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
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.
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:

$$M_1^{-1}AM_2^{-1}M_2x = M_1^{-1}b$$

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!
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

\[ Ax = b \] where \( A \in \mathbb{R}^{n \times n} \) s.p.d.

Two-level method

1. solve \( Ax_1 = b \rightarrow \) smoother iterations
2. compute residual \( r = b - Ax_1 \)
3. restrict residual to a coarse space \( r_c = P_c^T r \)
4. solve \( A_c x_c = r_c \rightarrow \) coarse-grid correction [solved in a recursive way]
5. update solution \( x = x_1 + P_c r_c \)
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
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
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


In the recent 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

Gather of small systems on GPU (1/2)

- On the CPU each thread collects a whole system

![Pattern of a row of F and 1 thread](image)

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 |

**Binary search on each A row**

**Expanded system to be gathered**

<table>
<thead>
<tr>
<th>1</th>
<th>35</th>
<th>78</th>
<th>111</th>
<th>291</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35</td>
<td>78</td>
<td>111</td>
<td>291</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>78</td>
<td>111</td>
<td>291</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>78</td>
<td>111</td>
<td>291</td>
</tr>
<tr>
<td>1</td>
<td>35</td>
<td>78</td>
<td>111</td>
<td>291</td>
</tr>
</tbody>
</table>

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 registers 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.


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

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 $k_{iter}$, $s$, $\tau$, $\varepsilon$;

**Output:** Matrix $G$;

if $G_0$ is given then
  $\tilde{G}_0 \leftarrow \text{diag}(G_0)^{-1}G_0$;
else
  $\tilde{G}_0 \leftarrow I$;
end

for $i = 1, n$ do
  for $k = 0, k_{iter} - 1$ do
    Compute $\nabla \psi_{k,i}$;
    Compute $\tilde{P}^{k+1}_i$ by adding to $\tilde{P}^k_i$ the indices of the $s$ largest components of $\nabla \psi_{k,i}$;
    Gather $A[\tilde{P}^{k+1}_i, \tilde{P}^{k+1}_i]$ and $A[\tilde{P}^{k+1}_i, i]$ from $A$;
    Solve $A[\tilde{P}^{k+1}_i, \tilde{P}^{k+1}_i] \tilde{\theta}_{k+1,i} = -A[\tilde{P}^{k+1}_i, i]$
    if $\psi_{k+1,i} \leq \varepsilon \cdot \psi_{0,i}$ then exit the loop over $k$;
    Drop the entries of $\tilde{\theta}_{k+1,i}$ such that $\tilde{\theta}_{k+1,i} \leq \tau \| \tilde{\theta}_{k+1,i} \|_2$;
  end
  $\tilde{d}_i \leftarrow (-\tilde{\theta}^T_{k+1,i} A[\tilde{P}^{k+1}_i, i])^{-1}$;
  $\tilde{g}_{k+1,i} \leftarrow \tilde{d}_i \tilde{g}_{k+1,i}$;
  Scatter the components of $\tilde{g}_{k+1,i}$ into $G_{k+1}[i, \tilde{P}^{k+1}_i]$;
end
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

<table>
<thead>
<tr>
<th>IW</th>
<th>4</th>
<th>8</th>
<th>3</th>
<th>7</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR</td>
<td>1.2</td>
<td>10.3</td>
<td>4.5</td>
<td>8.9</td>
<td>...</td>
</tr>
<tr>
<td>JWN</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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:

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

Strong Scalability Test
on V100 GPUs

![Graph showing strong scalability test results on V100 GPUs for Matrix 10, Matrix 11, and Matrix 16. The x-axis represents the number of GPUs (nGPU), and the y-axis represents the speed-up. The graphs show linear scalability with real results compared to ideal results.](image-url)
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](https://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 $\widehat{A}$, a matrix of weights that characterize the strength of the connection (i.e., the similarity) between pairs of variables.
- $\widehat{A}$ has the same dimension and sparsity pattern of $A$ 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 undirected 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:

$$C(M) = \sum_{(i,j) \in M} c_{ij}$$

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

- $M'$ matching in $G$
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
• $O(n(n + \text{nnz})\log(n))$

Preis algorithm:
• half-approximate
• $O(\text{nnz})$

Auction algorithm:
• Near optimal
• $O(n \cdot \text{nnz log}(n))$

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
- $O(n(n + nnz)\log(n))$

**Preis algorithm:**
- half-approximate
- $O(nnz)$

**Auction algorithm:**
- Near optimal
- $O(n \cdot nnz \log(n))$

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

Previous algorithms

Suitor
Preis
Auction
Optimal

G1
G2
G3
G4
Coarsening Algorithm

• $A^0 = A, \ k = 0, w^0 = w$

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

• Where $P^k$ (prolongator) is a constant piecewise operator obtained from the aggregates produced by the pairwise aggregation
Aggressive Coarsening Algorithm

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

  Can be repeated \( n \) times for each level \( k \)

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

  \[
  \begin{align*}
  &A_0 \quad p^0 \\
  &A_1' \quad A_1 \quad p_1 = p_1' \ p_1 \\
  &A_2' \quad A_2 \quad p_2 = p_2' \ p_2 \\
  &A_m' \quad A_m \quad p^m = p^m' \ p^m
  \end{align*}
  \]
Sparse Matrix Dense Vector product [SpMxV]  
General case

Sparse Matrix $A$:

![CSR row diagram]

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 $A[i,j]$ 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

Sparse Matrix $A$:

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 $A[i,j]$ and multiplies it by $x[j]$
3. Each warp performs an internal sum reduction

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

Threads are executed but not exploited for the computation
Improving Load Balancing via “mini-warp”

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

Mini-wars possible configuration:
- 16 mini-wars of 2 threads
- 8 mini-wars of 4 threads
- 4 mini-wars of 8 threads
- 2 mini-wars of 16 threads
SpMxV with “mini-warp”

Sparse Matrix $A$:

Warp:

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

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

Result on NVIDIA Volta architecture

Average total speedup: 1.8

Three systems arising from Linear Elasticity Problems:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>( n )</th>
<th>( nnz )</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>( 22 \times 10^4 )</td>
<td>( 17 \times 10^5 )</td>
</tr>
<tr>
<td>M2</td>
<td>( 11 \times 10^5 )</td>
<td>( 68 \times 10^5 )</td>
</tr>
<tr>
<td>M3</td>
<td>( 42 \times 10^5 )</td>
<td>( 51 \times 10^6 )</td>
</tr>
</tbody>
</table>
Flexible Conjugate Gradient (FCG) Optimizations [1]

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

- 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
**Flexible Conjugate Gradient (FCG) Optimizations** [2]

- 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 \(\frac{1}{3}\) 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)
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 ≤ $10^{-6}$

• **Hardware:**
  - NVIDIA **TITAN V** (CUDA 9.1)
    - Architecture: NVIDIA **Volta**
    - Memory: 12 GB
    - CUDA CORES: 5120
2D Diffusion Equation

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

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:

\[\epsilon = 0.001 \text{(anisotropy strain)}\]

\[\alpha = 0 \text{ and } \frac{\pi}{8} \text{ (anisotropy direction)}\]
Total Time Comparison

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

Building time

<table>
<thead>
<tr>
<th>P0M1</th>
<th>P0M2</th>
<th>P0M3</th>
<th>P8M1</th>
<th>P8M2</th>
<th>P8M3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

anisotropy direction: 0

anisotropy direction: $\pi/8$
**Iterations number**

- **Matrix**
  - **P*M1**: 168577, 1177473
  - **P*M2**: 673025, 4706049
  - **P*M3**: 2689537, 18816513

**BootCMatch** requires:
10% ~ 20% fewer iterations
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:
  \[
  \mu \Delta u + (\lambda + \mu) \nabla (\text{div } u) = f \quad x \in \Omega
  \]

- \( \mu = 0.42 \) 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

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>264450</td>
<td>1710589</td>
</tr>
<tr>
<td>M2</td>
<td>1053186</td>
<td>6829053</td>
</tr>
<tr>
<td>M3</td>
<td>4203522</td>
<td>50375672</td>
</tr>
</tbody>
</table>
Time Comparison

Average total speedup: 1.6

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

Building time
Solving iterations number

BootCMatch requires: 10% ~ 15% fewer iterations

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>264450</td>
<td>1710589</td>
</tr>
<tr>
<td>M2</td>
<td>1053186</td>
<td>6829053</td>
</tr>
<tr>
<td>M3</td>
<td>4203522</td>
<td>50375672</td>
</tr>
</tbody>
</table>

NVIDIA AmgX

BootCMatch
Time Comparison with AmgX Classic AMG

Average total speedup: 4.6

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>264450</td>
<td>1710589</td>
</tr>
<tr>
<td>M2</td>
<td>1053186</td>
<td>6829053</td>
</tr>
<tr>
<td>M3</td>
<td>4203522</td>
<td>50375672</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>15795</td>
<td>239717</td>
</tr>
<tr>
<td>M2</td>
<td>111843</td>
<td>1699647</td>
</tr>
<tr>
<td>M3</td>
<td>839619</td>
<td>64790700</td>
</tr>
</tbody>
</table>
Combining Dynamic FSAI and BootCMatch (2/3)

Systems arising from 3D Linear Elasticity Problems:

Building time:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>15795</td>
<td>239717</td>
</tr>
<tr>
<td>M2</td>
<td>111843</td>
<td>1699647</td>
</tr>
<tr>
<td>M3</td>
<td>839619</td>
<td>64790700</td>
</tr>
</tbody>
</table>
Combining Dynamic FSAI and BootCMatch (3/3)

Systems arising from 3D Linear Elasticity Problems:

Iterations number:

<table>
<thead>
<tr>
<th>Matrix</th>
<th>n</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>15795</td>
<td>239717</td>
</tr>
<tr>
<td>M2</td>
<td>111843</td>
<td>1699647</td>
</tr>
<tr>
<td>M3</td>
<td>839619</td>
<td>64790700</td>
</tr>
</tbody>
</table>
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