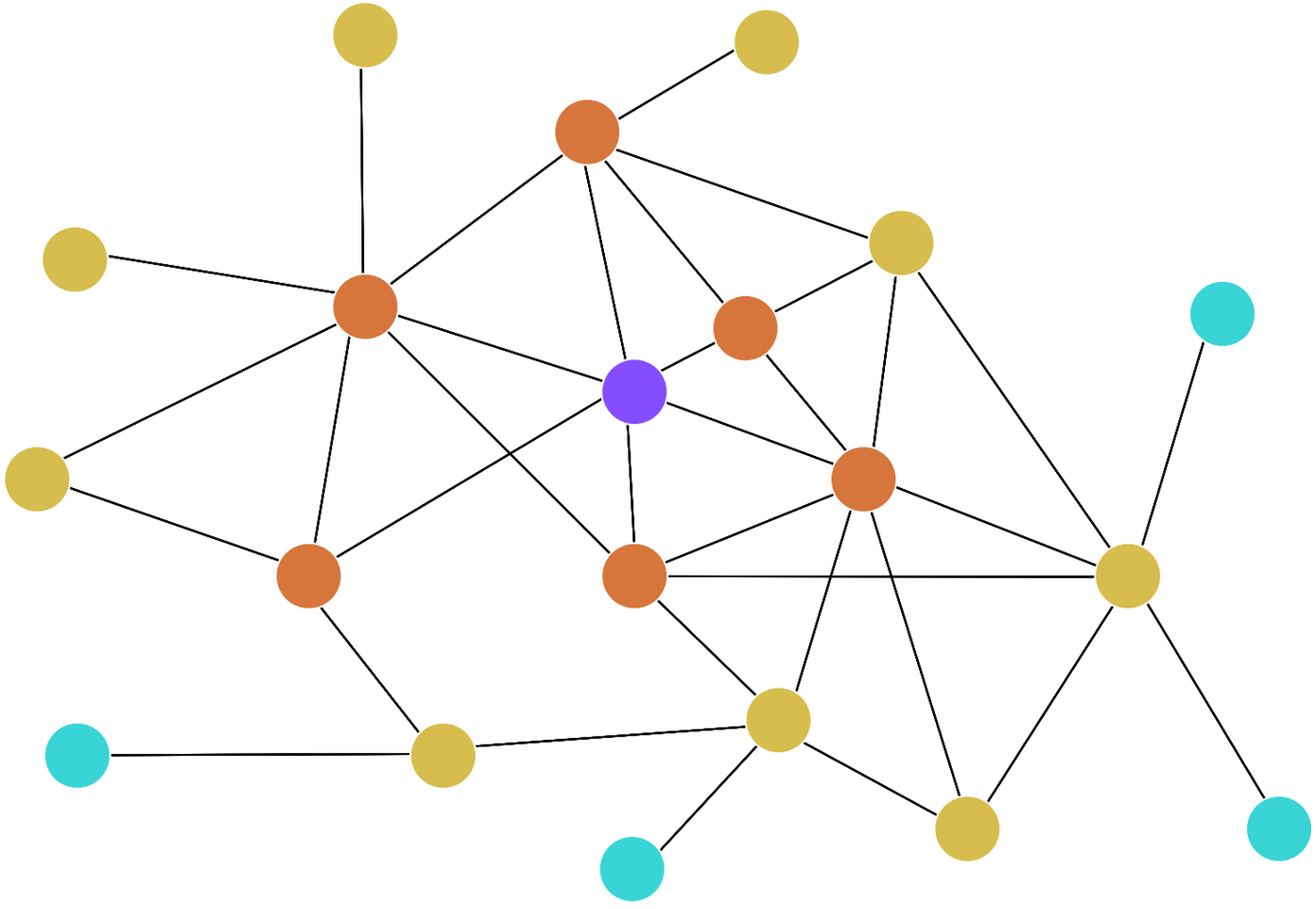
- Operation is conceptually simple



Repeated Set Union

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

- Operation is conceptually simple



Differing Approaches

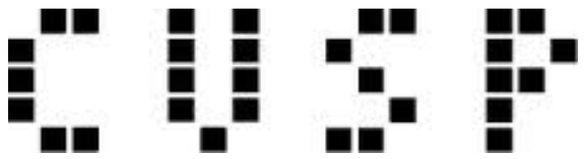
1. Worst case storage, sort & unique

   - Reliant on thrust

2. Construct boolean matrices for connections

   - Matrix-Matrix multiply

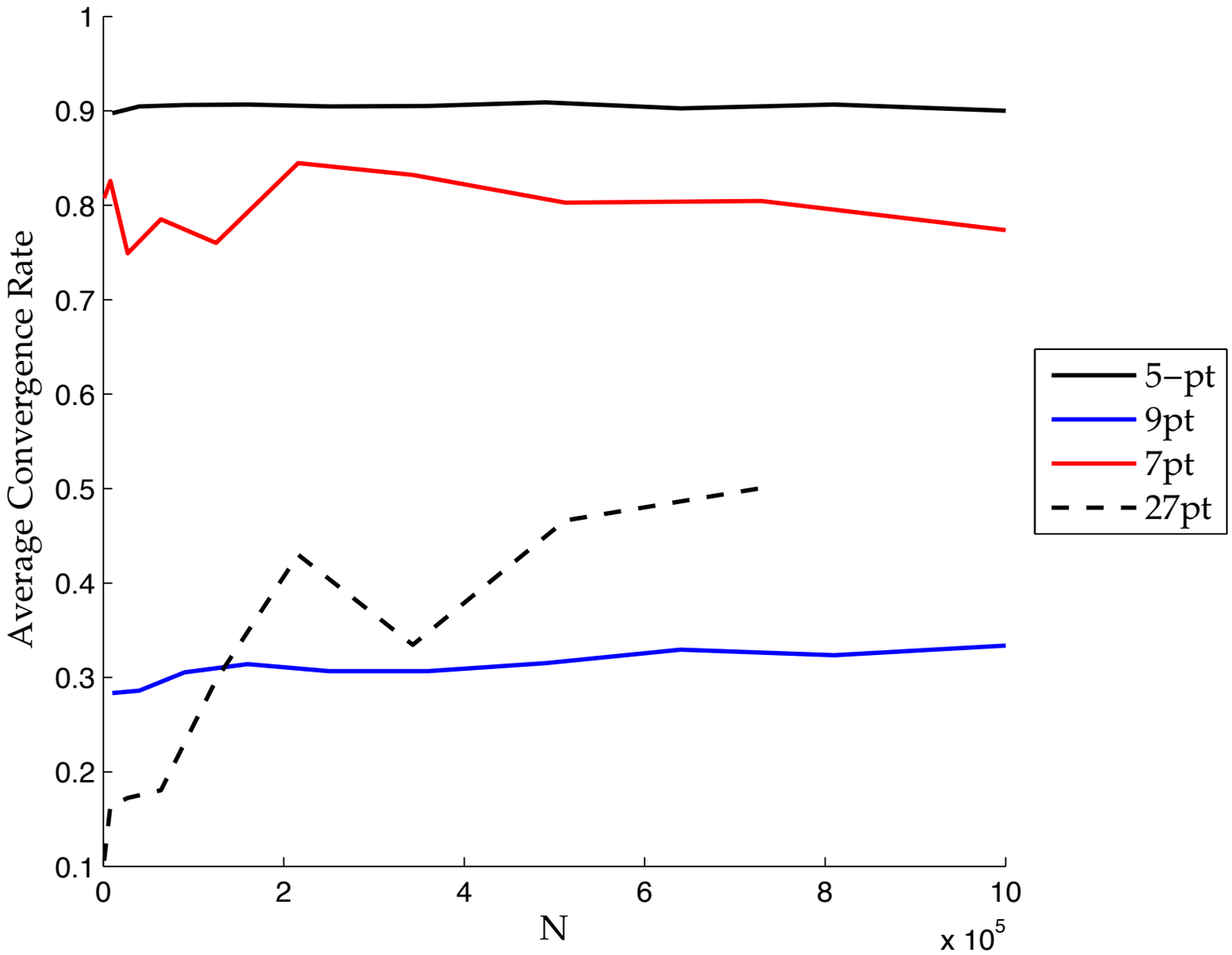
   - Reliant on Cusp



Results - Convergence

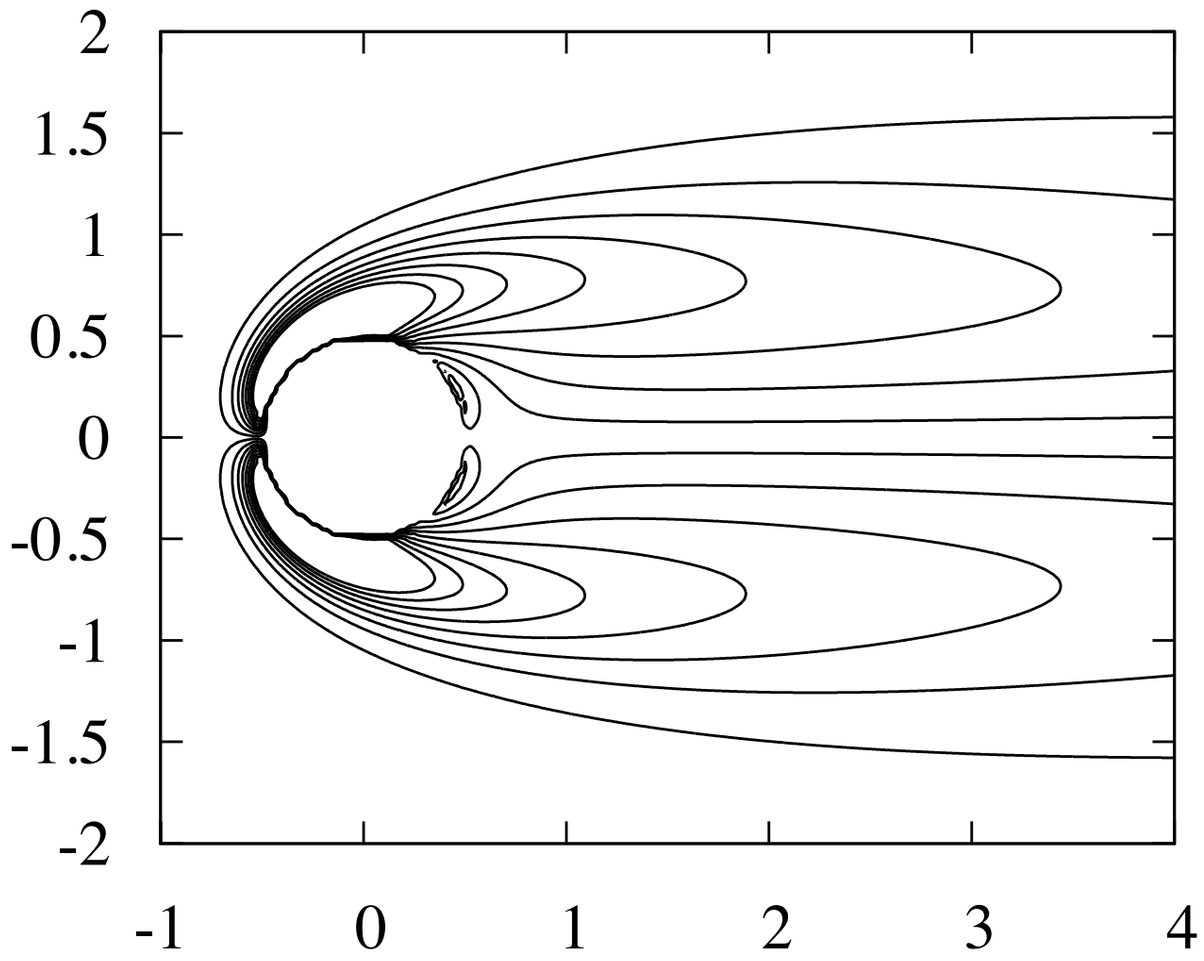
- Regular Poisson grids

  - Problem size invariance for convergence



Results - Performance

- System from slow flow past cylinder compared to Hypre



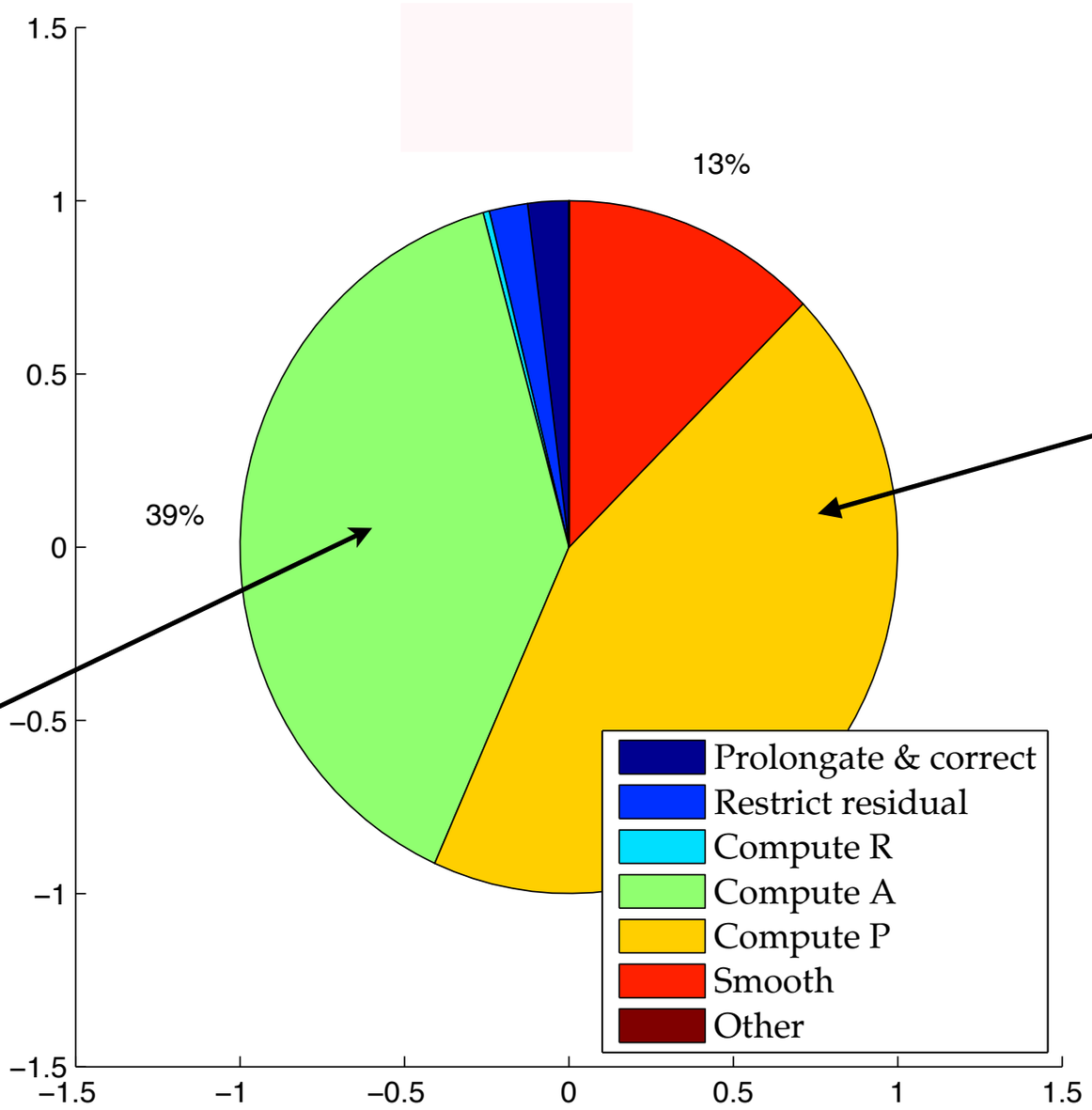
Code	*Time*	*Speedup*
CUDA	3.57s	1.87x
Hypre (1C)	6.69s	-
Hypre (2C)	5.29s	1.26x
Hypre (4C)	4.16s	1.61x
Hypre (6C)	4.00s	1.67x

Profiling

- 1,000,000 unknowns
- Error norm of 10^{-5}
- Single Tesla C2050 GPU

Profiling - Breakdown by Routine

- Generating Interpolation matrix, coarse A most expensive



Interpolation matrix

Triple matrix product - coarse A

Conclusions & Further Work

- Classical AMG entirely on GPU
  - Validated against Hypre
  - Not optimised

- Multiple approaches
  - Test different methods & compare