FEM Integration with Quadrature and Preconditioners on GPUs

Karl Rupp
https://karlrupp.net/

now:
Freelance Scientist

formerly:
Institute for Microelectronics, TU Wien

in collaboration with:
M. Knepley (Rice U.), A. Terrel (Fashion Metric), J. Weinbub,
F. Rudolf, A. Morhammer, T. Grasser, A. Jüngel (TU Wien)

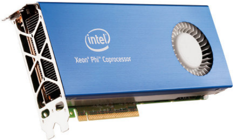
Platform for Research in Simulation Methods
Workshop on Embracing Accelerators
Imperial College, London
April 18th, 2016

Introduction

Recent Many-Core Architectures

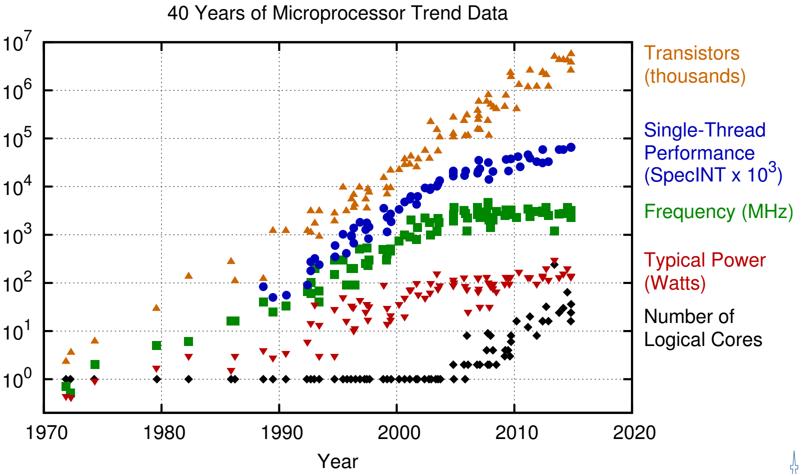
- High FLOP/Watt ratio
- High memory bandwidth
- Attached via PCI-Express

AMD FirePro W9100
320 GB/sec



INTEL Xeon Phi
320 (220?) GB/sec

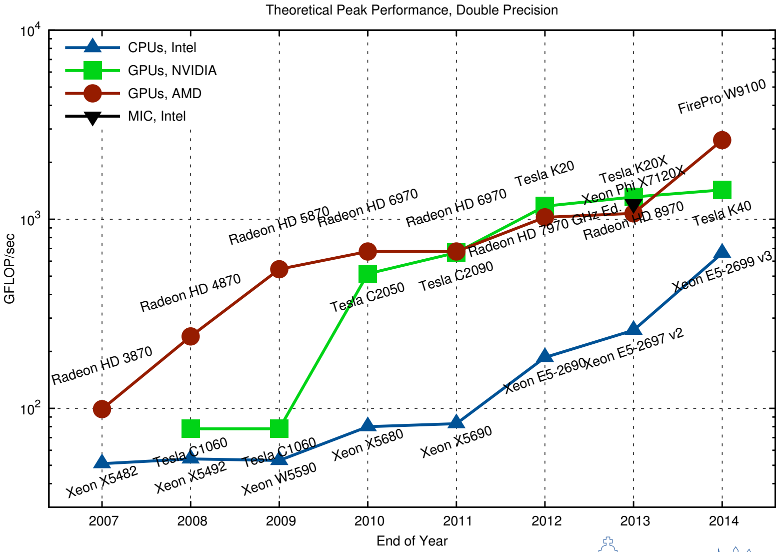
NVIDIA Tesla K20
250 (208) GB/sec



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

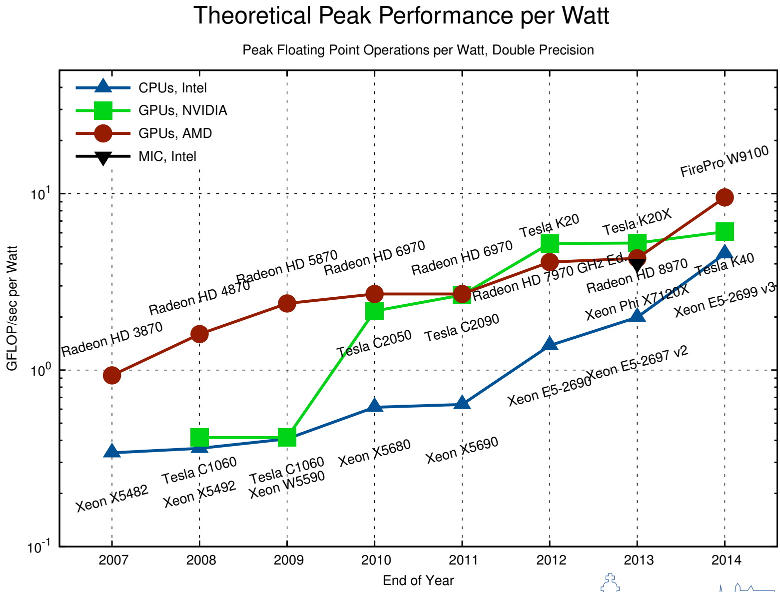
https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/

Theoretical Peak Performance



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

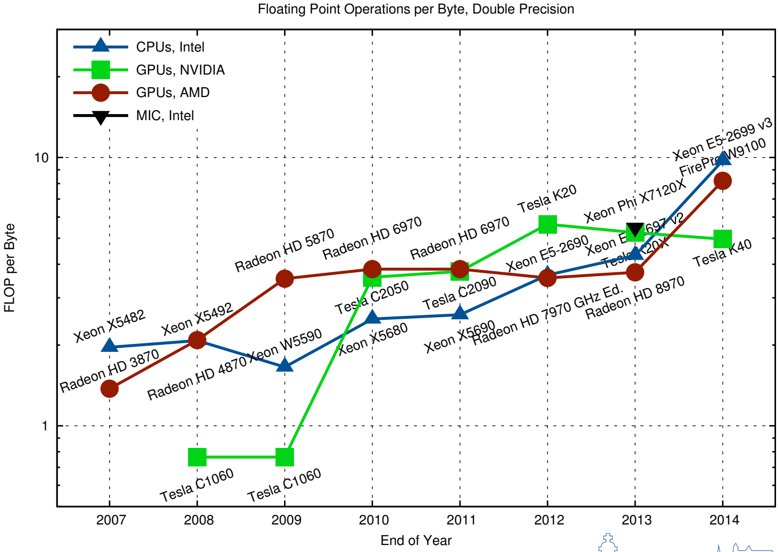
Introduction



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Introduction

Theoretical Peak Performance (FLOPs) per Byte of Memory Bandwidth



https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/

Part 1: FEM Integration with Quadrature

Finite Element Method

- Several basis functions per element
- Evaluation of integrals on each element

General Weak Form

- Residual formulation for test function ϕ

$$\int_\Omega \phi \cdot f_0(u, \nabla u) + \nabla \phi : \mathbf{f}_1(u, \nabla u) = 0.$$

Examples

- Laplace: $f_0 \equiv 0$, $\mathbf{f}_1 \equiv \nabla u$
- Poisson: $f_0 \equiv g$, $\mathbf{f}_1 \equiv \nabla u$

$$\int_\Omega \phi \cdot f_0(u, \nabla u) + \nabla \phi : \mathbf{f}_1(u, \nabla u) = 0.$$

Element-Wise General Weak Form

Evaluation using quadrature

$$\sum_e \mathcal{E}_e^T \left[B^T W f_0(u^q, \nabla u^q) + \sum_k D_k^T W \mathbf{f}_1^k(u^q, \nabla u^q) \right] = 0$$

$\mathcal{E}$ ... global vector

W ... quadrature weights

B, D_k ... reduction operations for global basis coefficients

Parallelization Options

Across elements

Quadrature points

Basis functions

Parallelization Across Elements

- Large memory per thread
- Synchronizations with neighbor elements
- [Cecka et al. 2011; Taylor et al. 2008; Williams 2012]

Parallelization per Quadrature Point

- No memory overhead
- Too many synchronizations

Parallelization via Basis Functions

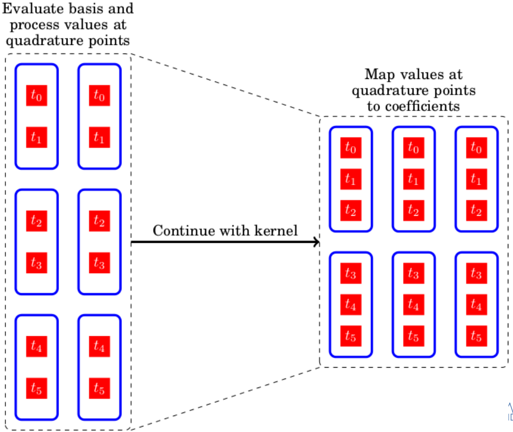
- Very little local memory
- Repeated loads of coefficients from global memory
- [Dabrowski et al. 2008]

Thread Block Works on Multiple Elements

Number of quadrature points N_q

Number of basis functions N_b

Minimum number of elements $\mathrm{LCM}(N_\mathrm{q}, N_\mathrm{b})$



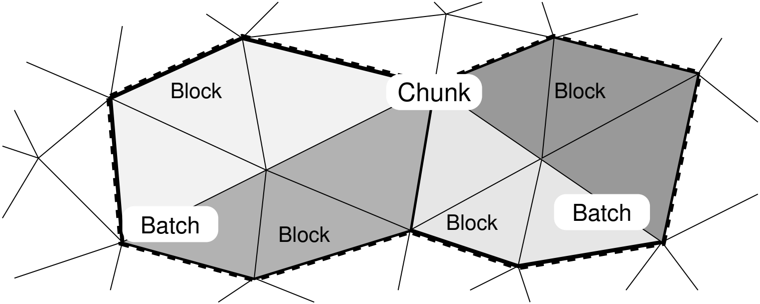
New Algorithm

High Level Decomposition

Chunks - Cells processed by each thread workgroup

Batches - Cells processed with one thread transposition

Blocks - Smallest unit of execution



Benchmark

OpenCL-enabled Hardware

- NVIDIA GTX 470
- NVIDIA GTX 580
- NVIDIA Tesla K20m
- AMD FirePro W9100
- (AMD A10-5800K)

Comparisons

- Single vs. double precision
- 2D vs. 3D

Invariants

- Variable coefficients
- First-order FEM
- Poisson equation

Benchmark

Choice of Block and Batch Numbers

NVIDIA GTX 470

Performance in GFLOPs/sec

Actual choice not very sensitive

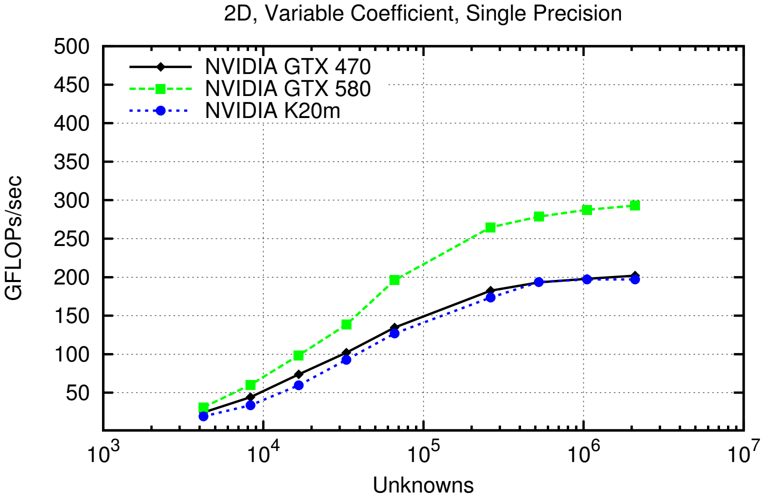
Blocks	Batches 16	20	24	28	32	36
4	113	120	118	122	**137**	119
8	109	116	113	120	108	117
12	102	112	110	109	115	113
16	108	100	99	111	130	106

(2D triangular mesh, variable coefficients, single precision, NVIDIA GTX 470)

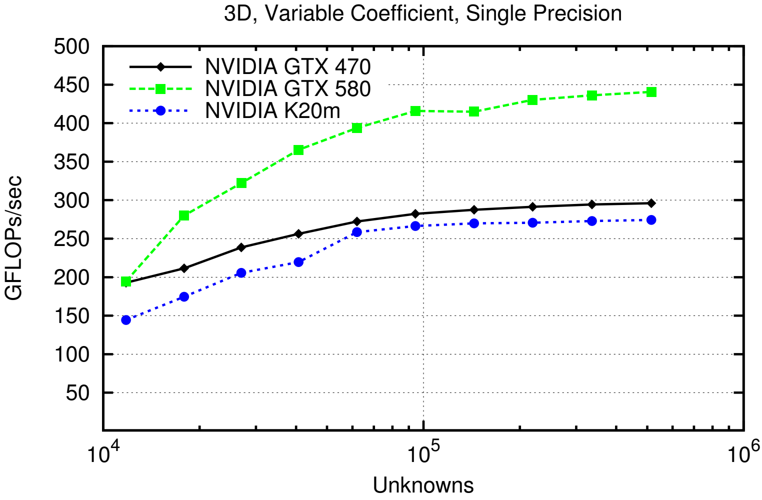
PETSc SNES ex12:

```
./ex12 -petscspace_order 1 -run_type perf -variable_coefficient field
-refinement_limit 0.00001 -show_solution false -petscfe_type opencl
-petscfe_num_blocks 4 -petscfe_num_batches 16
```

Benchmark

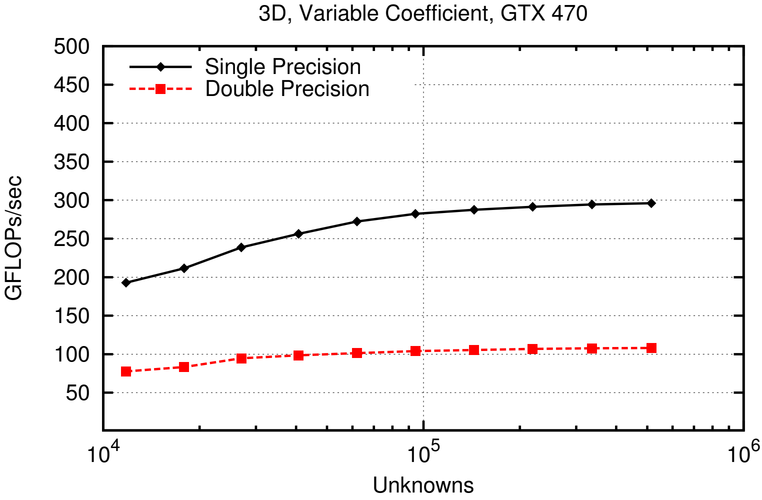

2D, Variable Coefficient, Single Precision
NVIDIA GTX 470
NVIDIA GTX 580
NVIDIA K20m
GFLOPs/sec
500
450
400
350
300
250
200
150
100
50
10^3
10^4
10^5
10^6
10^7
Unknowns

Benchmark

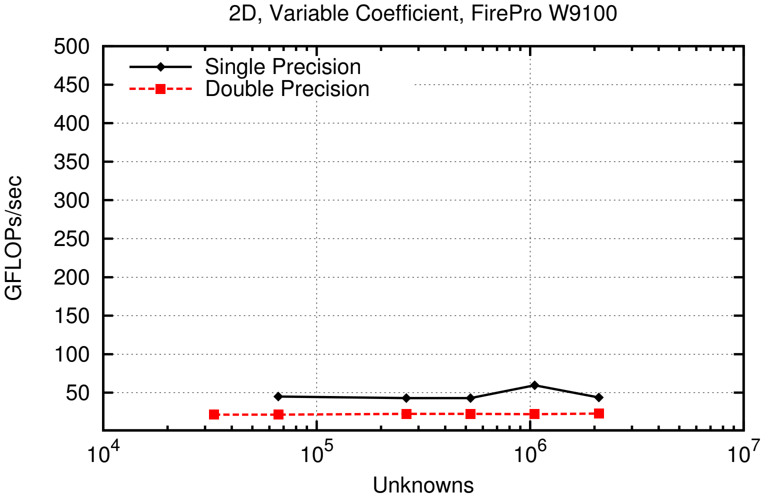


3D, Variable Coefficient, Single Precision
500
450
400
350
300
250
200
150
100
50
GFLOPs/sec
NVIDIA GTX 470
NVIDIA GTX 580
NVIDIA K20m
10^4
10^5
10^6
Unknowns

Benchmark

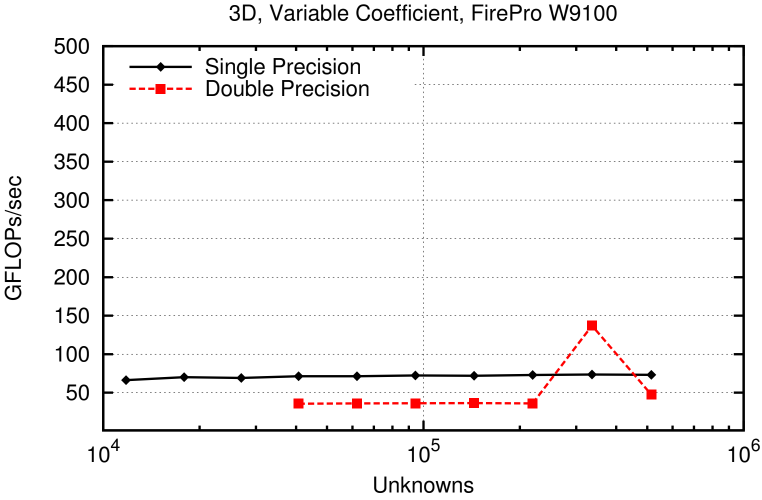

3D, Variable Coefficient, GTX 470
Single Precision
Double Precision
GFLOPs/sec
500
450
400
350
300
250
200
150
100
50
10^4
10^5
10^6
Unknowns

Benchmark



2D, Variable Coefficient, FirePro W9100
Single Precision
Double Precision
GFLOPs/sec
500
450
400
350
300
250
200
150
100
50
10^4
10^5
10^6
10^7
Unknowns

Benchmark


3D, Variable Coefficient, FirePro W9100
Single Precision
Double Precision
GFLOPs/sec
500
450
400
350
300
250
200
150
100
50
10^4
10^5
10^6
Unknowns

Performance Modeling

Limiting Factor?

GTX 470: 134 GB/sec memory bandwidth (theoretical)

GTX 470: 1088 GFLOPs/sec peak (theoretical)

Arithmetic Intensity

Count FLOPs and bytes loaded/stored

$$\beta = \frac{\left[(2+(2+2d)d)N_{\mathrm{bt}}N_{\mathrm{q}} + 2dN_{\mathrm{comp}}N_{\mathrm{q}} + (2+2d)dN_{\mathrm{q}}N_{\mathrm{bt}}\right]N_{\mathrm{bs}}N_{\mathrm{bl}}}{4N_{\mathrm{t}}\left((d^2+1)+N_{\mathrm{bt}}+(d+1)N_{\mathrm{q}}\right)}$$

2D Mesh, First-Order FEM, Single Precision

$\beta = 41/22 \approx 2$ FLOPs/Byte

GTX 470: $134 \times 41/22 = 250$ GFLOPs possible

GTX 470: 200 GFLOPs achieved (80 percent, cf. STREAM benchmark)

Summary - FEM Quadrature

FEM Quadrature on GPUs

- “Matrix-Free”
- Higher arithmetic intensity

Performance Results

- Good performance on NVIDIA GPUs and AMD APUs
- 5x improvements for discrete AMD GPUs desired

Performance Modeling

- Performance limited by memory bandwidth
- Excellent prediction accuracy

Reproducibility

- PETSc, SNES tutorial, ex12

Part 2: Solvers and Preconditioners

Pipelined CG

- Merge global reductions
- Kernel fusion

Parallel Incomplete LU Factorizations

- Level scheduling
- Nonlinear relaxation

Algebraic Multigrid

- Parallel aggregation
- Sparse matrix-matrix products

Performance Modeling: Conjugate Gradients

Pseudocode

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

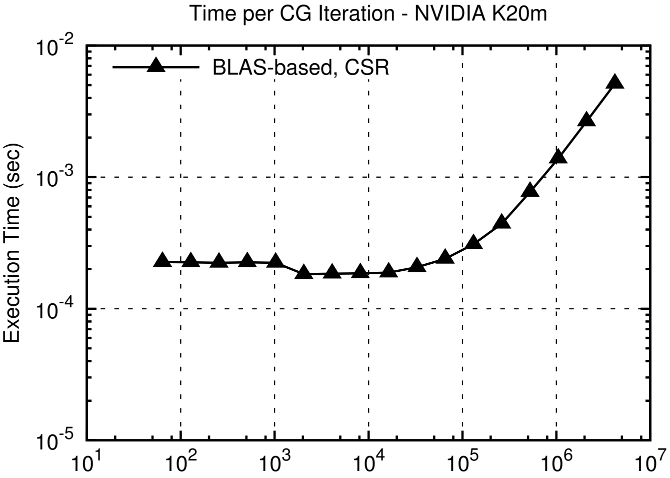
BLAS-based Implementation

-
SpMV, AXPY
For $i = 0$ until convergence

1. SpMV $\leftarrow$ No caching of Ap_i
2. DOT $\leftarrow$ Global sync!
3. -
4. AXPY
5. AXPY $\leftarrow$ No caching of r_{i+1}
6. DOT $\leftarrow$ Global sync!
7. -
8. AXPY

EndFor

Performance Modeling: Conjugate Gradients



(Poisson, 2D, Finite Differences)

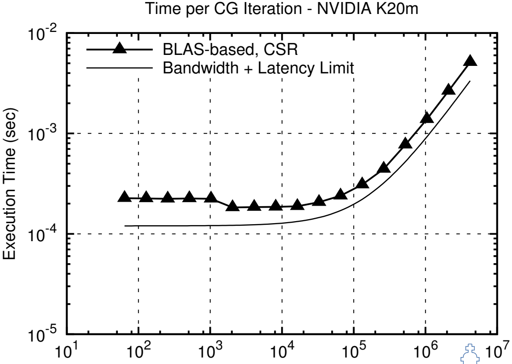
Performance Modelling

6 Kernel Launches (plus two for reductions)

Two device to host data reads from dot products

Model SpMV as seven vector accesses (5-point stencil)

$$T(N) = 8 \times 10^{-6} + 2 \times 2 \times 10^{-6} + (7+2+3+3+2+3) \times 8 \times N/\text{Bandwidth}$$



Optimization: Rearrange the algorithm

- Remove unnecessary reads
- Remove unnecessary synchronizations
- Use custom kernels instead of standard BLAS

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

EndFor

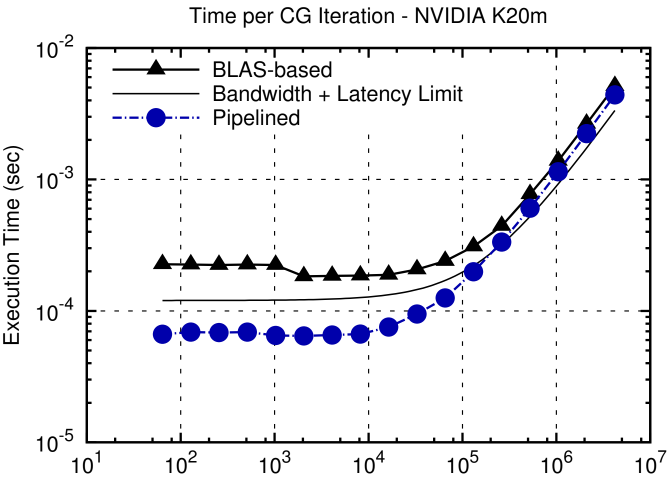
Pipelined CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence

1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

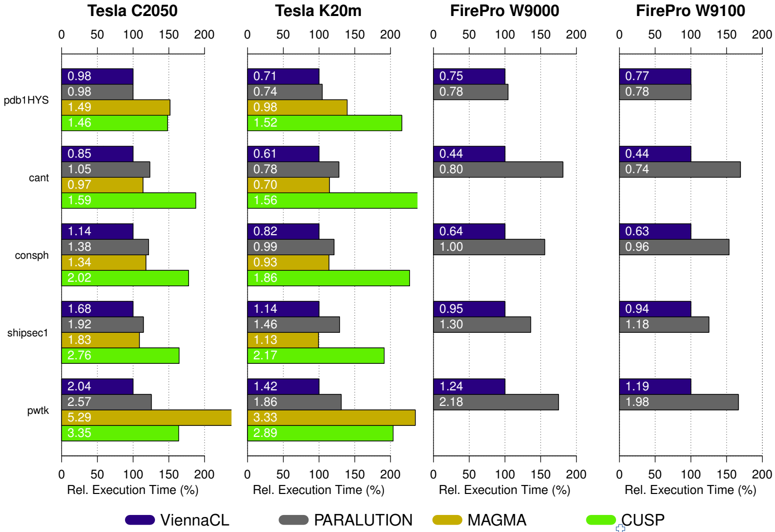
EndFor

Performance Modeling: Conjugate Gradients



(Poisson, 2D, Finite Differences)

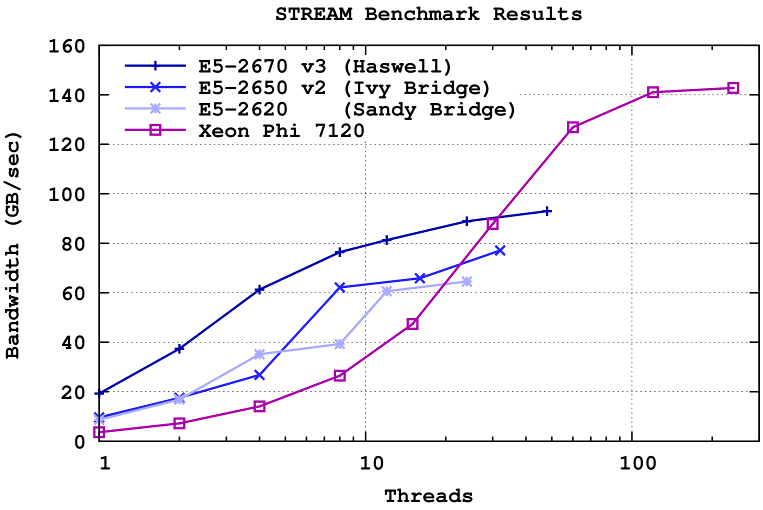
Benefits of Pipelining also for Large Matrices



Parallel Incomplete LU Factorizations

- Level scheduling
- Nonlinear relaxation

Memory Bandwidth vs. Parallelism



https://www.karlrupp.net/2015/02/stream-benchmark-results-on-intel-xeon-and-xeon-phi/

Parallel ILU

ILU - Basic Idea

Factor sparse matrix $A \approx \tilde{L}\tilde{U}$

$\tilde{L}$ and $\tilde{U}$ sparse, triangular

ILU0: Pattern of $\tilde{L}$, $\tilde{U}$ equal to A

ILUT: Keep k elements per row

Solver Cycle Phase

Residual correction $\color{red}{\tilde{L}}\color{black}{\tilde{U}x = z}$

Forward solve $\color{red}{\tilde{L}}\color{black}{y = z}$

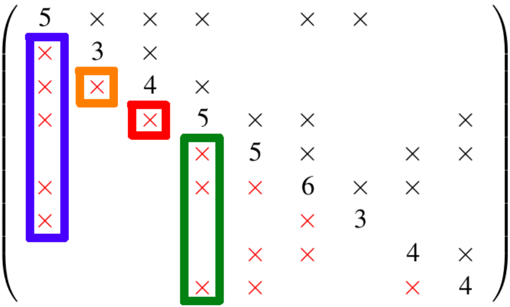
Backward solve $\tilde{U}x = y$

Little parallelism in general

$$\begin{pmatrix}
5 & \times & \times & \times & & \times & \times & & \\
\color{red}\times & 3 & \times & & & & & & \\
\color{red}\times & \color{red}\times & 4 & \times & & & & & \\
\color{red}\times & & \color{red}\times & 5 & \times & \times & & & \times \\
& & & \color{red}\times & 5 & \times & & \times & \times \\
\color{red}\times & & & \color{red}\times & \color{red}\times & 6 & \times & \times & \\
\color{red}\times & & & & & \color{red}\times & 3 & & \\
& & & & \color{red}\times & \color{red}\times & & 4 & \times \\
& & & \color{red}\times & \color{red}\times & & & \color{red}\times & 4
\end{pmatrix}$$

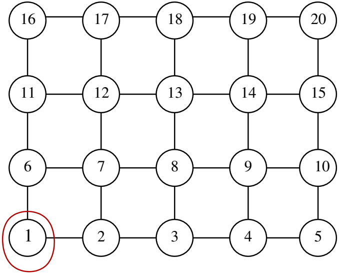
ILU Level Scheduling

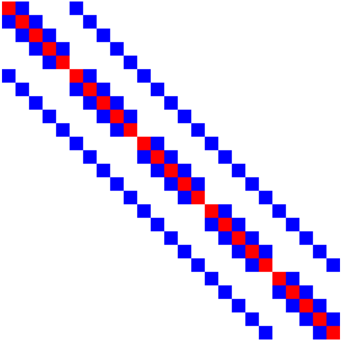
- Build dependency graph
- Substitute as many entries as possible simultaneously
- Trade-off: Each step vs. multiple steps in a single kernel



ILU Interpretation on Structured Grids

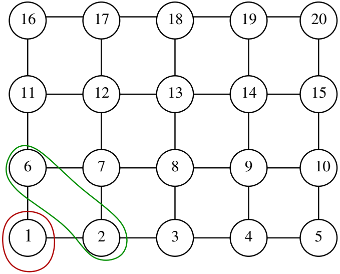
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

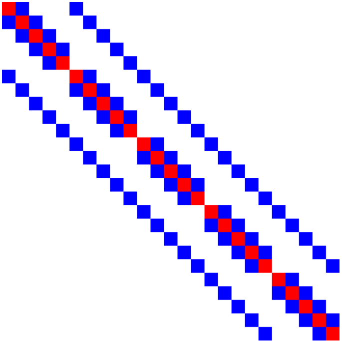




ILU Interpretation on Structured Grids

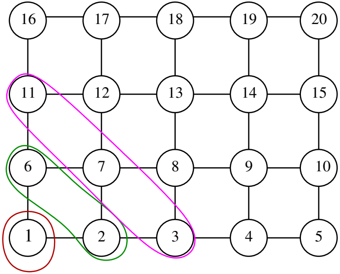
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

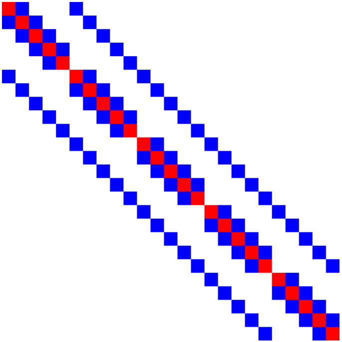




ILU Interpretation on Structured Grids

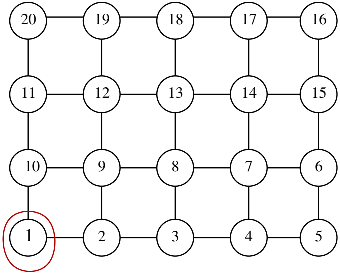
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

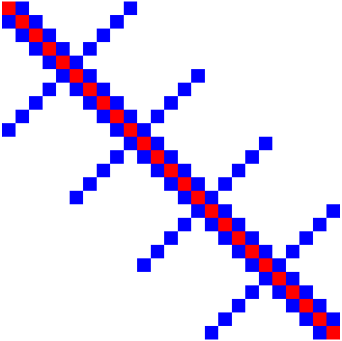




ILU Interpretation on Structured Grids

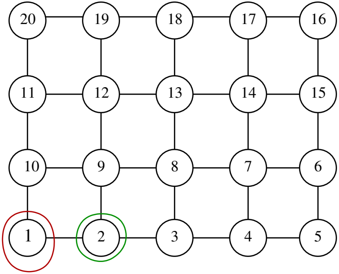
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

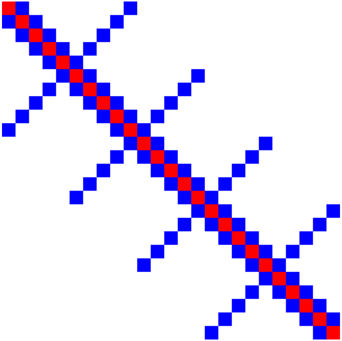




ILU Interpretation on Structured Grids

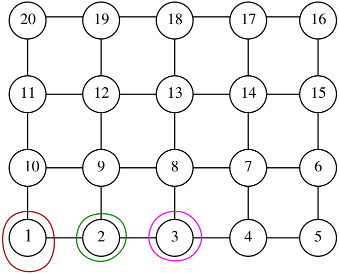
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

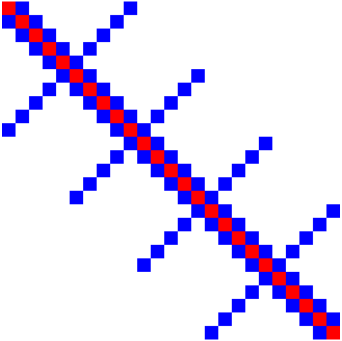




ILU Interpretation on Structured Grids

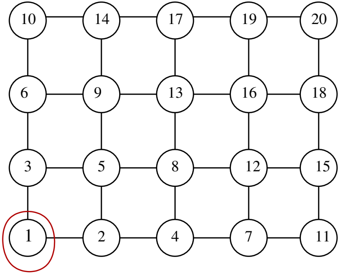
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

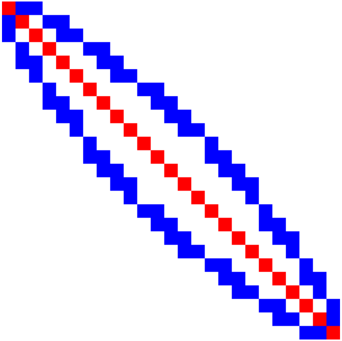




ILU Interpretation on Structured Grids

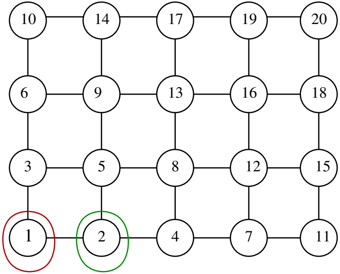
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

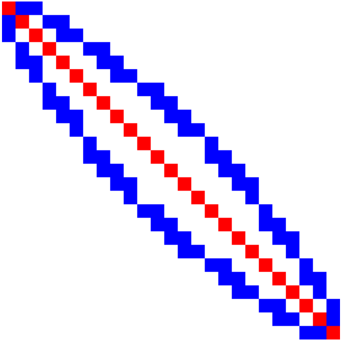




ILU Interpretation on Structured Grids

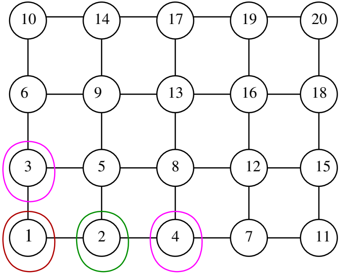
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

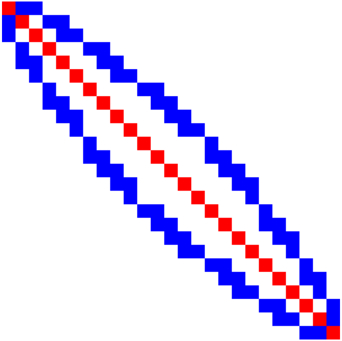




ILU Interpretation on Structured Grids

- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d



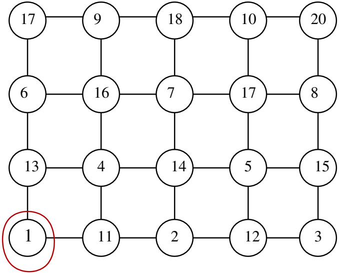


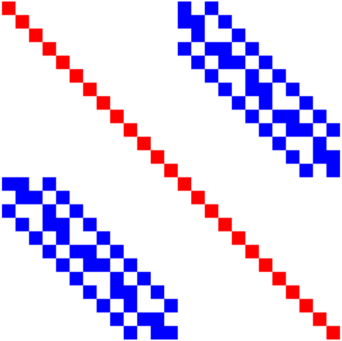
ILU Interpretation on Structured Grids

2d finite-difference discretization

Substitution whenever all neighbors with smaller index computed

Works particularly well in 3d

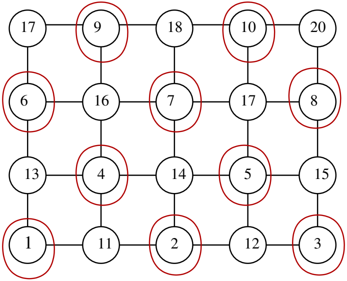


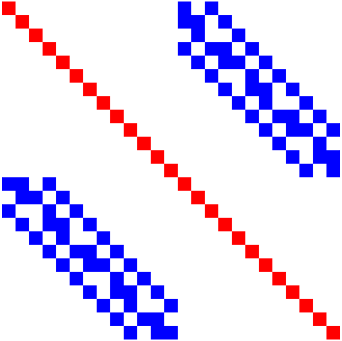


Parallel ILU

ILU Interpretation on Structured Grids

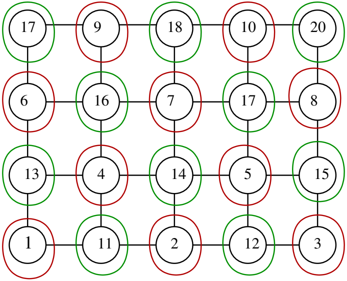
- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d

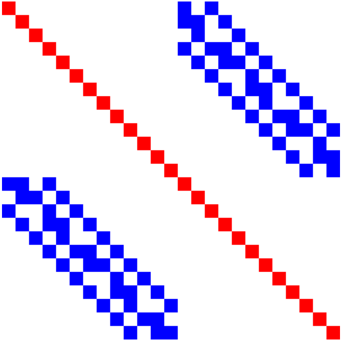




ILU Interpretation on Structured Grids

- 2d finite-difference discretization
- Substitution whenever all neighbors with smaller index computed
- Works particularly well in 3d





Parallel ILU

Sequential

for i=2..n
  for k=1..i-1, (i,k)in A
    $a_{ik} = a_{ik}/a_{kk}$
    for j=k+1..n, (i,j)in A
      $a_{ij} = a_{ij} - a_{ik}a_{kj}$

Parallel

for (sweep = 1, 2, ...)
  parallel **for** (i,j)in A
    if (i > j)
      $l_{ij} = (a_{ij} - \sum_{k=1}^{j=1} l_{ik}u_{kj})/u_{jj}$
    else
      $u_{ij} = a_{ij} - \sum_{k=1}^{=1} l_{ik}u_{kj}$

Fine-Grained Parallel ILU Setup

Proposed by Chow and Patel (SISC, vol. 37(2)) for CPUs and MICs

Massively parallel (one thread per row)

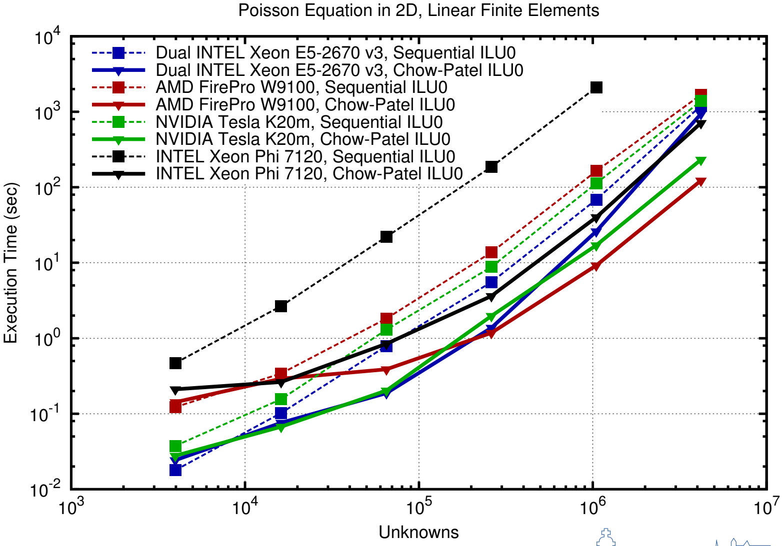
Preconditioner Application

Truncated Neumann series:

$$\mathbf{L}^{-1} \approx \sum_{k=0}^{K} (\mathbf{I} - \mathbf{L})^k, \quad \mathbf{U}^{-1} \approx \sum_{k=0}^{K} (\mathbf{I} - \mathbf{U})^k$$

Exact triangular solves not necessary

Parallel ILU



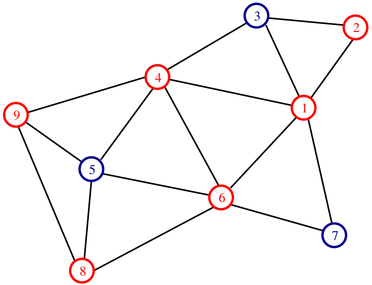
Content

Algebraic Multigrid

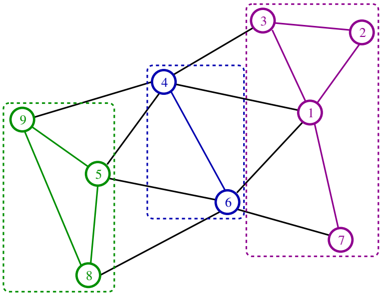
- Parallel aggregation
- Sparse matrix-matrix products

Ingredients of Algebraic Multigrid

- Smoother (Relaxation schemes, etc.)
- Coarsening
- Interpolation (Inter-grid transfer)



Classical coarsening

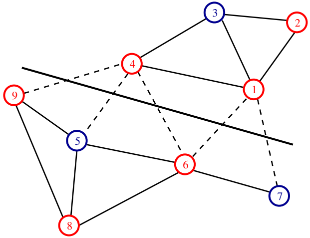


Aggregation coarsening

Multigrid Parallelization

Setup Phase

- Determination of coarse points in parallel by graph splitting
- Compute coarse operators $A^{k+1} = R^k A^k P^k$ (where $A^0 = A$)
- Datastructures: analyze and allocate
- Limited fine-grained parallelism



Cycle Phase

- Parallel Jacobi Smoother
- Restriction $R^k x^k$, prolongation $P^k x^{k+1}$
- Direct solution on coarsest level
- Static datastructures
- Enough fine-grained parallelism

AMG Sparse Matrix-Matrix Multiplication

Coarse Grid Operator

$A^{\text{coarse}} = RA^{\text{fine}}P$

Common choice: $R = P^{\mathrm{T}}$

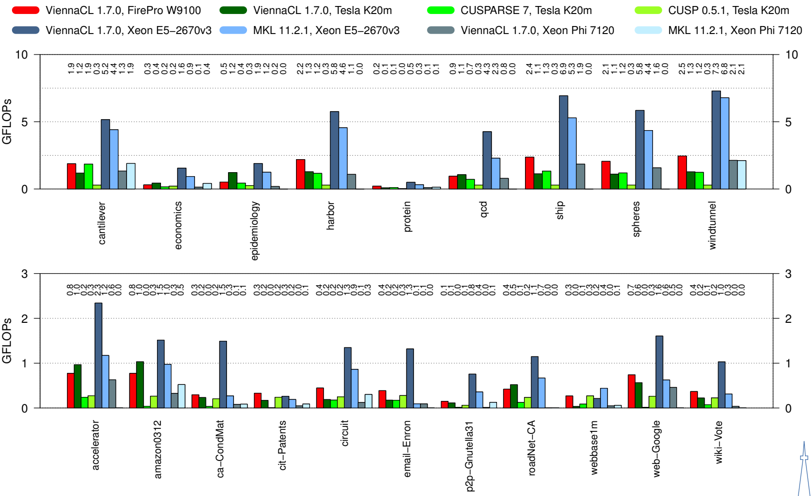
Computation

Explicitly set up $R = P^{\mathrm{T}}$ (hard in parallel)

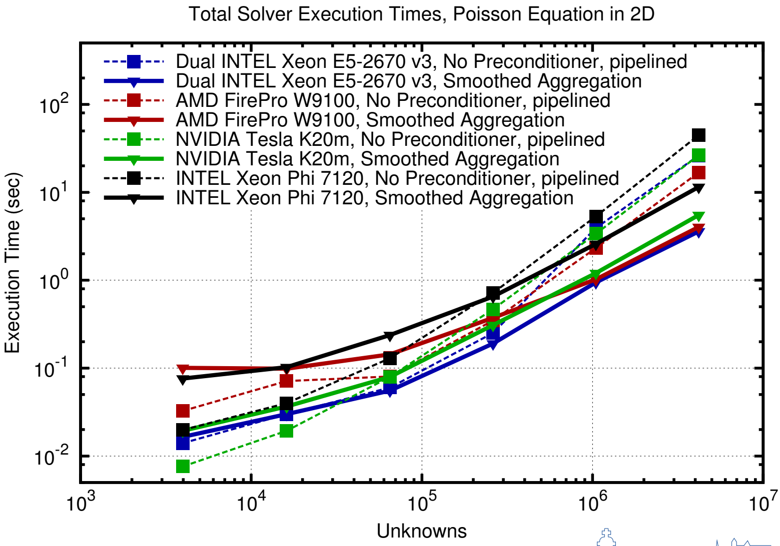
$C = A^{\text{fine}}P$

$A^{\text{coarse}} = RC$

AMG Sparse Matrix-Matrix Multiplication



AMG Benchmark



Summary

FEM Integration with Quadrature

- Thread transposition over cell patches
- Performance model with high accuracy
- Peak performance on NVIDIA GPUs

Fast Solvers

- Shift to parallel algorithms, sequentially inefficient
- ILU: Multiple sweeps for setup and solve
- AMG: Coarse grid computation on CPU?

How to Use and Reproduce?

- ViennaCL: `http://viennacl.sourceforge.net/`
- PETSc: `http://mcs.anl.gov/petsc/`