Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Hartwig Anzt, Edmond Chow, Thomas Huckle, Jack Dongarra

http://www.icl.utk.edu/~hanzt/talks/ISAI.pdf
Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b$, $A \in \mathbb{R}^{n \times n}$.

Compute factorization

$$(A = LU)_S$$

for some sparsity pattern $S$ \begin{cases} 
S = \mathbb{R}^{n \times n} & \text{exact fact.} \\
S = spy(A) & \text{ILU(0)}
\end{cases}$

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z$, $Ux = y$. 
Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b$, $A \in \mathbb{R}^{n \times n}$.

Compute factorization

$$(A = LU)_S$$

for some sparsity pattern $S$ \begin{align*}
S &= \mathbb{R}^{n \times n} \quad \text{exact fact.} \\
S &= \text{spy}(A) \quad \text{ILU(0)}
\end{align*}

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z$, $Ux = y$.

- Exact triangular solves
  - Inherently sequential, level scheduling often provides little parallelism.
Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem $Ax = b$, $A \in \mathbb{R}^{n \times n}$.

Compute factorization

$$\left( A = LU \right)_S$$

for some sparsity pattern $S$ \begin{align*}
S & = \mathbb{R}^{n \times n} \quad \text{exact fact.} \\
S & = \text{spy}(A) \quad \text{ILU(0)}
\end{align*}

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems $Ly = z$, $Ux = y$.

- Exact triangular solves
  - Inherently sequential, level scheduling often provides little parallelism.
- Replace with approximate triangular solve
  - Relaxation steps like (Block) Jacobi iterations.
Incomplete Sparse Approximate Inverse (ISAI) Preconditioner

Goal: Find solution to sparse linear problem \( Ax = b, A \in \mathbb{R}^{n \times n} \).

Compute factorization 
\[
(A = LU) \quad \text{for some sparsity pattern } S \quad \left\{ \begin{array}{l}
S = \mathbb{R}^{n \times n} \quad \text{exact fact.} \\
S = \text{spy}(A) \quad \text{ILU(0)}
\end{array} \right.
\]

Incomplete factorizations attractive for preconditioning iterative solvers.

Preconditioner application involves solving triangular systems \( Ly = z, Ux = y \).

- Exact triangular solves
  - Inherently sequential, level scheduling often provides little parallelism.
- Replace with approximate triangular solve
  - Relaxation steps like (Block) Jacobi iterations.
- Incomplete Sparse Approximate Inverse (ISAI)\(^1\):
  \[
  (L \cdot M_L = I) S^* \quad \text{for some sparsity pattern } S^*, \text{ e.g. } S^* = \text{spy}(A) \quad \text{ISAI(1)}
  \]
  \[
  S^* = \text{spy}(A^2) \quad \text{ISAI(2)}
  \]
  \[
  S^* = \text{spy}(A^3) \quad \text{ISAI(3)}
  \]
  \[
  S^* = JAC(4)
  \]

\(^1\)Huckle, Anzt, Dongarra “Parallel Preconditioning”. In: SIAM PP 2016.
Generating ISAI for triangular factor

\[ M_L \text{ with } (L \cdot M_L = I)_{S^*} \text{ for } S^* = spy(A) \]

for i=1:n
  J = find(M(:,i));
  generate L(J,J);
  solve L(J,J) M(J,i) = e_i(J);
  insert M(J,i) into M;
end

Algorithm composes into solving a set of small triangular systems
Generating ISAI for triangular factor

for i=1:n
    J = find(M(:,i));
    generate L(J,J);
    solve L(J,J) M(J,i) = e_i(J);
    insert M(J,i) into M;
end
Generating ISAI for triangular factor

for i = 1:n
    J = find(M(:,i));
    generate L(J,J);
    solve L(J,J) M(J,i) = e_i(J);
    insert M(J,i) into M;
end
Generating ISAI for triangular factor

for $i=1:n$
\[
J = \text{find}(M(:,i));
\]
\text{generate } L(J,J);
\text{solve } L(J,J) \times M(J,i) = e_i(\text{J});
\text{insert } M(J,i) \text{ into } M;
end
Generating ISAI for triangular factor

for i=1:n
    J = find(M(:,i));
    generate L(J,J);
    solve L(J,J) M(J,i) = e_i(J);
    insert M(J,i) into M;
end
Generating ISAI for triangular factor

for i=1:n
    J = find(M(:,i));
    generate L(J,J);
    solve L(J,J) M(J,i) = e_i(J);
    insert M(J,i) into M;
end

batched generation of M(:,i), i=1..n

generate set of small systems

trsv for system i
Generating ISAI for triangular factor

for i=1:n
    J = find(M(:,i));
    generate L(J,J);
    solve L(J,J) M(J,i) = e_i(J);
    insert M(J,i) into M;
end
for i=1:n
    J = find(M(:,i));
    generate L(J,J);
    solve L(J,J) M(J,i) = e_i(J);
    insert M(J,i) into M;
end

Generating ISAI for triangular factor

**Four Batched Routines:**

- **Find the locations in each row**
  - store size information for small tri-systems
  - store nonzero-locations to find matches

- **Generate batch of small triangular systems**
  - different sizes in uniformly-sized blocks

- **Batched trsv**
  - different sizes
  - non-coalescent in memory (uniform blocks)
  - use kernel-switch for hard-coded sizes

- **Batched re-insertion into sparse ISAI matrix**
  - non-coalescent reads/writes
Generating ISAI for triangular factor

for \( i = 1:n \)
  \[ J = \text{find}(M(:,i)) \]
  generate \( L(J,J) \);
  solve \( L(J,J) \times M(J,i) = e_i(J) \);
  insert \( M(J,i) \) into \( M \);
end

\[ L(J,J) \times M(J,i) = e_i(J) \]
Generating ISAI using one batched routine

1. Shrink the thread block to the number of nonzero locations in column i of M. Assign each thread to one row.

2. Traverse the relevant rows in L and M (starting from the main diagonal) to identify matches. Load the matching entries into thread-local registers.

3. The right-hand-side is given as 1 for the first thread and 0 for all others.

4. Triangular solve using remote register reads.

5. Insertion of the solution into the ISAI preconditioner structure. Every thread writes its solution component.
Generating ISAI using one batched routine

1. Shrink the thread block to the number of nonzero locations in column i of M. Assign each thread to one row.

2. Traverse the relevant rows in L and M (starting from the main diagonal) to identify matches. Load the matching entries into thread-local registers.

3. The right-hand-side is given as 1 for the first thread and 0 for all others.

4. Triangular solve using remote register reads.

5. Insertion of the solution into the ISAI preconditioner structure. Every thread writes its solution component.

- Generate triangular system in **registers** only & batched triangular solve in registers².

---

Mapping sparsity pattern

Nonzero locations in $L$ (blue) in $M_L$ (green), and matching locations (red).

- $S^* = JAC(2)$
- $S^* = JAC(6)$
- $S^* = spy(A^2)$

- Generate triangular system in registers only & batched triangular solve in registers$^2$.
- Identify matching locations by traversing triangular matrices from the diagonal.
Two strategies for generating ISAI

Combination of 4 batched routines
- Batch of systems in main memory
- Coalesced memory access

One batched routine
- Linear system in registers
- \texttt{shfl()} for communication
Performance comparison of ISAI generation

Speedup of batched routine vs. sequence of batched routines for block Jacobi pattern.

Nvidia K40 GPU
• 1.4 TF DP, 280 GB/s (4.86 : 1)
• Local memory as L1 cache/shared memory

Nvidia P100 GPU
• 5.3 TF DP, 720 GB/s (7.36 : 1)
• Dedicated L1/texture cache

(MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)
Performance comparison of ISAI generation

Speedup of batched routine vs. sequence of batched routines for ISAI pattern.

Nvidia K40 GPU
- 1.4 TF DP, 280 GB/s (4.86 : 1)
- Local memory as L1 cache/shared memory

Nvidia P100 GPU
- 5.3 TF DP, 720 GB/s (7.36 : 1)
- Dedicated L1/texture cache

(MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)
Incomplete Sparse Approximate Inverse in Iterative Solvers

Overhead for ISAI generation.

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)
Performance of Batched ISAI implementation

- ISAI(1):
  - 251 systems
  - Overhead: 37.8%

- ISAI(2):
  - 192 systems
  - Overhead: 77.4%

- ISAI(3):
  - 164 systems
  - Overhead: 113.9%

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)
Performance of Batched ISAI implementation

- **ISAI(1)**: 251 systems, overhead -- 20.9%
- **ISAI(2)**: 192 systems, overhead -- 13.9%
- **ISAI(3)**: 164 systems, overhead -- 6.6%

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)
Performance of Batched ISAI implementation

ISAI generation successful for 251 of 460 UFSMC matrices

- IDR[4]
- IDR[4] + ILU[0]

Fastest solver without prec. setup

Fastest solver with prec. setup

(Nvidia K40 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)
Performance of Batched ISAI implementation

ISAI generation successful for 251 of 460 UFSMC matrices

- IDR[4]
- IDR[4] + ILU[0]

Fastest solver without prec. setup
(Nvidia KP100 GPU, MAGMA 2.1.0, ILU taken from cuSPARSE 8.0)

Fastest solver with prec. setup
This research is based on a cooperation between Hartwig Anzt (University of Tennessee), Edmond Chow (Georgia Tech), and Thomas Huckle (TU Munich), and partly funded by the Department of Energy.

All functionalities are included in the MAGMA 2.2.0 release.

**MAGMA SPARSE**

<table>
<thead>
<tr>
<th>ROUTINES</th>
<th>BiCG, BiCGSTAB, Block-Asynchronous Jacobi, CG, CGS, GMRES, IDR, Iterative refinement, LOBPCG, LSQR, QMR, TFQMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRECONDITIONERS</td>
<td>ILU / IC, Jacobi, ParILU, ParILUT, Block Jacobi, ISAI</td>
</tr>
<tr>
<td>KERNELS</td>
<td>SpMV, SpMM</td>
</tr>
<tr>
<td>DATA FORMATS</td>
<td>CSR, ELL, SELL-P, CSR5, HYB</td>
</tr>
</tbody>
</table>

http://icl.cs.utk.edu/magma/

http://www.icl.utk.edu/~hanzt/talks/ISAI.pdf