CUDA Implementation of Modern Preconditioning Techniques for Iterative Solvers of Linear Systems

Massimo Bernaschi

massimo.bernaschi@cnr.it

Giving Credit where Credit is due

Andrea Franceschini, Post-Doc, University of Padua (currently in Stanford)

Victor Magri, PhD Student, University of Padua (joining soon LLNL)

Mauro Carrozzo, CNR, Roma

Mauro Bisson, NVIDIA, U.S.

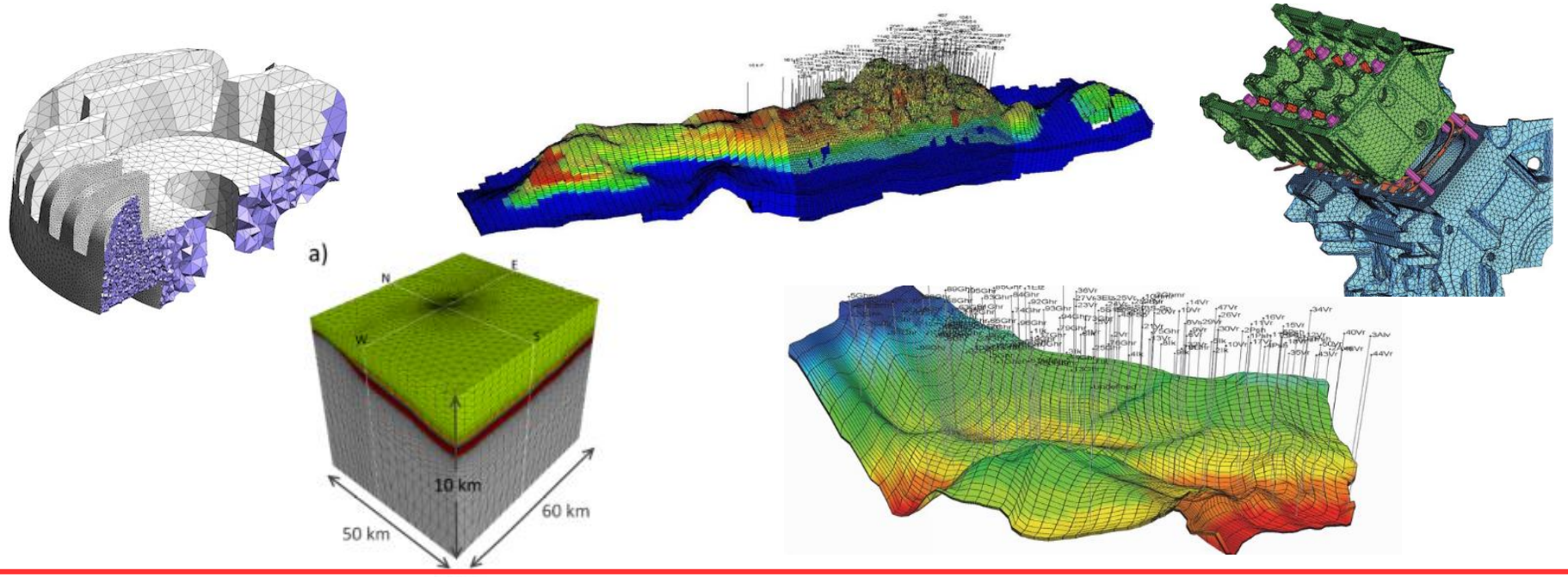
Dario Pasquini, PhD Student, University of Rome “Sapienza”

Carlo Janna, University of Padua

Pasqua D’Ambra, CNR, Naples

Motivation

The computational requirements of models in research and industrial applications are steadily increasing



The linear solver is often the most time consuming kernel in numerical simulation methods.

Introduction (1/2)

- Iterative Conjugate Gradient-like methods are very effective especially on HPC systems but…
- … the linear system MUST be preconditioned to achieve fast convergence!
- Preconditioning is *"the art of transforming a problem that appears intractable into another whose solution can be approximated rapidly"* [Trefethen and Bau, 1997]
- A preconditioned system is:

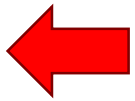
$$M_1^{-1} A M_2^{-1} M_2 \mathbf{x} = M_1^{-1} \mathbf{b}$$

where $M^{-1}=M_1^{-1}M_2^{-1}$ is the "preconditioner"

- Convergence is accelerated if M^{-1} resembles, somehow, A^{-1}
- At the same time, M^{-1} must be sparse, so as to keep the cost for the preconditioner computation, storage and application to a vector as low as possible
- No rules: even, apparently, naïve ideas can work surprisingly well!

Introduction (2/2)

- The development of algebraic preconditioners has experienced a big impulse in the last two decades
- Algebraic preconditioners: robust algorithms that generate a preconditioner from the knowledge of the system matrix only without taking into account the problem (geometry) it arises from
- Most popular and successful preconditioners:
  - Incomplete LU factorization
  - Approximate inverse techniques
  - Algebraic multigrid



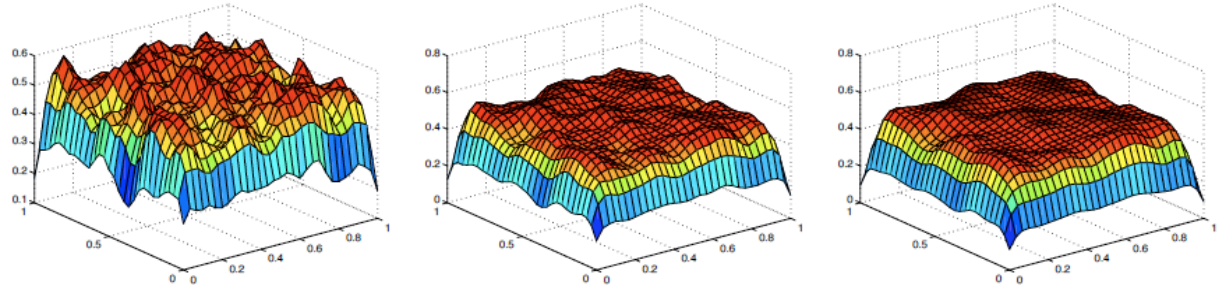
An approximate inverse can also be used as **smoother** in an **AMG**

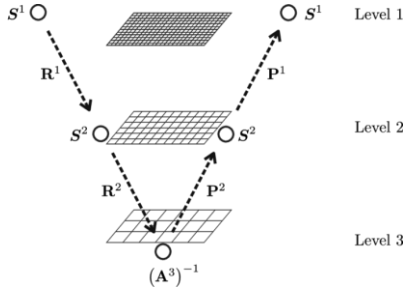
Algebraic Multigrid (AMG): Main Components

$$Ax = b \text{ where } A \in \mathbb{R}^{n \times n} \text{ s.p.d.}$$

Two-level method

1. solve $A\mathbf{x}_1 = \mathbf{b} \rightarrow$ smoother iterations
2. compute residual $\mathbf{r} = \mathbf{b} - A\mathbf{x}_1$
3. restrict residual to a coarse space $\mathbf{r}_c = P_c^T \mathbf{r}$
4. solve $A_c \mathbf{x}_c = \mathbf{r}_c \rightarrow$ coarse-grid correction **[solved in a recursive way]**
5. update solution $\mathbf{x} = \mathbf{x}_1 + P_c \mathbf{r}_c$
6. optional post-smoothing iterations (to symmetrize the algorithm)





Factorized Sparse Approximate Inverse (FSAI)

- Factorized Sparse Approximate Inverse (FSAI): an almost perfectly parallel factored preconditioner for SPD problems [Kolotilina & Yeremin, 1993] :

$$M^{-1} = F^T F$$

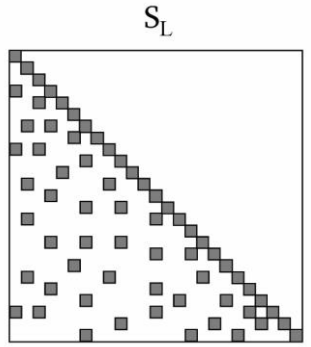
with F a lower triangular matrix such that:

$$\|I - FL\|_F \rightarrow \min$$

over the set of matrices with a prescribed lower triangular sparsity pattern S_L, e.g. the pattern of A or A^2, where L is the exact Cholesky factor of A

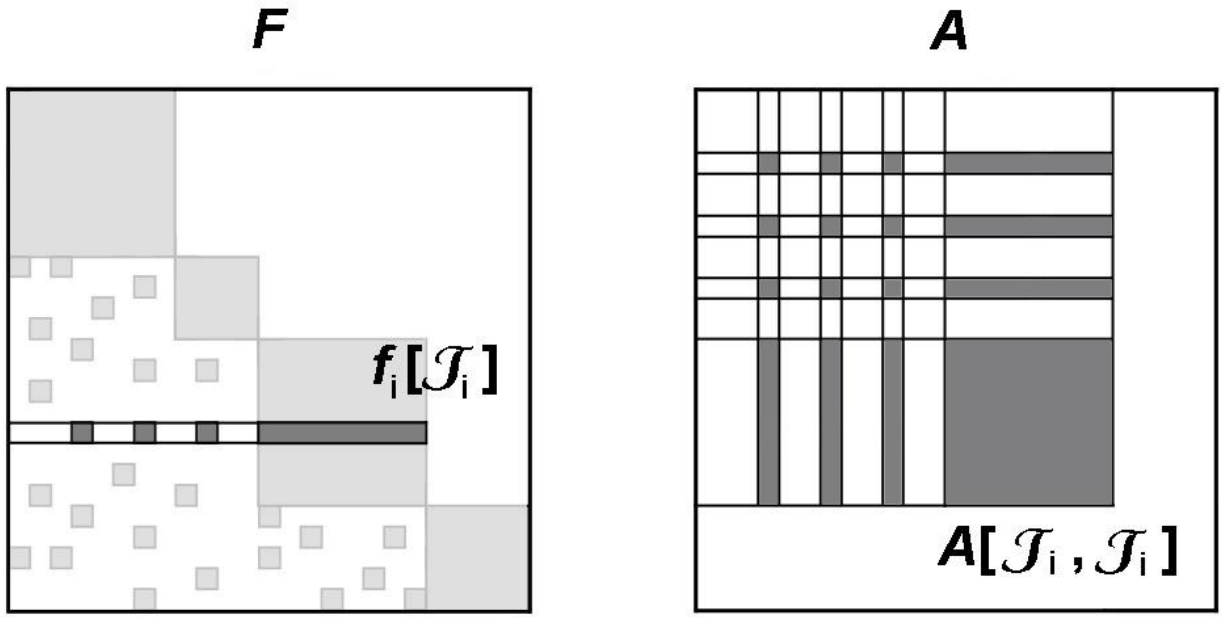
→ L is not actually required for computing F!

- Computed via the solution of n independent small dense systems and applied via matrix-vector products



FSAI construction

- The computation of each row of F is performed by:
  1. Gathering a small dense system whose entries are in A
  2. Solving that system with respect to a unitary rhs



FSAI features

1. The FSAI preconditioner is always SPD ⟹ we can use PCG

2. FSAI is applied through a matrix by vector product ⟹ almost perfectly parallel application

3. FSAI construction has a very high degree of natural parallelism

- In tough problems, the preconditioner computation may require large computational resources and the natural parallelism of FSAI construction is very attractive

FSAI variants

A key-point in the computation of FSAI preconditioner is the choice of the non-zero pattern. Two possible alternatives:

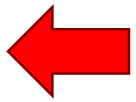
1. **Static pattern FSAI**
   - The pattern is chosen a priori
   - Easier implementation and cheaper computation

2. **Dynamic pattern FSAI**
   - The pattern is adaptively chosen during set-up
   - More effective but also more expensive and difficult to implement

C. Janna and M. Ferronato, "*Adaptive pattern research for block FSAI preconditioning*", SIAM J. Sci. Comp. 33, 3357-3380 (2011)

C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati, "FAIPACK: A Software Package for High-Performance Factored Sparse Approximate Inverse Preconditioning", ACM Trans. Math. Soft. 41, (2015)

Static Pattern FSAI

- In the recente past, we successfully ported on GPU the static pattern FSAI*
- Three main kernels:
  - Systems gather;
  - Dense system solver;
  - Sparse matrix by vector product;  we rely on **CUSPARSE**
- An overall **10x speed-up** has been obtained w.r.t. to a highly tuned OpenMP CPU implementation

*M.B., M. Bisson, C. Fantozzi and C. Janna, “A Factored Sparse Approximate Inverse preconditioned conjugate gradient solver on graphics processing unites”, SIAM J. Sci. Comp. 38, C53-C72 (2016)

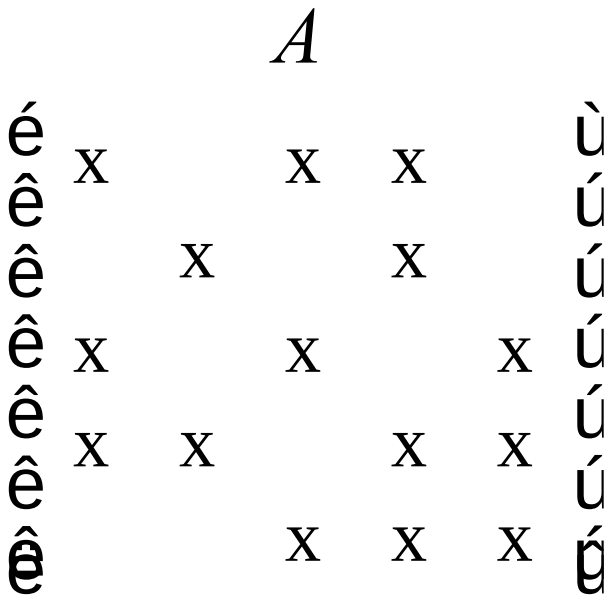
Gather of small systems on GPU (1/2)

- On the CPU each thread collects a whole system

Pattern of a row of **F**

1	35	78	111	291

1 thread



In this example **1 thread** enters 5 rows of ***A*** looking for 5 column indices on each of them

Linear search on each ***A*** row

- As the GPU are **SIMD** systems, this approach is very ineffective

Gather of small systems on GPU (2/2)

- To achieve a better performance we use an approach we* proposed for the breadth first search on large scale graphs

Pattern of a row of **F**

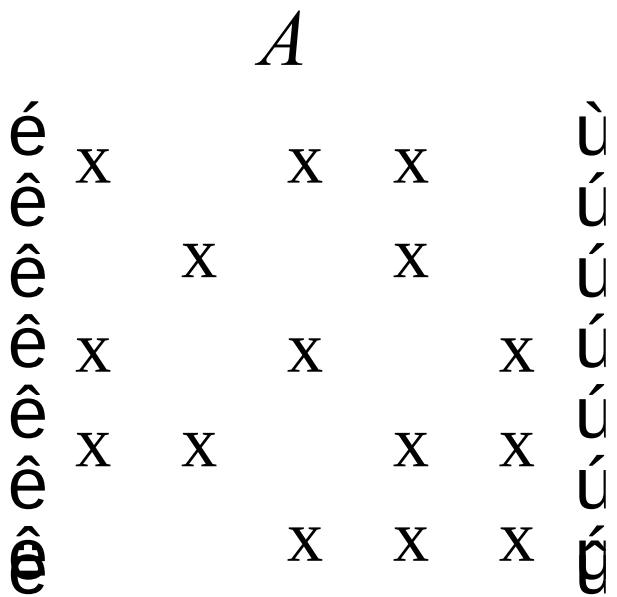
1	35	78	111	291

Expanded system to be be gathered

1	35	78	111	291
1	35	78	111	291
1	35	78	111	291
1	35	78	111	291
1	35	78	111	291

25 thread

Binary search on each ***A*** row



* M.B. and E. Mastrostefano, "*Efficient breadth first search on multi-GPU systems*", J. Par. Distr. Comp. 73, 1292-1305 (2013)

Cholesky decomposition of small systems on GPU

- The solution to long sequences of small dense linear systems is a kind of problem that, at first sight, does not fit to the GPU features
- Most numerical libraries aim at solving a small (1) number of large linear systems
- To optimize the GPU performance it is necessary to use **register**s and, as a consequence, to write a different specialized kernel for each system dimension
- We used an approach similar to that proposed by Anderson *et al.**
- We have a set of kernels to solve problem of sizes 32, 64, 96,…
- Systems of intermediate dimension are properly padded

* M. J. Anderson, D. Sheffield and K. Keutzer, “A predictive model for solving small linear algebra problems on GPU registers”, IEEE 26th International Parallel and Distributed Processing Symposium (2012)

*M.B., M. Bisson, C. Fantozzi and C. Janna, “A Factored Sparse Approximate Inverse preconditioned conjugate gradient solver on graphics processing unites”, SIAM J. Sci. Comp. 38, C53-C72 (2016)

Adaptive (a.k.a. Dynamic) FSAI (1/3)

Good news:

- Dynamic pattern FSAI uses all the kernels already developed for static pattern FSAI and its application to a vector is identical

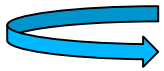
Bad news:

- Dynamic pattern FSAI set-up is based on the computation of the "*Kaporin Gradient*"
- the *"Kaporin Gradient"* is essentially a sparse matrix by sparse vector product
- This operation needs to be performed several times for each row during set-up and it is both time-consuming and memory *unfriendly*

M. B., M. Carrozzo, A. Franceschini and C. Janna, "A dynamic pattern Factored Sparse Approximate Inverse preconditioner on Graphics Processing Units", accepted for publication in the SIAM Journal on Scientific Computing.

Adaptive (a.k.a. Dynamic) FSAI (2/3)

Since GPUs are SIMD architectures, the computation of dynamic pattern of GPUs requires a flip in the loop for its computation

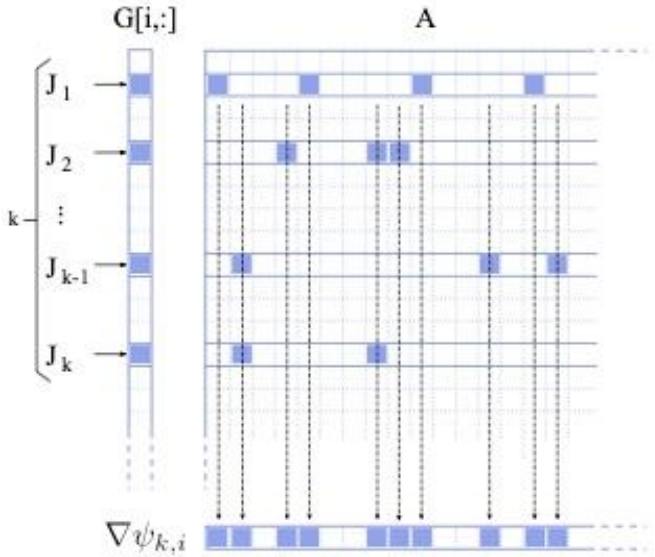


ALGORITHM 3: ADAPT_FSAI - Dynamic FSAI with adaptive pattern generation.

Input: Matrix A, Matrix G_0 (optional), Parameters k_{iter}, s, τ, ε;
Output: Matrix G;
if G_0 is given **then**
    $\widetilde{G}_0 \leftarrow \text{diag}(G_0)^{-1} G_0$;
else
    $\widetilde{G}_0 \leftarrow I$;
end
for $i = 1, n$ **do**
    for $k = 0, k_{iter} - 1$ **do**
        Compute $\nabla \psi_{k,i}$;
        Compute $\overline{\mathcal{P}}_i^{k+1}$ by adding to $\overline{\mathcal{P}}_i^{k}$ the indices of the s largest components of $\nabla \psi_{k,i}$;
        Gather $A[\overline{\mathcal{P}}_i^{k+1}, \overline{\mathcal{P}}_i^{k+1}]$ and $A[\overline{\mathcal{P}}_i^{k+1}, i]$ from A;
        Solve $A[\overline{\mathcal{P}}_i^{k+1}, \overline{\mathcal{P}}_i^{k+1}] \widetilde{\boldsymbol{g}}_{k+1,i} = -A[\overline{\mathcal{P}}_i^{k+1}, i]$;
        if $\psi_{k+1,i} \leq \varepsilon \cdot \psi_{0,i}$ **then** exit the loop over k;
        Drop the entries of $\widetilde{\boldsymbol{g}}_{k+1,i}$ such that $\widetilde{g}_{k+1,ij} \leq \tau \|\widetilde{\boldsymbol{g}}_{k+1,i}\|_2$;
    end
    $\widetilde{d}_{ii} \leftarrow (-\widetilde{\boldsymbol{g}}_{k+1,i}^T A[\overline{\mathcal{P}}_i^{k+1}, i])^{-\frac{1}{2}}$;
    $\widetilde{\boldsymbol{g}}_{k+1,i} \leftarrow \widetilde{d}_{ii} \widetilde{\boldsymbol{g}}_{k+1,i}$;
    Scatter the components of $\widetilde{\boldsymbol{g}}_{k+1,i}$ into $G_{k+1}[i, \mathcal{P}_i^{k+1}]$
end

Adaptive (a.k.a. Dynamic) FSAI (3/3)

- The computation of the Kaporin gradient involves a sparse matrix by sparse vector product that currently represents the bottleneck of the procedure



- A problem similar to merging a set of (scaled) rows of the matrix in a vector
- The output pattern is determined dynamically

Hash based computation of Kaporin gradient

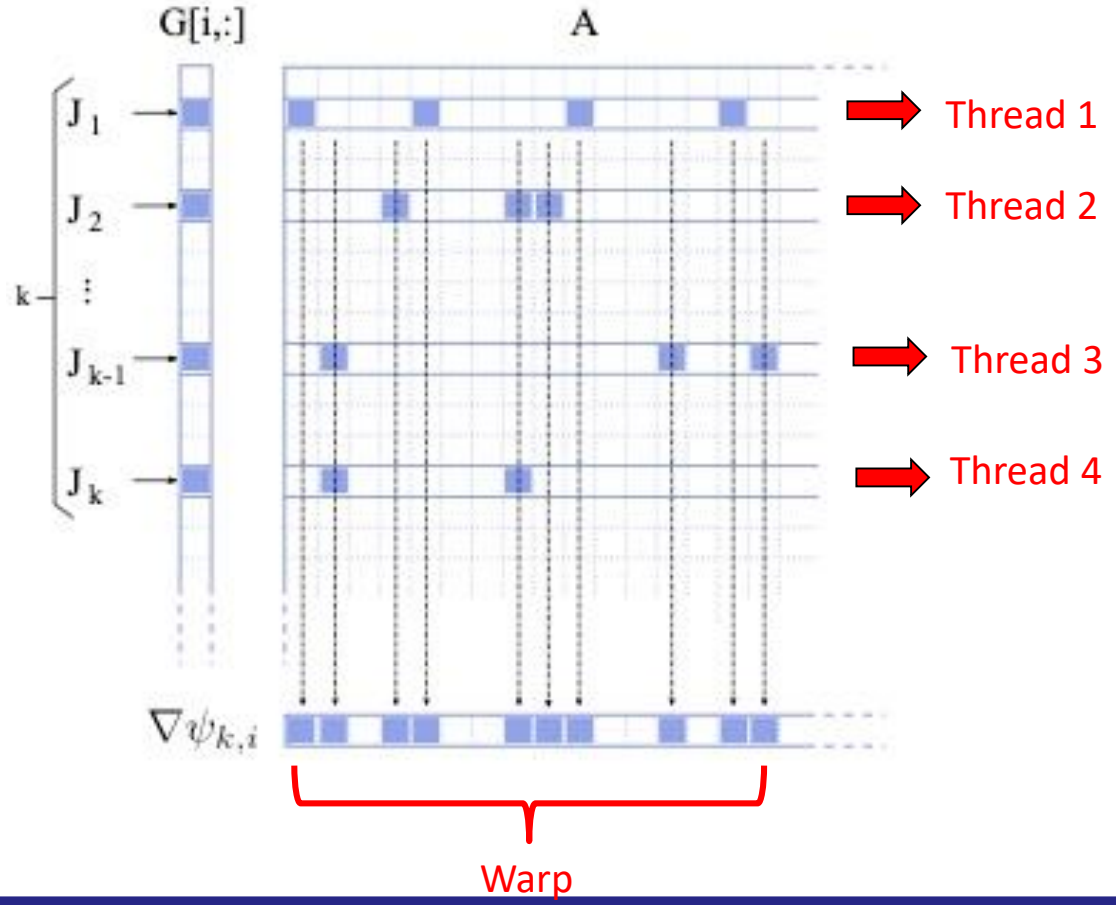
- Three vectors to manage the Kaporin Gradient ***g***:
  - IW: containing the unsorted column indices of ***g***
  - WR: containing the unsorted values of ***g***
  - JWN: a non-zero indicator containing the position of non-zeroes of ***g*** in IW and WR

IW	4	8	3	7	…					
WR	1.2	10.3	4.5	8.9	…					
JWN	0	0	3	1	0	0	4	2	0	0

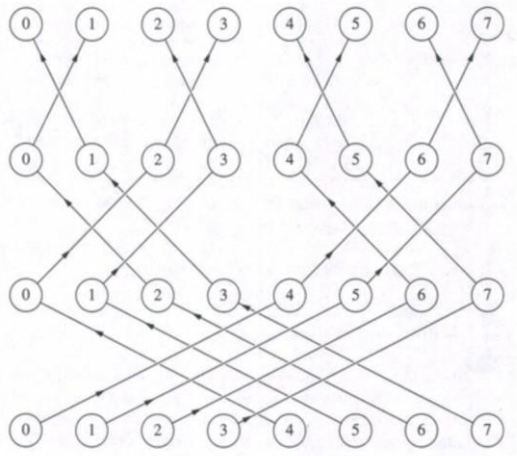
- This method is very efficient on CPU, but not suitable to GPU.
- The main problem is the size of JWN (of the order of the matrix)

Warp centric computation of Kaporin gradient

- The gradient ***g*** is managed by an entire warp
- Each thread in the warp explores one row of *A*



For the choice of the element having the max absolute value we resort to a butterfly data exchange pattern



Experimental results (1/3)

CPU: 2 Intel(R) Xeon® E5-1620 @ 3.50GHz (4 cores each)
GPU:

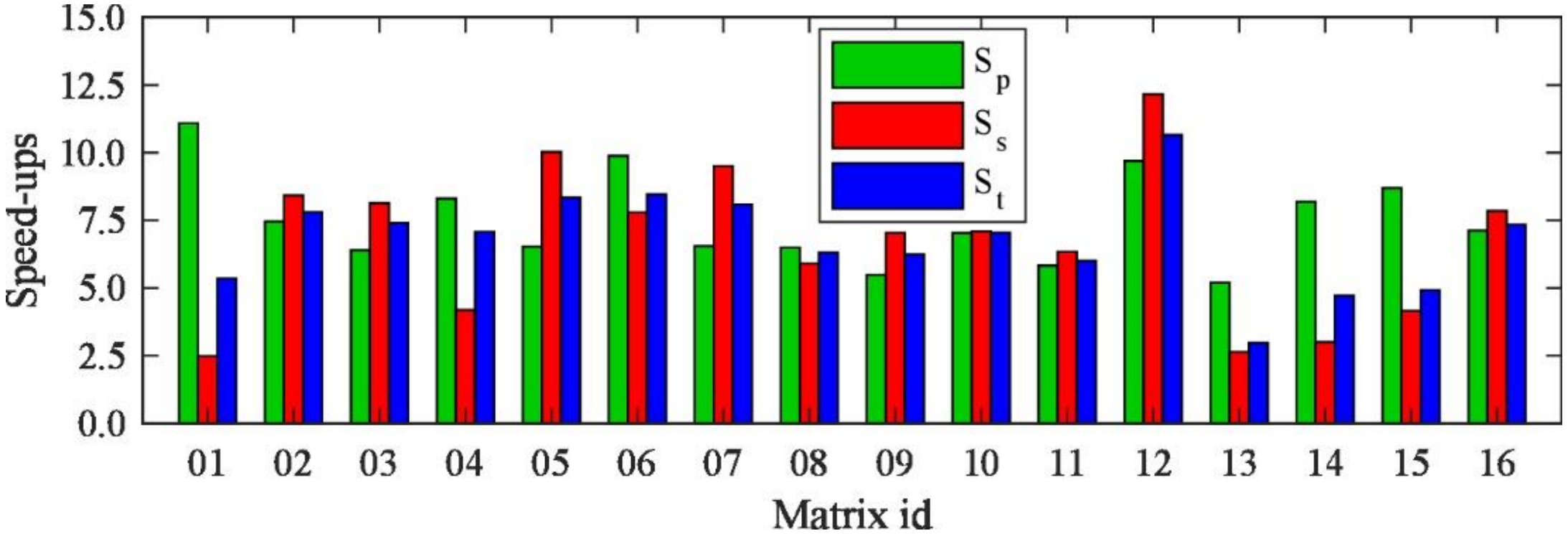
- Pascal P100, with 16 Gb Ram, 3584 cores in 56 SM
- Volta Titan V. with 12 Gb Ram. 5120 cores in 80 SM

Test matrices:

id	matrix	nrows	non-zeros	nnz_{avg}	Application Field
01	Res_small*	137,140	6,234,348	45.46	P1 geomechanical model
02	Cube*	190,581	7,531,389	39.52	Q1 Structural Problem
03	pwtk	217,918	11,634,424	53.39	Pressurized Wind Tunnel
04	hood	220,542	10,768,436	48.83	Structural Problem
05	BenElechi1	245,874	13,150,496	53.48	2D/3D Problem
06	Mexico*	297,945	12,294,999	41.27	P1 subsurface structural model
07	msdoor	415,863	19,173,163	46.10	Structural Problem
08	Fault_639	638,802	27,245,944	42.65	Contact Mechanics
09	Pflow_742	742,793	37,138,461	50.00	3D Pressure in Reservoir
10	Res_big*	910,122	40,019,202	43.97	P1 geomechanical model
11	Emilia_923	923,136	41,005,206	44.42	3D geomechanics
12	thermal2	1,228,045	8,580,313	6.99	Thermal Problem
13	StocF-1465	1,465,137	21,005,389	14.34	CFD
14	G3_circuit	1,585,478	7,660,826	4.83	Circuit Simulation
15	pi8grid8*	2,689,537	18,816,513	7.00	2D anisotropic Laplacian
16	Bump_2911	2,911,419	127,729,899	43.87	3D geomechanics

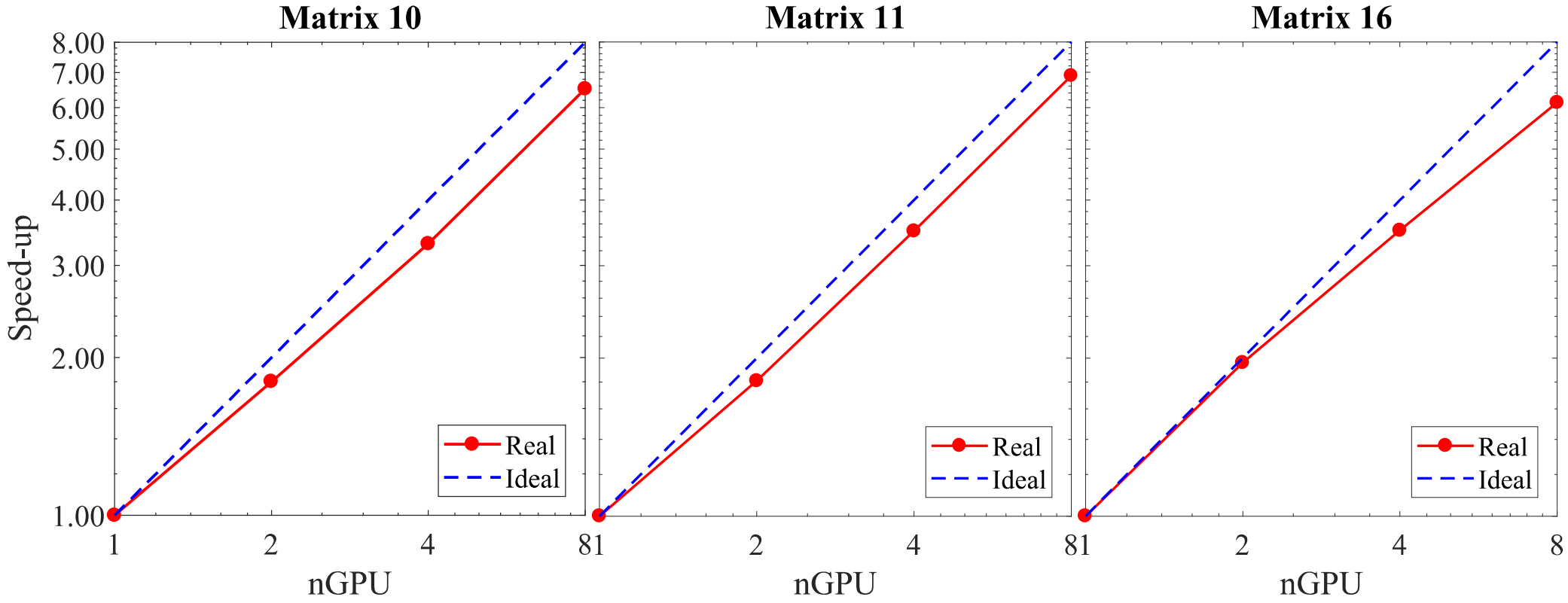
Experimental results (2/3)

GPU (Titan V) speed up w.r.t. OpenMP CPU (8 cores)

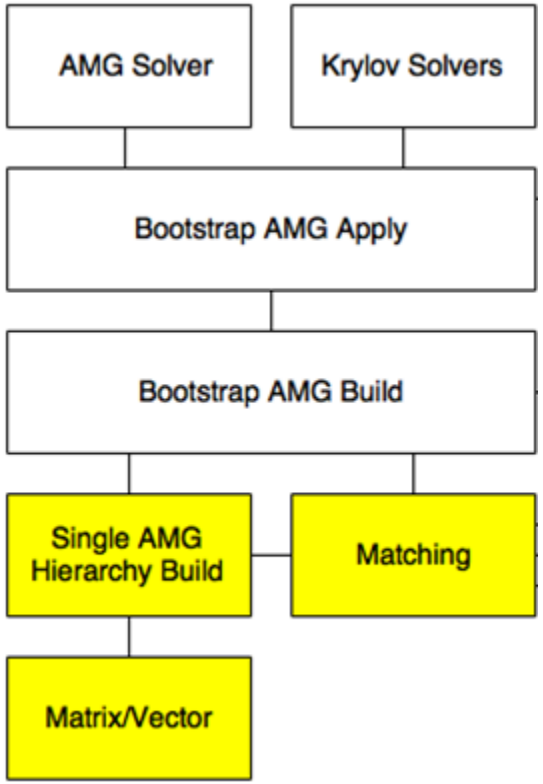


- *Maximum Speed-up* ~ 12.5
- *Minimum Speed-up ~ 2.5*
- *Average Speed-up ~ 7*

Experimental results (3/3)



BootCMatch: AMG based on compatible weighted matching



AMG setup:

- **Aggregation based AMG**
- Pairwise aggregation based on **maximum weight matching**

AMG application:

- **Krylov Solvers** (i.e., PFCG)
- **V, H, W, K cycles** (both as preconditioners and standalone solvers)
- Bootstrap to build composite AMG with desired convergence rate

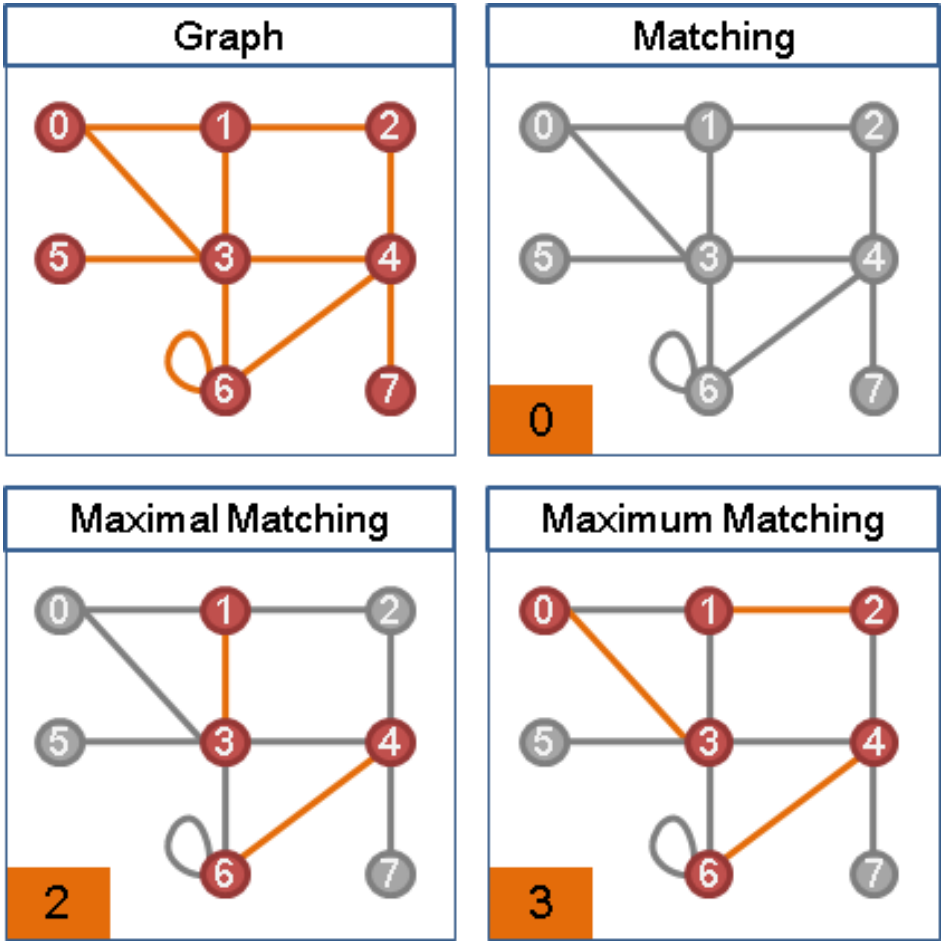
CPU version github.com/bootcmatch/BootCMatch

[P. D'Ambra *et al.*, BootCMatch: a Software package for Bootstrap AMG based on Graph Weighted Matching, ACM Transactions on Mathematical Software, Vol.44 2018.]

Pairwise Aggregation

- **Given** the input matrix A, it is necessary to build $\widehat{\boldsymbol{A}}$, **a matrix of weights that characterize the strength of the connection (*i.e.,* the similarity) between pairs of variables**
- $\widehat{\boldsymbol{A}}$ has the same dimension and sparsity pattern of $\boldsymbol{A}$ but with **null diagonal**;
  [P. D'Ambra, et al., Adaptive AMG with Coarsening based on Compatible Weighted Matching]
  - The procedure is applicable to general matrices and does not require thresholds or other user defined parameters.
- Next step is to select the pairs of variables having the highest connection each other
  - Each variable *i* will be connected to **a single** variable *j* or it will be a *singleton*
- Finding the set of edges that couples the variables according to the strength of their connection is an instance of the classic *matching* problem in graph theory.

(Maximum Weighted) Graph Matching (1/2)



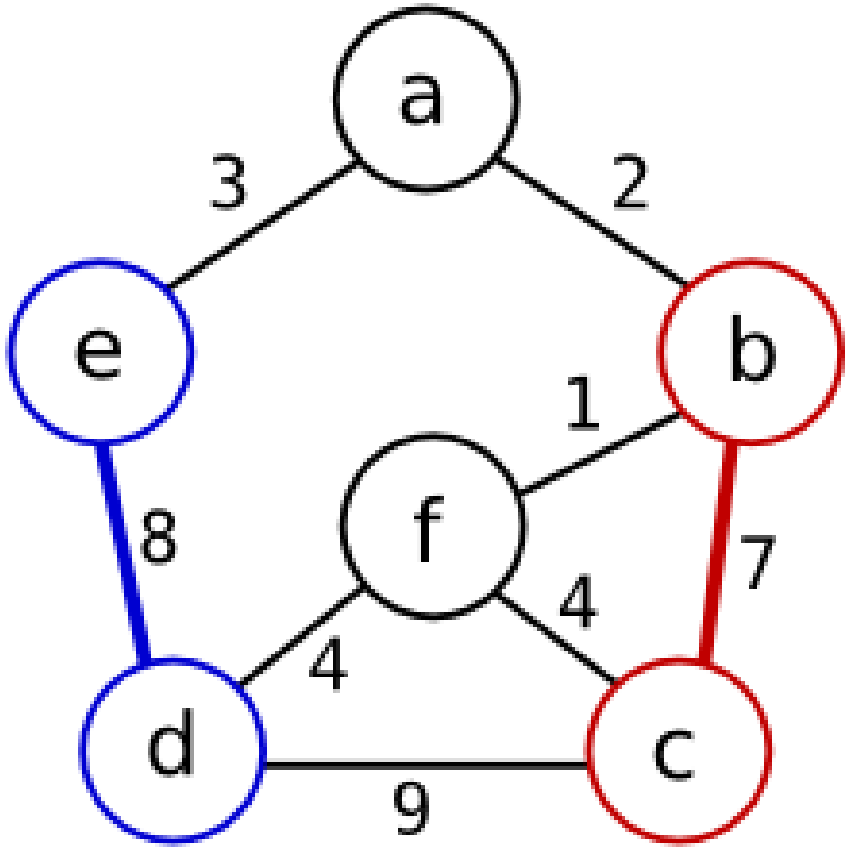
Given a undirect graph $G = \{V, E\}$ described by a weighted adjacency matrix

- $M \subseteq E$ is a **matching** for G iff it includes only **non-adjacent edges**
- M is a **perfect matching** if its edges **touch all the vertices** V
- M is a **maximum weighted matching** iff the **sum** of the weights of its edges is maximized:

$$C(M) = \sum_{(i,j)\in M} c_{ij} \qquad \max_{M' matching\ in\ G} C(M')$$

where c_{ij} is the weight of edge i, j

(Maximum Weighted) Graph Matching (2/2)



- A matching of maximum weight 15 can be found by pairing vertex b to vertex c and vertex d to vertex e
  (leaving vertices a and f unpaired).
- A *perfect* matching (including the pair **a-b**, **f-c** and **d-e**) would have weight equal to 14

Matching Algorithms

MC64:

- Optimal Exact Matching
- $\mathcal{O}(n(n+nnz)\log(n))$

Preis algorithm:

- half-approximate
- $\mathcal{O}(nnz)$

Auction algorithm:

- Near optimal
- $\mathcal{O}(n \cdot nnz\ log(n))$

Matching algorithms for CPU are not very suitable for GPU

(nnz is the number of non-zero elements in the matrix of size n)

Matching Algorithm: Suitor (1/2)

Matching algorithms for CPU are not very suitable for GPU:

MC64:

- Optimal Matching
- $\mathcal{O}(n(n+nnz)\log(n))$

Preis algorithm:

- half-approximate
- $\mathcal{O}(nnz)$

Auction algorithm:

- Near optimal
- $\mathcal{O}(n \cdot nnz\ log(n))$

We resorted to a relatively new matching algorithm:

Suitor Algorithm

- **half-approximate algorithm**
- Time complexity: $O(n\ \Delta)$
- based on the dominant edge strategy
- Very easily to parallelize

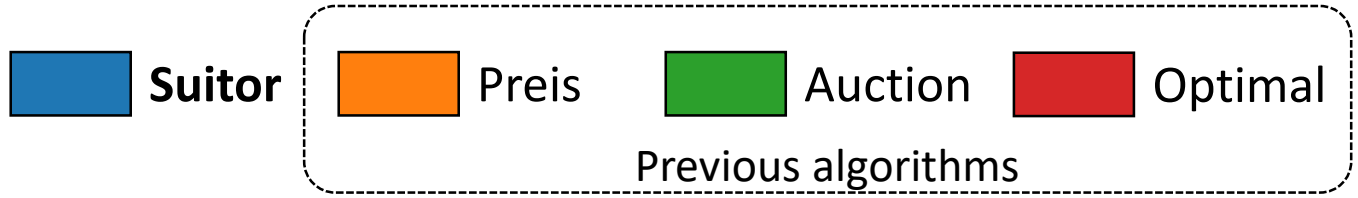
[M. Halappanavar, et al, "Approximate weighted matching on emerging manycore and multithreaded architectures“]
[Md. Naim et al “Optimizing Approximate Weighted Matching on Nvidia Kepler K40”]

Matching Algorithm: Suitor (2/2)

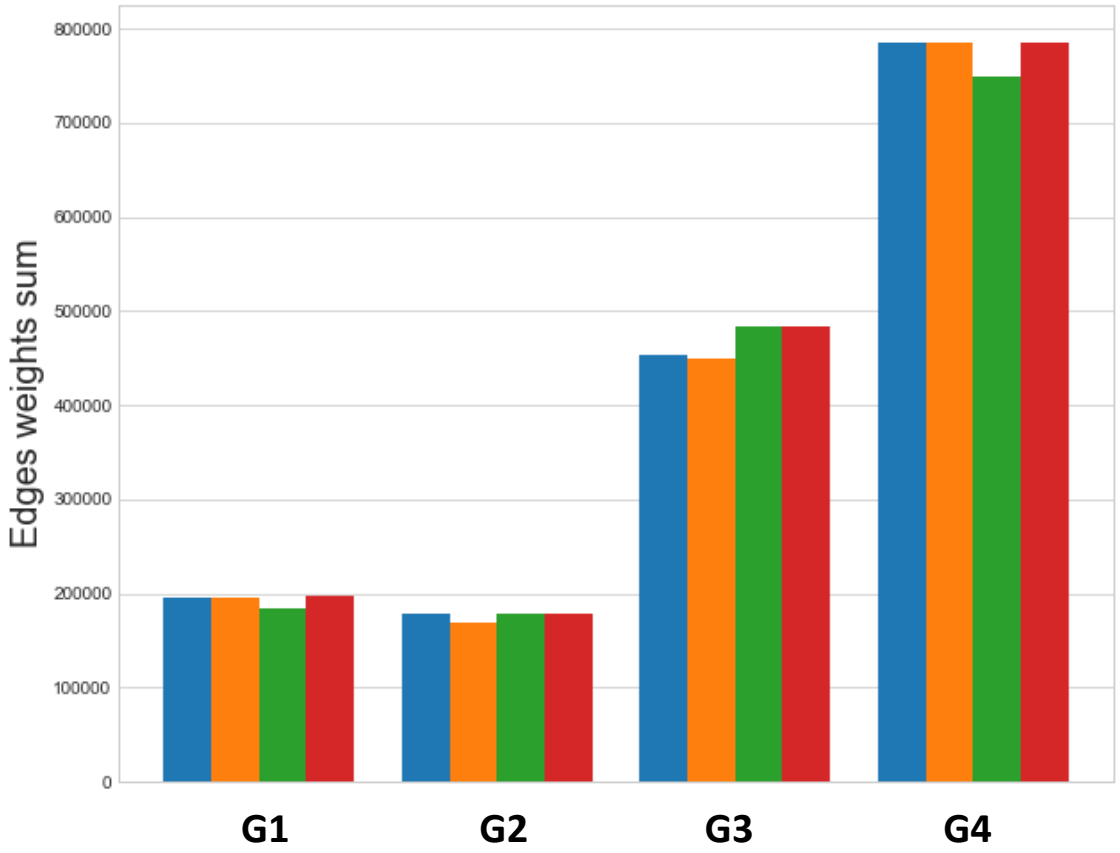
- A vertex *u* proposes (tentatively matches) to its heaviest neighbor *v* that does not already have a proposal of heavier weight.
  - This reduces the number of neighbors a vertex considers as candidate mates.
- If *v* has a proposal of lower weight, then *u* matches itself to v and unmatches *v*'s mate *w*
- The algorithm now has to find a new mate for *w*

[M. Halappanavar, et al, "Approximate weighted matching on emerging manycore and multithreaded architectures"]
[Md. Naim et al "Optimizing Approximate Weighted Matching on Nvidia Kepler K40"]

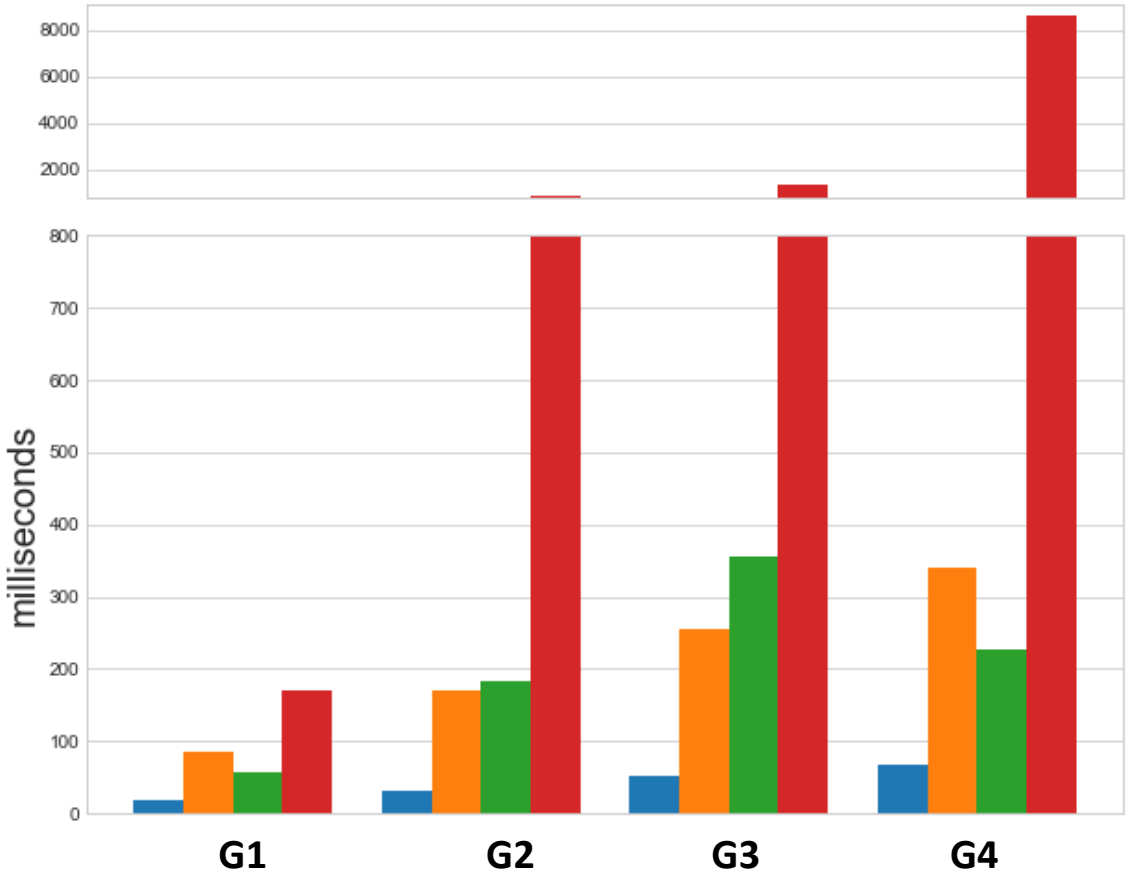
Suitor performance



Solution Quality



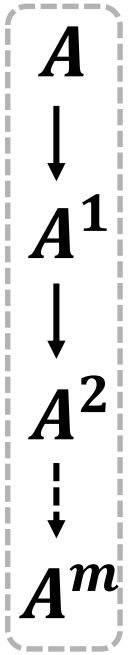
Execution Time



Coarsening Algorithm

- $\boldsymbol{A^0 = A},\ k = 0, w^0 = w$
- $\boldsymbol{while}\ size(\boldsymbol{A^k}) > maxsize$:
  - $\boldsymbol{P^k} = pairwiseAggregation(\boldsymbol{A}, \boldsymbol{w^k})$
  - $\boldsymbol{R^k} = (\boldsymbol{P^k})^T$
  - $\boldsymbol{A^{k+1}} = \boldsymbol{R^k A^k P^k}$ *(Galerkin product)*
  - $\boldsymbol{w^{k+1}} = \boldsymbol{R^k w^k}$
  - $k++$

Hierarchy:



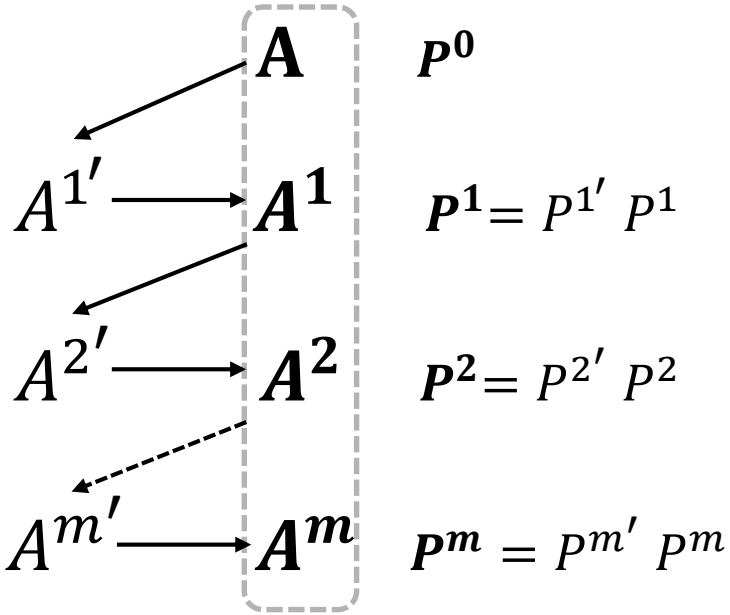
- Where $\boldsymbol{P^k}$ **(prolongator)** is a constant piecewise operator obtained from the aggregates produced by the pairwise aggregation

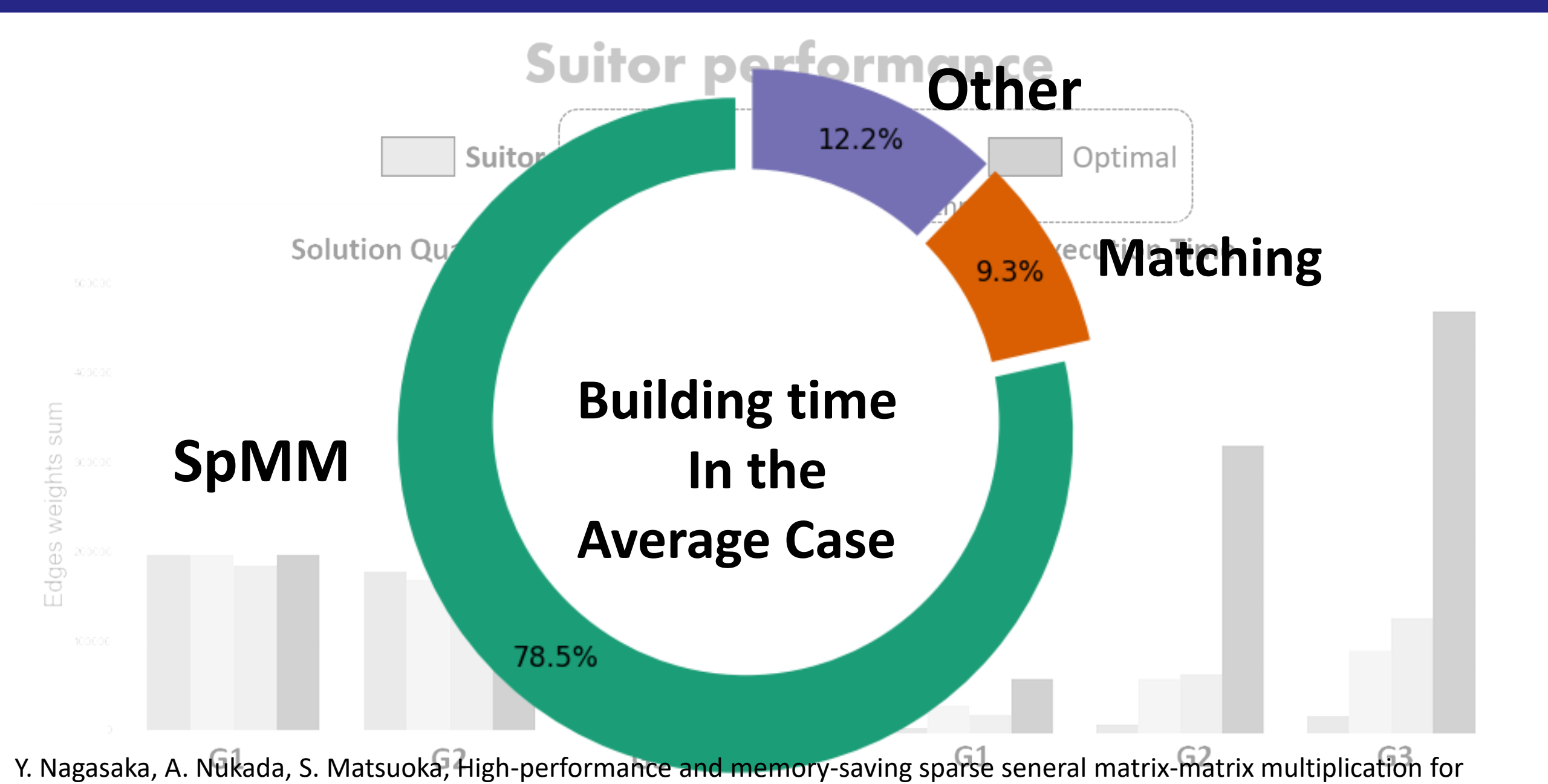
Aggressive Coarsening Algorithm

- $\boldsymbol{A^0 = A},\ k = 0, w^0 = w$
- $\boldsymbol{while}\ size(\boldsymbol{A^k}) > maxsize$:
  - $\boldsymbol{P^k} = pairwiseAggregation(\boldsymbol{A}, \boldsymbol{w^k})$
  - $\boldsymbol{R^k} = (\boldsymbol{P^k})^T$
  - $\boldsymbol{A^{k+1}} = \boldsymbol{R^k A^k P^k}$
  - $\boldsymbol{w^{k+1}} = \boldsymbol{R^k w^k}$
  - $k++$

Can be repeated n times for each level k

e.g. **Hierarchy** obtained by a **double pairwise aggregation :**

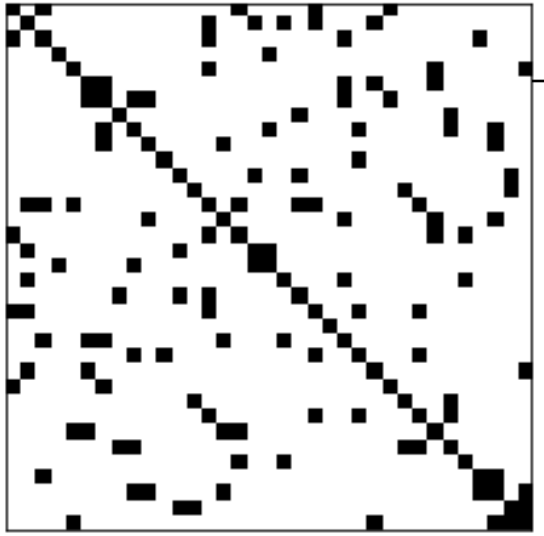




Y. Nagasaka, A. Nukada, S. Matsuoka, High-performance and memory-saving sparse seneral matrix-matrix multiplication for Nvidia Pascal GPU, in: IEEE 46th International Conference on Parallel Processing, IEEE, 2017

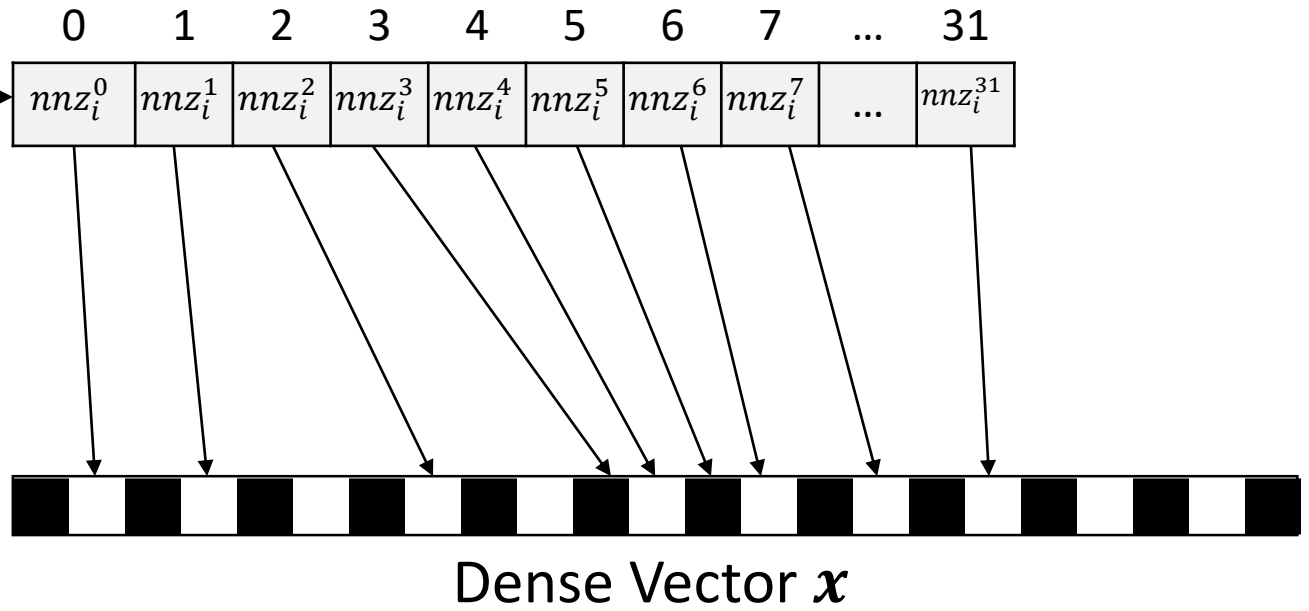
Sparse Matrix Dense Vector product [SpMxV] General case

Sparse Matrix $\boldsymbol{A}$:



CSR row

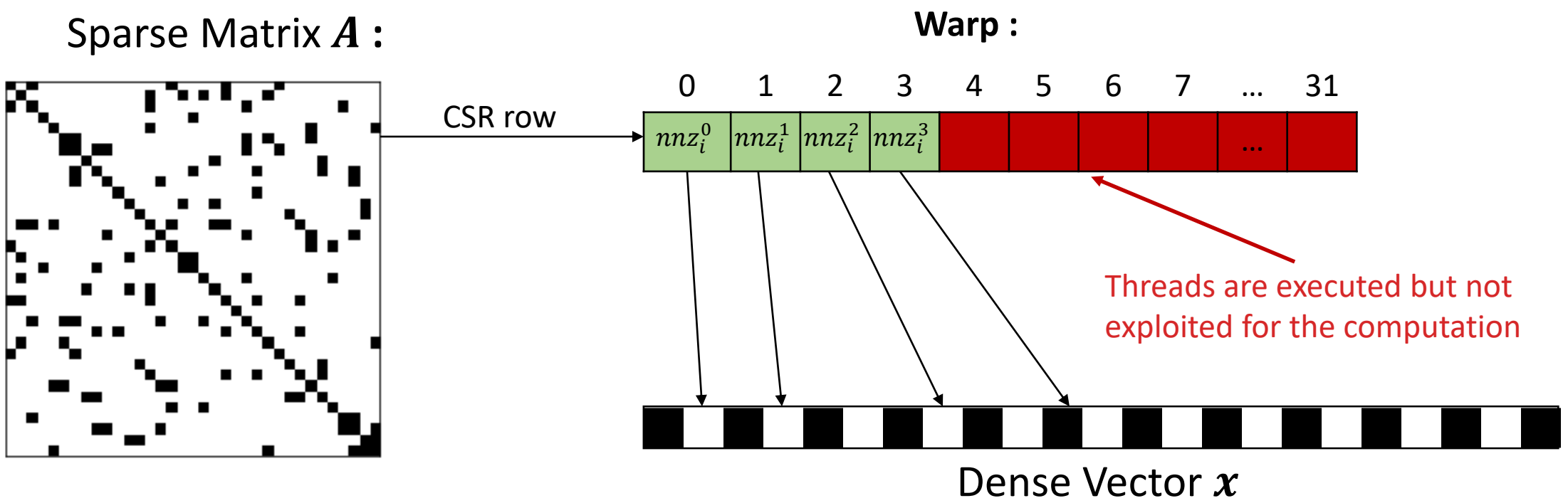
Warp :



1. Each **warp** is in charge of a row *i* of the matrix A
2. Each thread in the **warp** gets a nnz element $\mathrm{A}[i,j]$ and multiplies it by $x[j]$
3. Each warp performs an internal sum reduction

$$y[i] = \sum_{j=0}^{n} A[i,j]x[j]$$

SpMxV on very sparse matrices



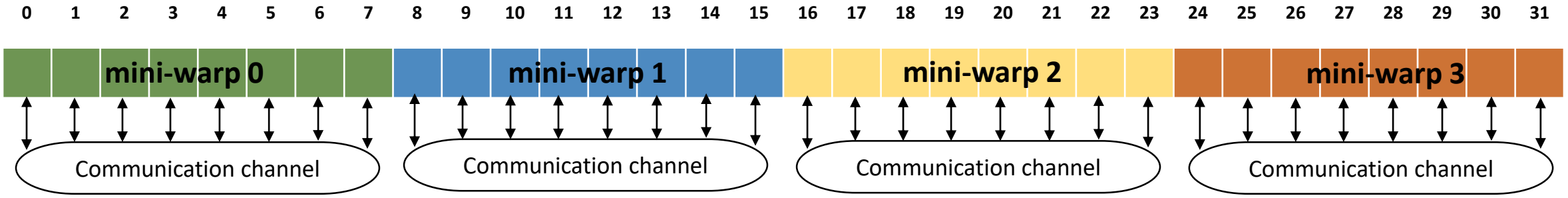
1. Each **warp** is in charge of a row *i* of the matrix A
2. Each thread in the **warp** gets a nnz element $\mathrm{A}[i,j]$ and multiplies it by $x[j]$
3. Each warp performs an internal sum reduction

$$y[i] = \sum_{j=0}^{n} A[i,j]x[j]$$

Improving Load Balancing via "mini-warp"

Full Warp:

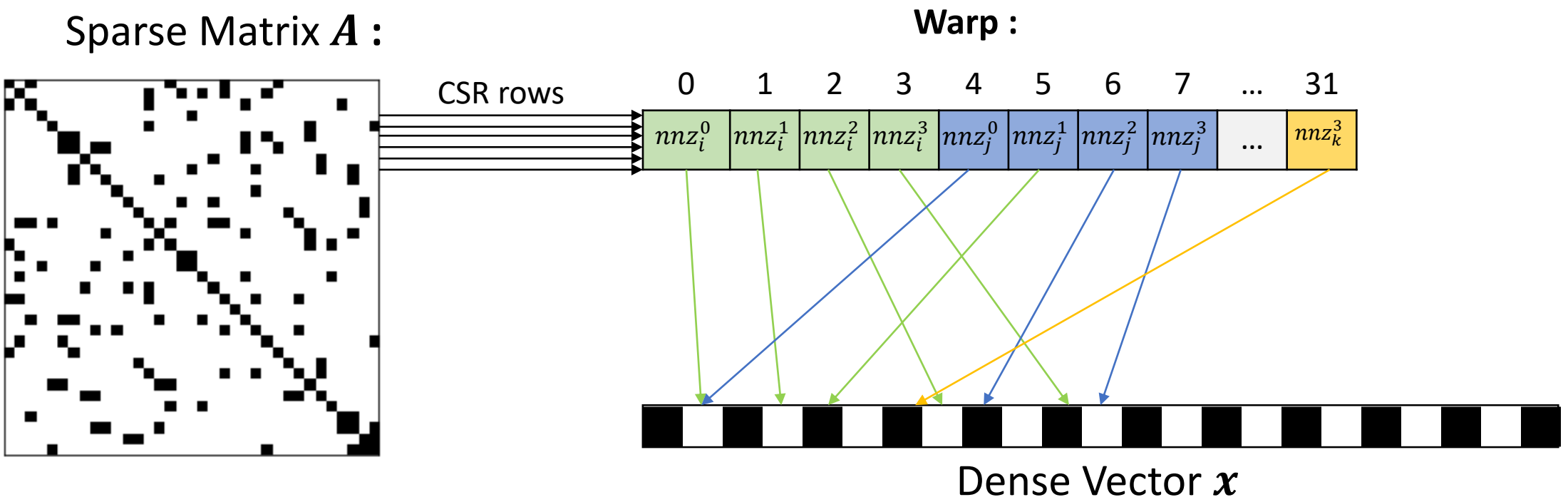
(e.g. 4 mini-warps of **8** threads each**)**



Mini-warps possible configuration:

- **16** mini-warps of **2** threads
- **8** mini-warps of **4** threads
- **4** mini-warps of **8** threads
- **2** mini-warps of **16** threads

SpMxV with "mini-warp"

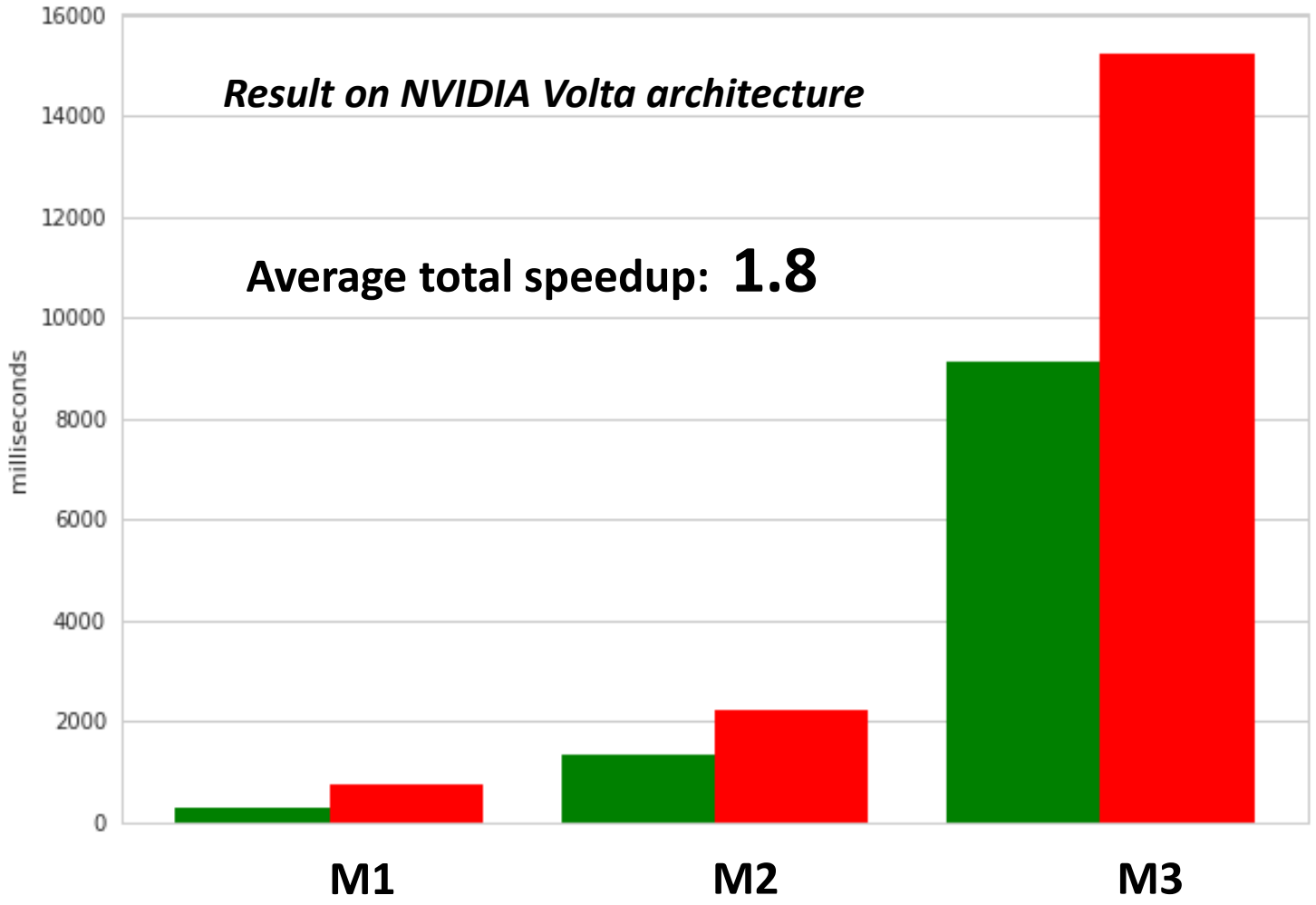


- Each **warp is divided in mini-warps of equal size.**
  Possible setups per warp:
- **16** mini-warps of **2** threads
- **8** mini-warps of **4** threads
- **4** mini-warps of **8** threads
- **2** mini-warps of **16** threads

Each mini-warp takes a different row and manages it as if it were a full warp

Mini-warp Product *vs.* cuSPARSE Product

Solving time [V-cycle + FPCG]:



cuSPARSE

Adaptive mini-warp

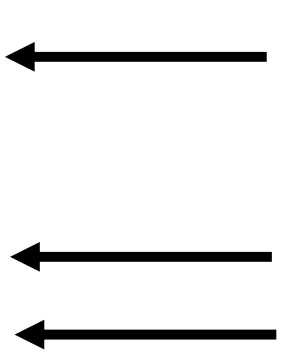
Three systems arising from Linear Elasticity Problems:

Matrix	n	nnz
M1	22×10^4	17×10^5
M2	11×10^5	68×10^5
M3	42×10^5	51×10^6

For each function call:
The mini-warp size is chosen **adaptively** by **averaging the number of *nnz*** per row of the input matrix

Flexible Conjugate Gradient (FCG) Optimizations [1]

1: Given u_0 and set $r_0 = b - Au_0$
2: $w_0 = d_0 = \mathcal{B}(r_0)$
3: $q_0 = Aw_0$
4: $\alpha_0 = w_0^T r_0$
5: $\beta_0 = w_0^T q_0$
6:
7: $u_1 = u_0 + \alpha_0/\beta_0 d_0$
8: $r_1 = r_0 - \alpha_0/\beta_0 q_0$
9:
10: **for** $i = 1, \ldots$ **do**
11: $w_i = \mathcal{B}(r_i)$
12:
13: $\gamma_i = w_i^T q_{i-1}$
14: $d_i = w_i - \gamma_i/\beta_{i-1} d_{i-1}$
15: $q_i = Ad_i$
16:
17: $\alpha_i = d_i^T r_i$
18: $\beta_i = d_i^T q_i$
19:
20: $u_{i+1} = u_i + \alpha_i/\beta_i d_i$
21: $r_{i+1} = r_i - \alpha_i/\beta_i q_i$
22:
23: **end for**



- It's possible to **rearrange** FCG algorithm in order to obtain these **three scalar products in sequence**

[Y. Notay, A. Napov, A massively parallel solver for discrete Poisson-like problems]

Scalar product between two dense vectors:

$$\sum_{i=0}^{n} x[i]y[i]$$

Flexible Conjugate Gradient (FCG) Optimizations [2]

1: Given u_0 and set $r_0 = b - Au_0$
2: $w_0 = d_0 = \mathcal{B}(r_0)$
3: $v_0 = q_0 = Aw_0$
4: $\alpha_0 = w_0^T r_0$
5: $\beta_0 = \rho_0 = w_0^T v_0$
6:
7: $u_1 = u_0 + \alpha_0/\rho_0 d_0$
8: $r_1 = r_0 - \alpha_0/\rho_0 q_0$
9:
10: **for** $i = 1, \ldots$ **do**
11: $w_i = \mathcal{B}(r_i)$
12: $v_i = Aw_i$
13:
14: $\alpha_i = w_i^T r_i$
15: $\beta_i = w_i^T v_i$
16: $\gamma_i = w_i^T q_{i-1}$
17: $\rho_i = \beta_i - \gamma_i^2/\rho_{i-1}$
18:
19: $d_i = w_i - \gamma_i/\rho_{i-1} d_{i-1}$
20: $u_{i+1} = u_i + \alpha_i/\rho_i d_i$
21:
22: $q_i = v_i - \gamma_i/\rho_{i-1} q_{i-1}$
23: $r_{i+1} = r_i - \alpha_i/\rho_i q_i$
24:
25: **end for**

- Originally designed for distributed implementation in order to reduce nodes' communication during the sum reduction
- It's possible to exploit this optimization on the GPU

Flexible Conjugate Gradient (FCG) Optimizations [2]

1: Given u_0 and set $r_0 = b - Au_0$
2: $w_0 = d_0 = \mathcal{B}(r_0)$
3: $v_0 = q_0 = Aw_0$
4: $\alpha_0 = w_0^T r_0$
5: $\beta_0 = \rho_0 = w_0^T v_0$
6:
7: $u_1 = u_0 + \alpha_0/\rho_0 d_0$
8: $r_1 = r_0 - \alpha_0/\rho_0 q_0$
9:
10: **for** $i = 1, \ldots$ **do**
11: $w_i = \mathcal{B}(r_i)$
12: $v_i = Aw_i$
13:
14: $\alpha_i = w_i^T r_i$
15: $\beta_i = w_i^T v_i$
16: $\gamma_i = w_i^T q_{i-1}$
17: $\rho_i = \beta_i - \gamma_i^2/\rho_{i-1}$
18:
19: $d_i = w_i - \gamma_i/\rho_{i-1} d_{i-1}$
20: $u_{i+1} = u_i + \alpha_i/\rho_i d_i$
21:
22: $q_i = v_i - \gamma_i/\rho_{i-1} q_{i-1}$
23: $r_{i+1} = r_i - \alpha_i/\rho_i q_i$
24:
25: **end for**

- Originally designed for distributed implementation in order to reduce nodes' communication during the sum reduction
- It's possible to exploit this optimization on the GPU

Performing the triple scalar product in a single kernel means:

1. Saving $^1/_3$ of global memory data transfer (dense vector w is read once)
2. Avoiding the overhead of two device synchronizations

GPU Kernel

Flexible Conjugate Gradient (FCG) Optimizations [2]

1: Given u_0 and set $r_0 = b - Au_0$
2: $w_0 = d_0 = \mathcal{B}(r_0)$
3: $v_0 = q_0 = Aw_0$
4: $\alpha_0 = w_0^T r_0$
5: $\beta_0 = \rho_0 = w_0^T v_0$
6:
7: $u_1 = u_0 + \alpha_0/\rho_0 d_0$
8: $r_1 = r_0 - \alpha_0/\rho_0 q_0$
9:
10: **for** $i = 1, \ldots$ **do**
11: $w_i = \mathcal{B}(r_i)$
12: $v_i = Aw_i$
13:
14: $\alpha_i = w_i^T r_i$
15: $\beta_i = w_i^T v_i$
16: $\gamma_i = w_i^T q_{i-1}$
17: $\rho_i = \beta_i - \gamma_i^2/\rho_{i-1}$
18:
19: $d_i = w_i - \gamma_i/\rho_{i-1} d_{i-1}$
20: $u_{i+1} = u_i + \alpha_i/\rho_i d_i$
21:
22: $q_i = v_i - \gamma_i/\rho_{i-1} q_{i-1}$
23: $r_{i+1} = r_i - \alpha_i/\rho_i q_i$
24:
25: **end for**

- Originally designed for distributed implementation in order to reduce nodes' communication during the sum reduction
- It's possible to exploit this optimization on the GPU

Performing the triple scalar product in a single kernel means:

1. Saving $^1/_3$ of global memory data transfer (dense vector w is read once)
2. Avoiding the overhead of two device synchronizations

One additional vector update operation (or axpy)

GPU Kernel

NVIDIA AmgX

- **State-of-the-art AMG library for GPUs**
- Flexible configuration allows for nested solvers, smoothers and preconditioners
- Support distributed multi-GPUs computation

- **We chose AmgX as benchmark**

[M. NAUMOV et al AMGX: A LIBRARY FOR GPU ACCELERATED ALGEBRAIC MULTIGRID AND PRECONDITIONED ITERATIVE METHODS]

BootCMatch Experiment Setting

AMG based on compatible weighted matching **as preconditioner of the FCG**

- **Coarsening:**
  - maximum weighted matching obtained by **Suitor algorithm**
  - **combination of 2 sweeps of pairwise (unsmoothed) aggregation**, corresponding to a coarsening ratio of at most 4
  - vector of all ones as initial smooth vector
- **Cycle:**
  - V-Cycle
  - **1** sweep of **Jacobi L1 smoother** as pre/post smoothing
  - **20** sweeps of **Jacobi L1 smoother** as coarsest solver
- **Solver:**
  - **stopping criterion**: relative residual $\leq 10^{-6}$
- **Hardware:**
  - NVIDIA **TITAN V** (CUDA 9.1)
    - Architecture: **NVIDIA Volta**
    - **Memory: 12 GB**
    - **CUDA CORES: 5120**

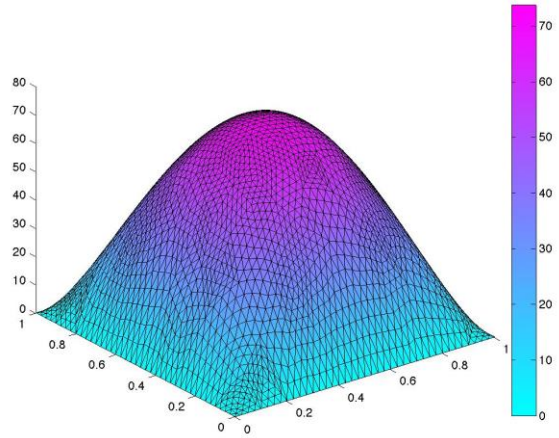
2D Diffusion Equation

$$-(\text{div } \mathrm{K}\nabla \mathrm{u}) = \mathrm{f} \quad \text{in} \quad \Omega = [0,1] \times [0,1]$$

with homogeneous Dirichlet BC, $u = 0$ on $\partial\Omega$, and

$$K = \begin{bmatrix} a & c \\ c & b \end{bmatrix} \text{ with } \begin{cases} a = \epsilon + \cos^2(\alpha) \\ b = \epsilon + sin^2(\alpha) \\ c = \cos(\alpha)\ \sin(\alpha) \end{cases}$$

Discretized using linear finite elements on triangular meshes by uniform refinement.

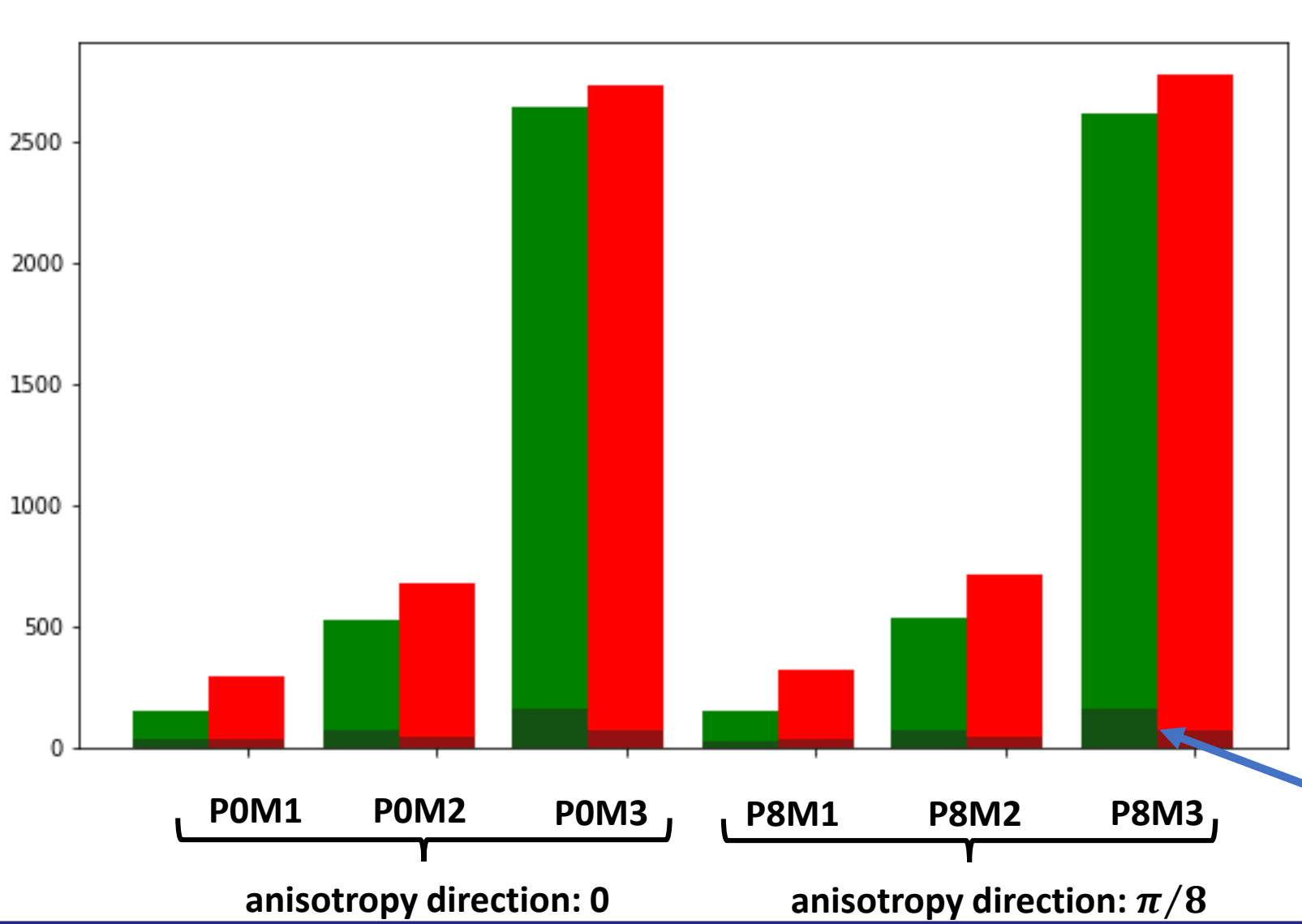


Test Case: anisotropic diffusion with:

$$\epsilon = 0.001(anisotropy\ strain)$$

$$\alpha = 0\ and \frac{\pi}{8}\ (anisotropy\ direction)$$

Total Time Comparison



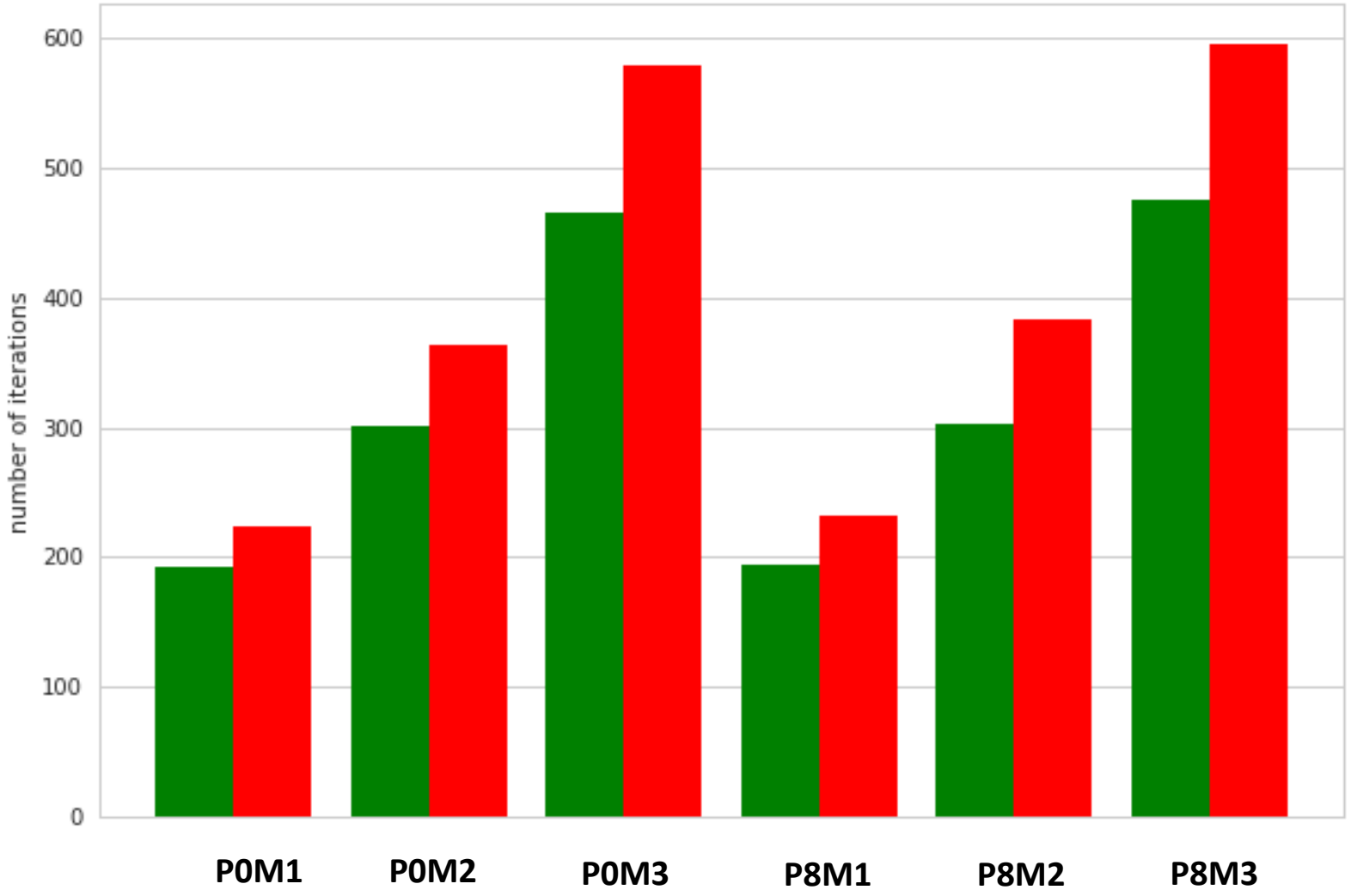
NVIDIA AmgX

BootCMatch

Matrix	n	nnz
P*M1	168577	1177473
P*M2	673025	4706049
P*M3	2689537	18816513

Building time

Iterations number



NVIDIA AmgX

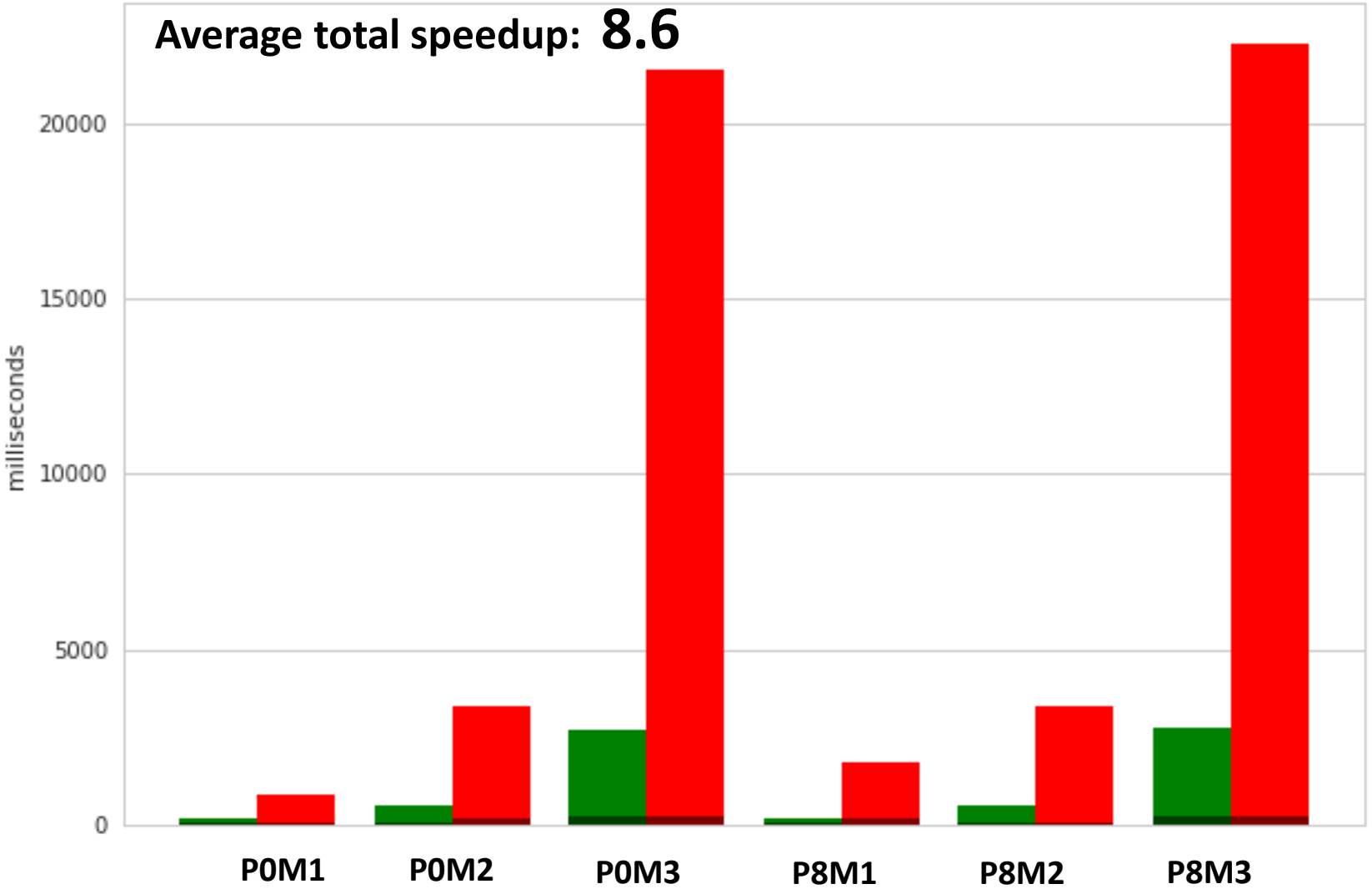
BootCMatch

Matrix	n	nnz
P*M1	168577	1177473
P*M2	673025	4706049
P*M3	2689537	18816513

BootCMatch requires:

$10\% \sim 20\%$ fewer iterations

Time Comparison with AmgX Classic AMG



NVIDIA AmgX

BootCMatch

Matrix	n	nnz
P*M1	168577	1177473
P*M2	673025	4706049
P*M3	2689537	18816513

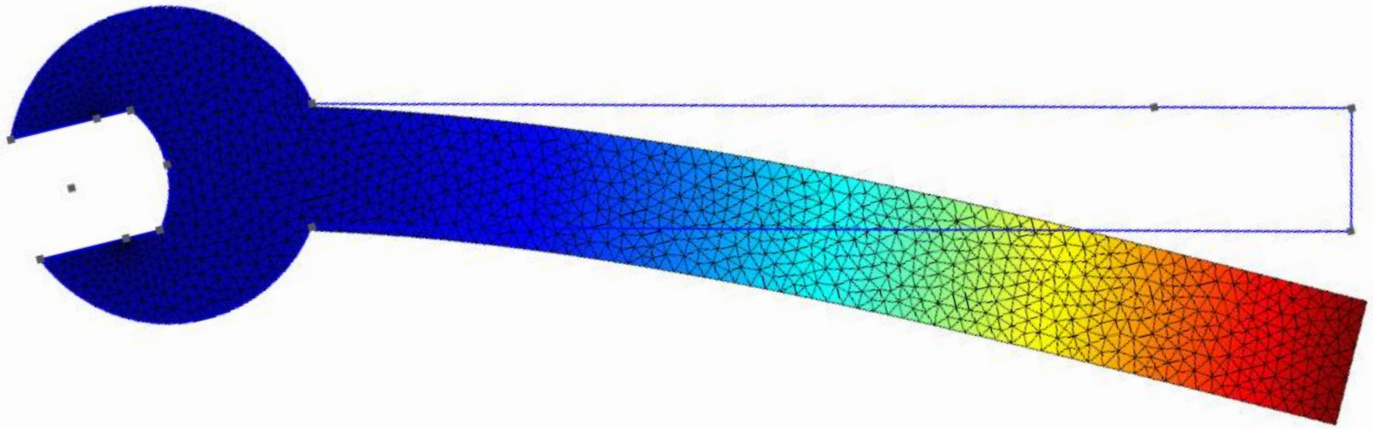
The parallel implementation of **Classic AMG** in AMGX is based largely on **HYPRE**

[Hans De Sterck, et al Reducing complexity in parallel algebraic multigrid preconditioners 2006]

Systems arising from 2D Linear Elasticity Problems

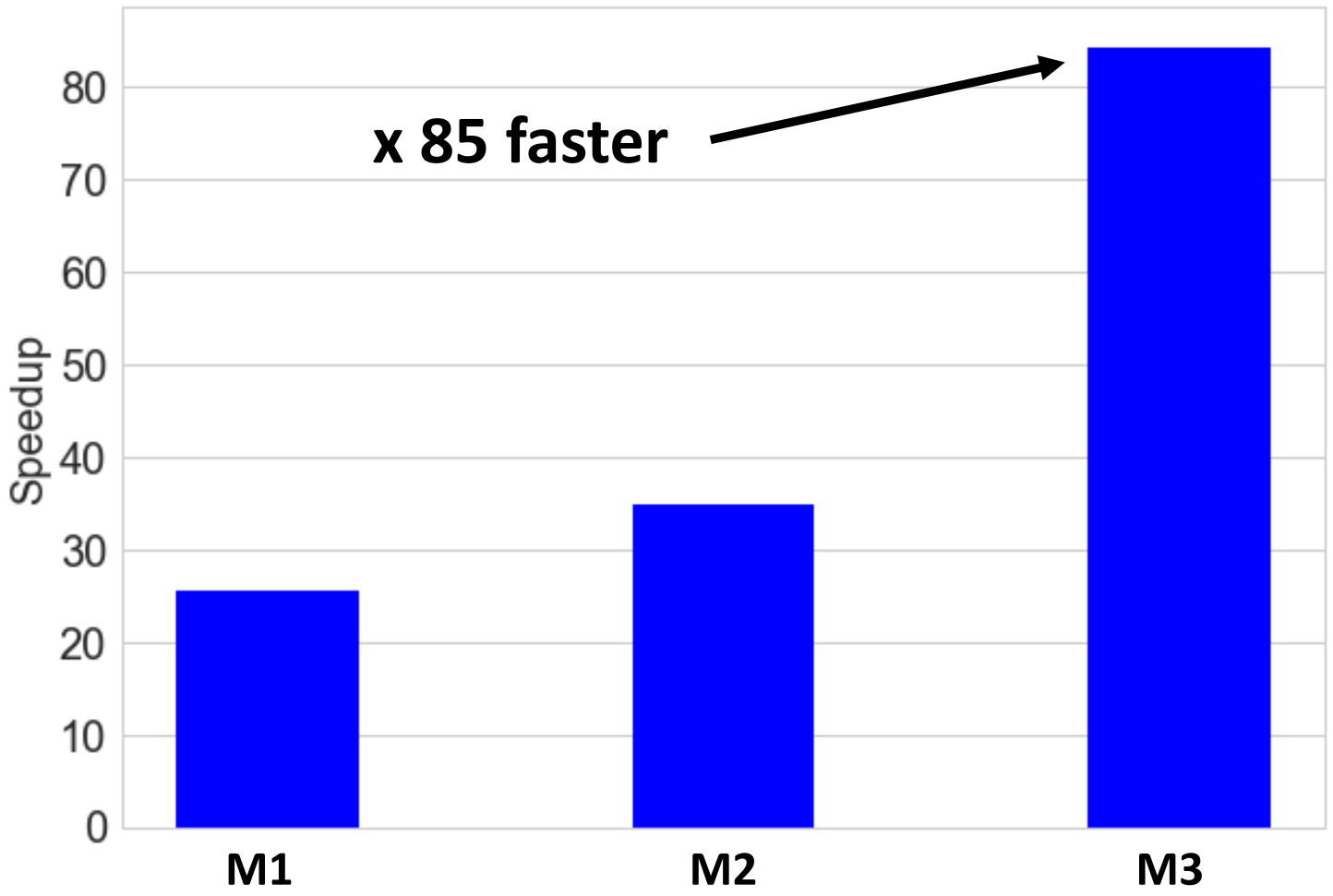
- **Lamé equation:**

$$\mu\Delta \mathrm{u} + (\lambda + \mu)\nabla(\mathrm{div}\,\mathrm{u}) = \mathrm{f} \qquad \mathrm{x} \in \Omega$$



- $\mu = 0.42$ and $\lambda = 1.7$ one side of the beam is fixed and the opposite end pushed downwards (Dirichlet and traction conditions)
- each scalar component of the displacement vector is chosen as a block unknown: unknown-based matrix ordering
- linear finite 2D elements on triangular mesh with three sizes (264450, 1053186, 4203522)

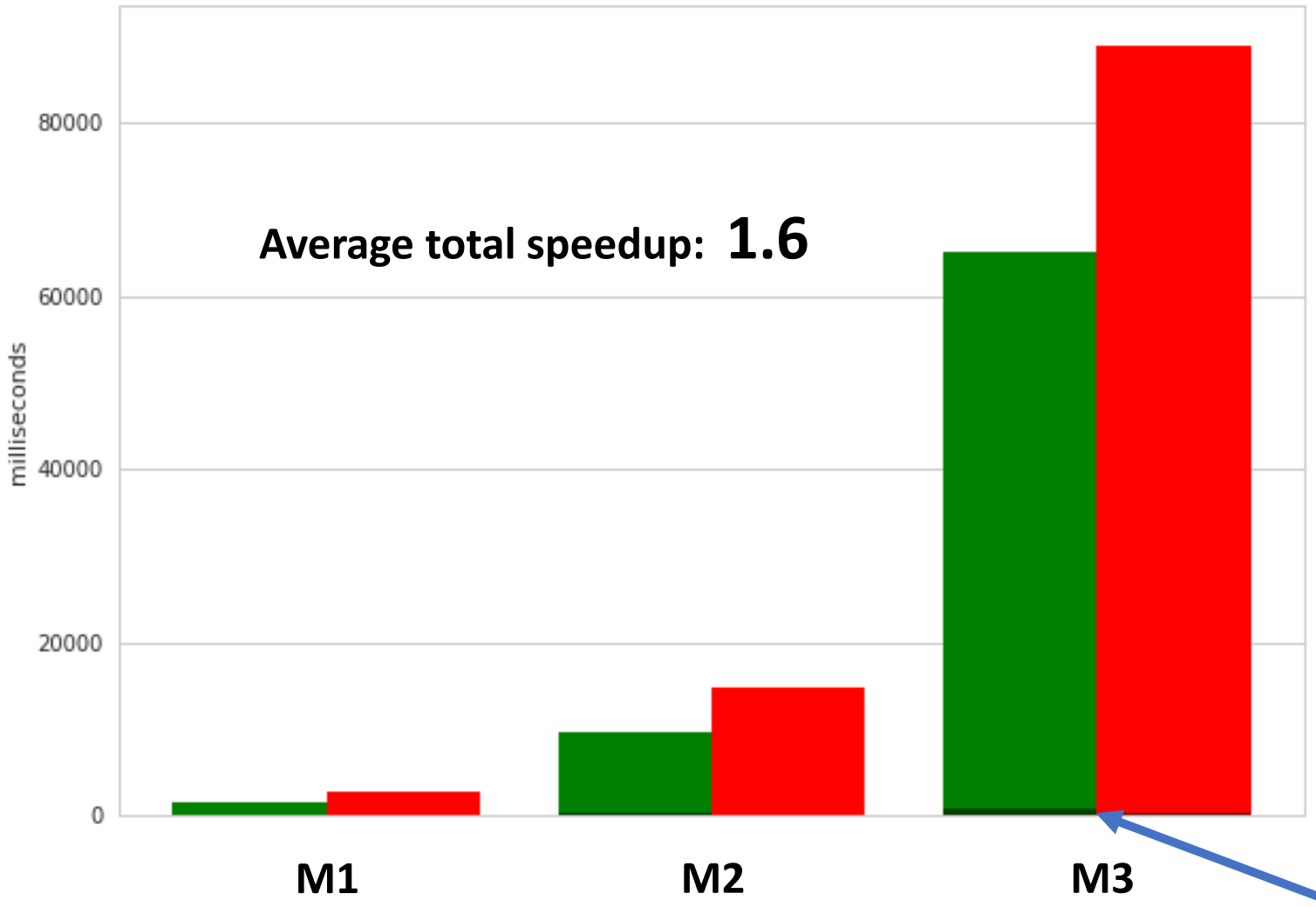
BootCMatch: GPU vs CPU speedup



Intel(R) Xeon(R) Platinum 8176 CPU
Titan V GPU

Matrix	n	nnz
M1	264450	1710589
M2	1053186	6829053
M3	4203522	50375672

Time Comparison

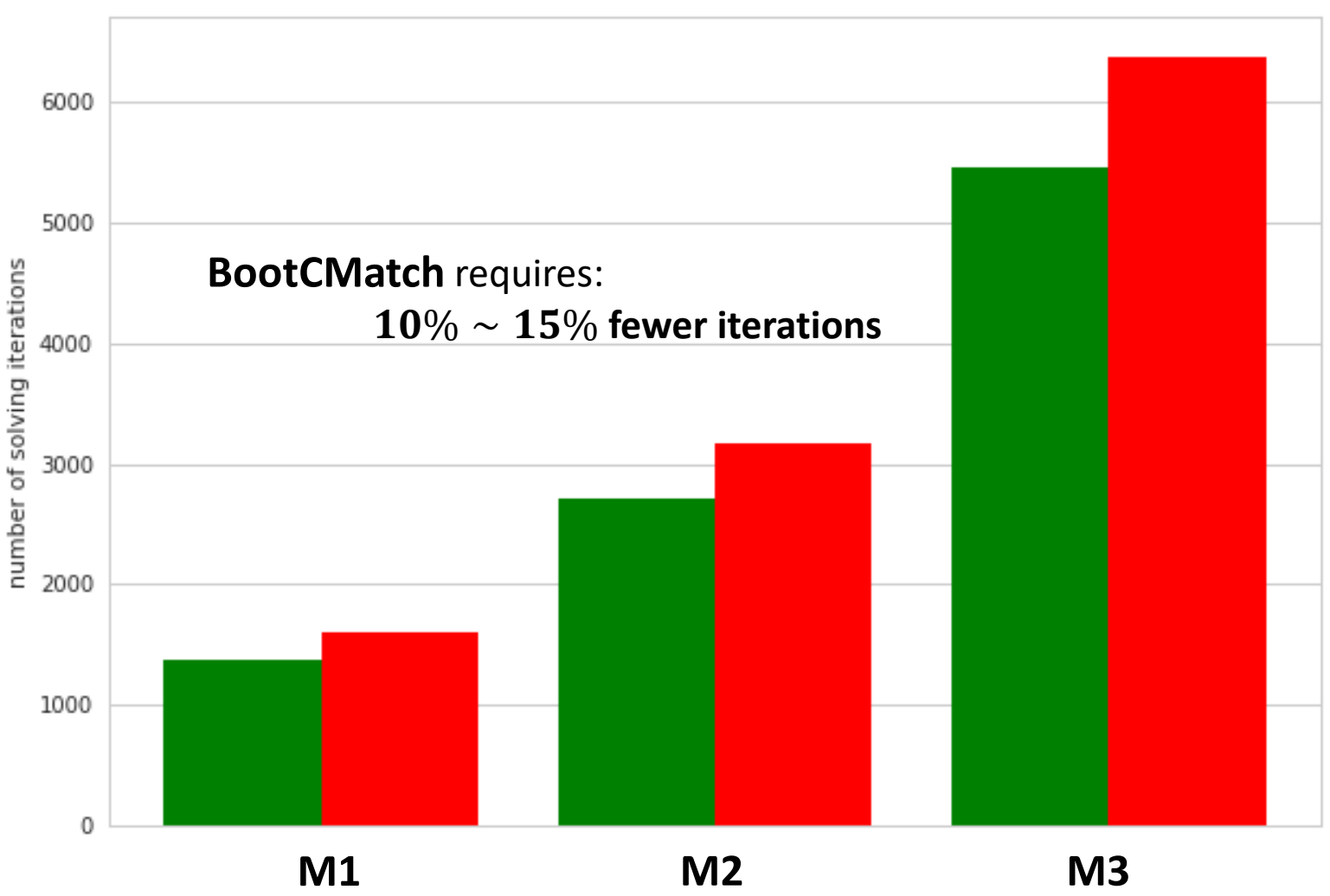


NVIDIA AmgX

BootCMatch

Matrix	n	nnz
M1	264450	1710589
M2	1053186	6829053
M3	4203522	50375672

Solving iterations number

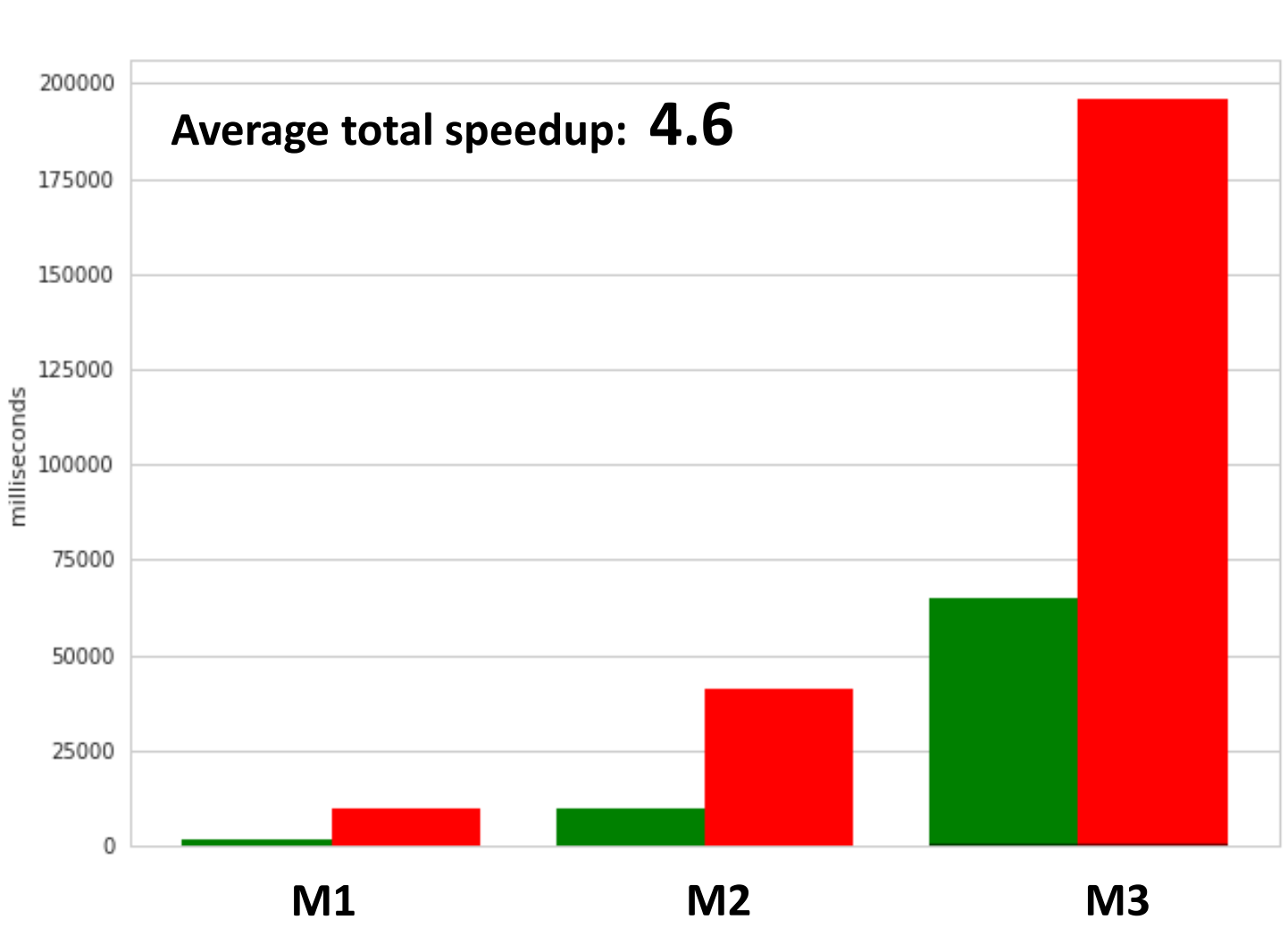


NVIDIA AmgX

BootCMatch

Matrix	n	nnz
M1	264450	1710589
M2	1053186	6829053
M3	4203522	50375672

Time Comparison with AmgX Classic AMG



Matrix	n	nnz
M1	264450	1710589
M2	1053186	6829053
M3	4203522	50375672

Combining Dynamic FSAI and BootCMatch

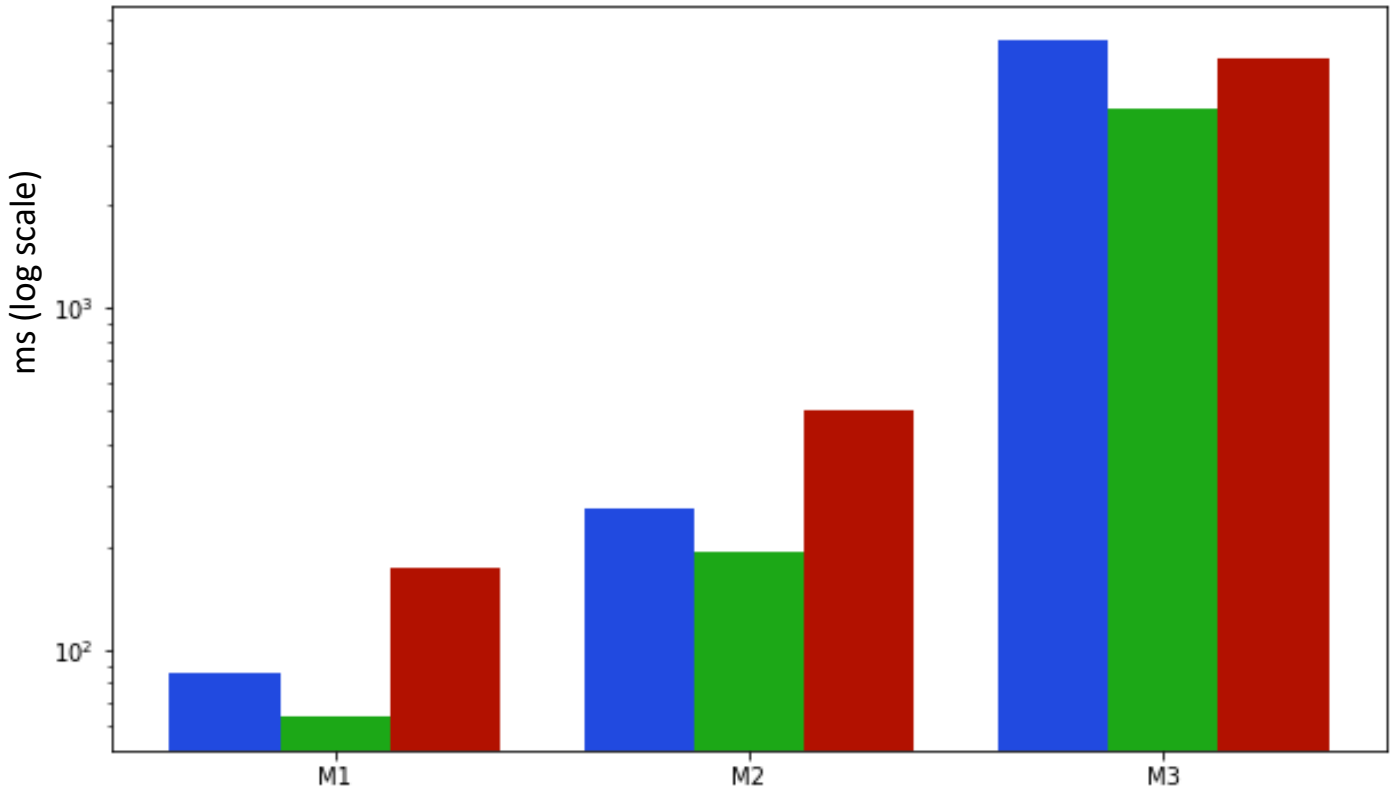
Limitation of classic AMG packages

- Jacobi or Gauss-Seidel are usually adopted as smoothers
- AMG may fail in ill-conditioned problems as in the linear elasticity problem shown before
- The idea is to apply the *power* of the FSAI approach to improve the smoothing phase of the AMG

Combining Dynamic FSAI and BootCMatch (1/3)

Systems arising from 3D Linear Elasticity Problems:

Solve time:



BootCMatch

BootCMatch+FSAI

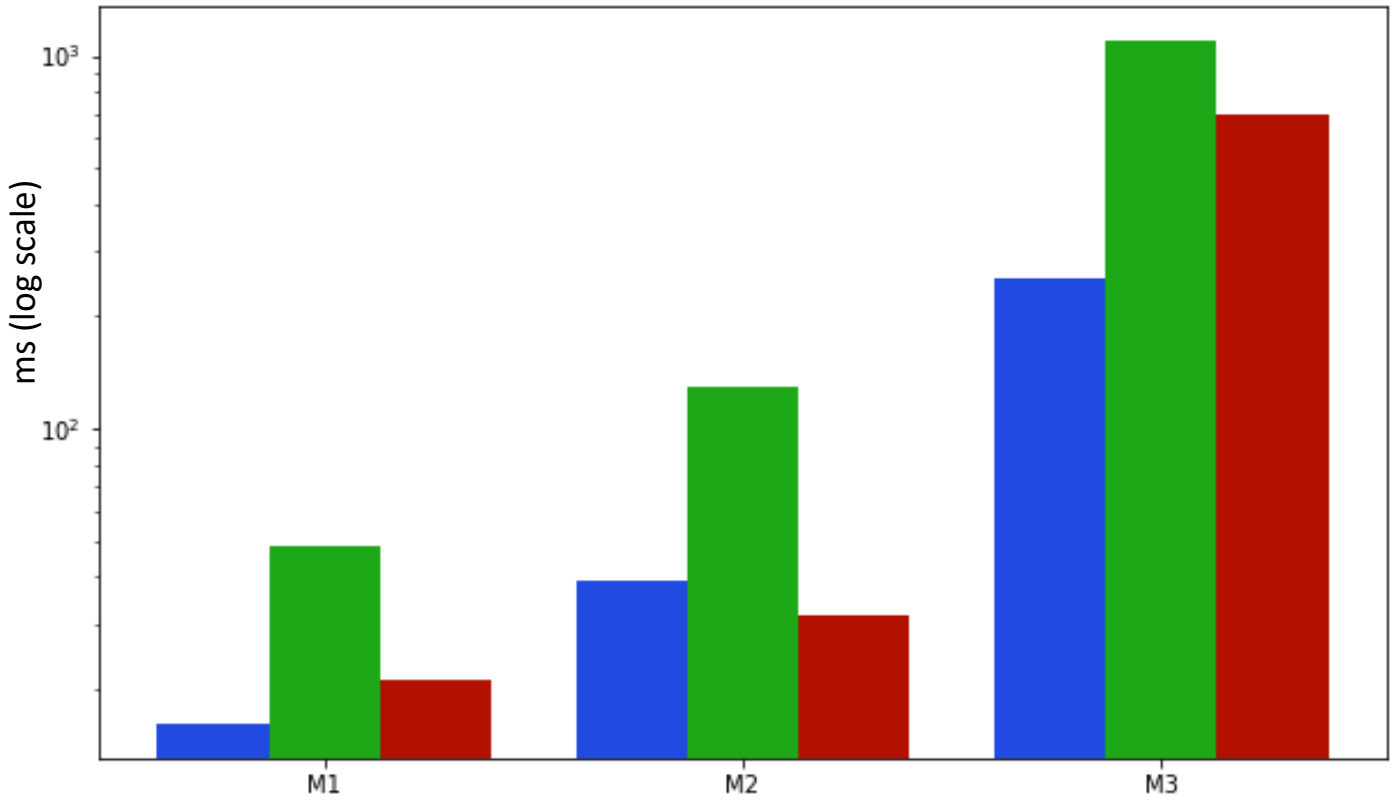
NVIDIA AmgX

Matrix	n	nnz
M1	15795	239717
M2	111843	1699647
M3	839619	64790700

Combining Dynamic FSAI and BootCMatch (2/3)

Systems arising from 3D Linear Elasticity Problems:

Building time:



BootCMatch

BootCMatch+FSAI

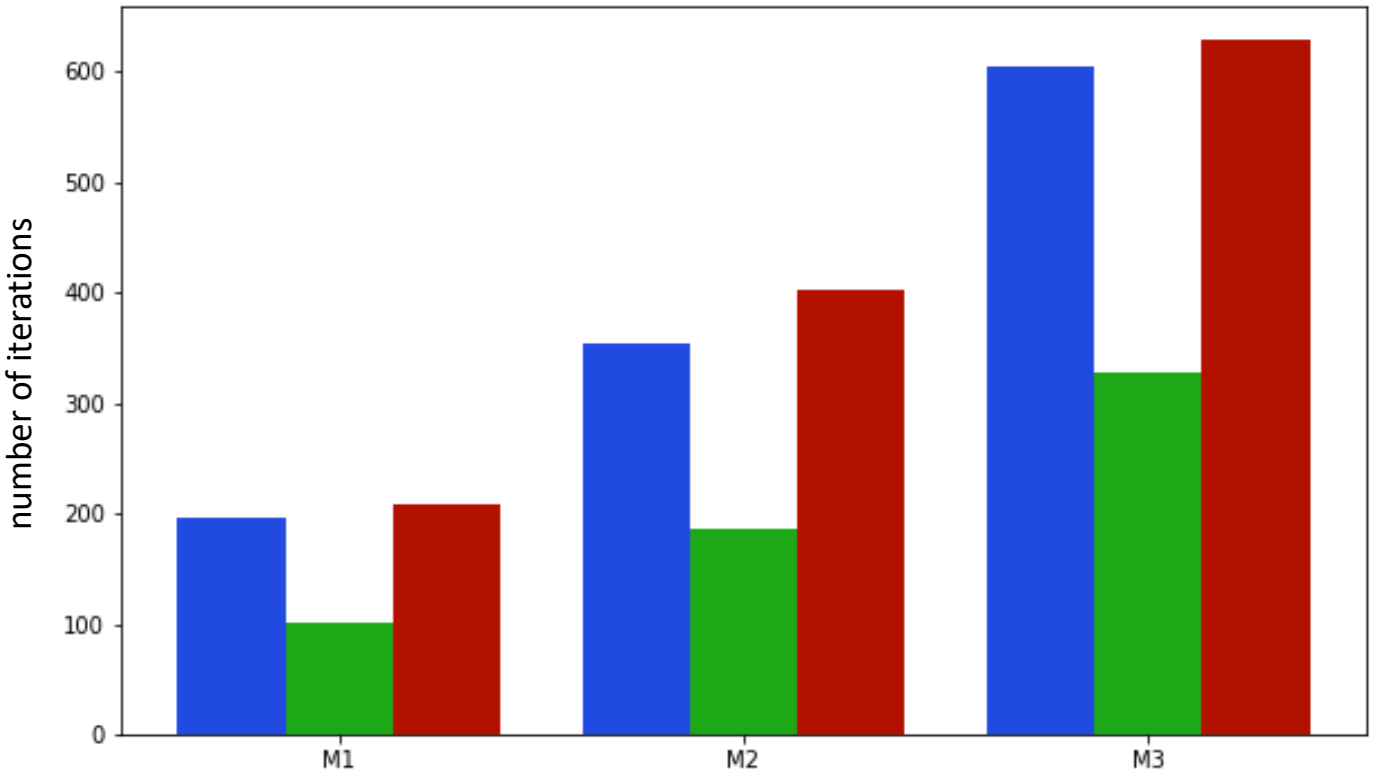
NVIDIA AmgX

Matrix	n	nnz
M1	15795	239717
M2	111843	1699647
M3	839619	64790700

Combining Dynamic FSAI and BootCMatch (3/3)

Systems arising from 3D Linear Elasticity Problems:

Iterations number:



BootCMatch

BootCMatch+FSAI

NVIDIA AmgX

Matrix	n	nnz
M1	15795	239717
M2	111843	1699647
M3	839619	64790700

Future Work

- Further optimizations in the building phase
  - In the graph matching and triple matrix product
- Increase the number of software options
  - Including (complete / incomplete) **LU** decomposition for the coarsest solver
- Support for distributed **multi-GPUs** computation
  - For both the building and the solver phase
- Release in the public domain
  - Currently the code is available on demand

Thanks for Your Attention