Large scale SVD using polar decomposition

M. Faverge
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   ▶ The QDWH-based Polar Decomposition
   ▶ The ZOLO-based Polar Decomposition
   ▶ Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
SVD - Singular Value Decomposition

\[ A = U \Sigma V^T \]

- A is general matrix
- \( \Sigma \) are the singular values of \( A \)
- \( U \) are the left singular vectors
- \( V \) are the right singular vectors
Singular Value Decomposition

\[ A = U \Sigma V^T \]

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:
  - GE2BND: Reduce the general matrix to general band
  - BND2BD: Reduce the general band to bidiagonal
  - BD2VAL: Compute the singular values from the bidiagonal
Singular Value Decomposition

\[ A = U \Sigma V^T \]

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:
  - **GE2BND** Reduce the general matrix to general band
  - **BND2BD** Reduce the general band to bidiagonal
  - **BD2VAL** Compute the singular values from the bidiagonal
Singular Value Decomposition

\[ A = U \Sigma V^T \]

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:
  - **GE2BND** Reduce the general matrix to general band
  - **BND2BD** Reduce the general band to bidiagonal (PLASMA)
  - **BD2VAL** Compute the singular values from the bidiagonal
Singular Value Decomposition

\[ A = U \Sigma V^T \]

- Focus on getting the singular values only (GEVAL)
- Use three steps algorithms:
  - GE2BND: Reduce the general matrix to general band
  - BND2BD: Reduce the general band to bidiagonal (PLASMA)
  - BD2VAL: Compute the singular values from the bidiagonal (MKL)
Two(-Three) stages algorithms

1. Reduction to tridiagonal form $A = U' B V'^T$
   - $B$ is a bidiagonal matrix
   - $U'$ and $V'$ are unitary matrices

2. Find the singular values of the bidiagonal matrix $B$: $B = Q * \Sigma * P^t$

3. Eventually compute the eigenvectors: $U = U' Q$, and $V^t = (V' P)^t$

Problem: Reduction to tridiagonal is using BLAS 2
Two(-Three) stages algorithms

1. Reduction to tridiagonal form \( A = U' B V'^T \)
   - \( B \) is a bidiagonal matrix
   - \( U' \) and \( V' \) are unitary matrices

2. Find the singular values of the bidiagonal matrix \( B \):
   \( B = Q * \Sigma * P^t \)

3. Eventually compute the eigenvectors:
   \( U = U' Q \), and \( V^t = (V' P)^t \)

Problem: Reduction to tridiagonal is using BLAS 2
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
What is The Polar Decomposition?

- The polar decomposition:

\[ A = U_p H, \ A \in \mathbb{R}^{m \times n} (m \geq n), \]

where \( U_p \) is an orthogonal matrix and \( H = \sqrt{A^T A} \) is a symmetric positive semidefinite matrix.

- The polar decomposition is a critical numerical algorithm for various applications, including aerospace computations, chemistry, factor analysis.
A Major Building Block Toward Important DLA Algorithms

The polar decomposition can be used as a pre-processing step toward solving:

- **the symmetric eigenvalue problem**: \( A = V \Lambda V^T, \ V = [V_1 \ V_2] \)
- **the singular value decomposition**: \( A = U \Sigma V^T \)

\[
A = U_p H = U_p (V \Sigma V^T) = (U_p V) \Sigma V^T = U \Sigma V^T
\]
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
QDWH Polar Decomposition Algorithm

\[ A = U_p H \]

where, \( U_p U_p^T = I_n \), \( H \) is symmetric positive semidefinite

- Backward stable algorithm for computing the polar decomposition
- Based on conventional computational kernels, i.e., Cholesky/QR factorizations (\( \leq 6 \) iterations for double precision) and GEMM
- The total flop count for QDWH depends on the condition number of the matrix \( \kappa \):

<table>
<thead>
<tr>
<th>( \kappa )</th>
<th>1</th>
<th>( 10^{16} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>flops</td>
<td>( (10 + \frac{2}{3})n^3 )</td>
<td>( 43n^3 )</td>
</tr>
</tbody>
</table>
The QDWH iteration is:

\[
X_0 = A/\alpha, \begin{bmatrix} \sqrt{c_k} X_k & I \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R, \quad X_{k+1} = \frac{b_k}{c_k} X_k + \frac{1}{\sqrt{c_k}} \left( a_k - \frac{b_k}{c_k} \right) Q_1 Q_2^T, \quad k \geq 0
\]

(1)

When, \( X_k \) becomes well-conditioned, it is possible to replace Equation 1 with a Cholesky-based implementation as follows:

\[
X_{k+1} = \frac{b_k}{c_k} X_k + \left( a_k - \frac{b_k}{c_k} \right) (X_k W_k^{-1}) W_k^{-\top}, \quad W_k = \text{chol}(Z_k), \quad Z_k = I + c_k X_k^\top X_k
\]

(2)
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
QDWH/ZOLO Polar Decomposition Algorithms

- **QDWH:**
  \[
  \begin{bmatrix}
  \sqrt{c_k}X_k \\
  I
  \end{bmatrix} =
  \begin{bmatrix}
  Q_1 \\
  Q_2
  \end{bmatrix} R,
  X_{k+1} = \frac{b_k}{c_k} X_k + \frac{1}{\sqrt{c_k}} \left( a_k - \frac{b_k}{c_k} \right) Q_1 Q_2^*.
  \]

- **ZOLO:**
  \[
  \begin{bmatrix}
  X_k \\
  \sqrt{c_{2j-1}} I
  \end{bmatrix} =
  \begin{bmatrix}
  Q_{j1} \\
  Q_{j2}
  \end{bmatrix} R_j,
  X_{k+1} = X_k + \sum_{j=1}^{r} \frac{a_j}{\sqrt{c_{2j-1}}} Q_{j1} Q_{j2}^*.
  \]

For ill-conditioned matrices, in double precision, QDWH converges after 6 successive iterations, while ZOLO converges after 2 successive iterations, each execute 8 independent embarrassingly parallel factorizations.
**ZOLO Arithmetic Complexity VS QDWH**

**Table 1:** Algorithmic complexity and memory footprint for various PD algorithms with $\kappa_2(A) = 10^{12}$.

<table>
<thead>
<tr>
<th></th>
<th>QDWH</th>
<th>Successive ZOLO</th>
<th>Independent ZOLO</th>
</tr>
</thead>
<tbody>
<tr>
<td># QR-based iterations</td>
<td>2</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td># Cholesky-based iterations</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Algorithmic complexity</td>
<td>$33n^3$</td>
<td>$100n^3$</td>
<td>$15n^3$</td>
</tr>
<tr>
<td>Memory footprint</td>
<td>$6n^2$</td>
<td>$6n^2$</td>
<td>$48n^2$</td>
</tr>
</tbody>
</table>
The Big Picture (Similar w/ SVD)
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
Performance Comparison on *Shaheen-2* (Polar-Decomposition)

Figure 1: QDWH versus ZOLO-PD.

(a) 200 nodes.  
(b) 400 nodes.  
(c) 800 nodes.
Performance Results: From PD To SVD on 800 nodes of Shaheen-2

(a) Polar Decomposition

(b) SVD solvers
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   - The QDWH-based Polar Decomposition
   - The ZOLO-based Polar Decomposition
   - Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
## Summary of what is done

### QDWH
- DPLASMA [Cluster 2019]
  - Distributed memory / No GPUs
- Chameleon [TPDS 2017]
  - Shared Memory / GPUs
- Distributed+GPUs ???

### ZOLO
- Chameleon: on-going
Performance Comparisons Using Well and Ill-Conditioned Matrices

![Graphs showing performance comparisons using well and ill-conditioned matrices across different node counts and matrix sizes.](image)
Performance Breakdown on # Nodes / Matrix Size N

![Diagram showing performance breakdown on different matrix sizes and node counts.](image)

- **DPOSV**
- **QR**
- **DGEMM**
- **Other**
- **PO-based**
- **QR-based**
- **Condest**

**Time (s)**

**ScaLAPACK**

18/50K, 72/70K, 288/100K, 1152/120K

**PaRSEC**

18/50K, 72/70K, 288/100K, 1152/120K
What is missing for an efficient ZOLO algorithm

- Is it possible to save some memory thanks to the task-based algorithm? (Avoid the 48 factor)
- Exploit the dynamic task-based computations to better balanced the replicated problems
- Efficient reduction step to merge the partial solutions together
- Will there be scheduling issues with the very large amount of tasks and the pipelining of the stages?
Outline

1. Classic solution to solve large SVD problems

2. Using the polar decomposition
   ▶ The QDWH-based Polar Decomposition
   ▶ The ZOLO-based Polar Decomposition
   ▶ Preliminary results with Scalapack [D. Sukkari’s PhD]

3. Task based algorithms

4. Alternative solutions to (partial-SVD)
Other solutions that can be used for (partial-)SVD

- Randomized SVD (cf Diodon project)
- Partial QR with column pivoting
  Pb with the norm computations and the pivoting
- Randomized QR with column pivoting
  Similar issue as before but can localize the pivoting in a smaller matrix than can be replicated to avoid communications.
- Truncated QR factorization algorithms
  Issue with the storage of the updates
- In previous solutions, the pivoting strategy can be replace by a rotation solution, that replaces the column pivoting by a matrix-matrix product.
Thank you 😊
for (k = 0; k < NT; k+)
    potrf( RW, A[k][k] );
for (n = k+1; n < NT; n++)
    trsm( READ, A[k][k], RW, A[k][n] );
for (m = k+1; m < NT; m++)
    syrk( READ, A[k][m], RW, A[m][m] );
for (n = m+1; n < NT; n++) {
    gemm( READ, A[k][m], READ, A[k][n],
          RW, A[m][n] );
# Leveraging QDWH-TB from Shmem to Distmem

<table>
<thead>
<tr>
<th>Year</th>
<th>Authors</th>
<th>Conference/Journal</th>
<th>Paper Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1877</td>
<td>Zolotarev</td>
<td></td>
<td>Best rational approximant for the scalar sign function.</td>
</tr>
<tr>
<td>1994</td>
<td>Higham and Papadimitriou</td>
<td><em>SIAM</em></td>
<td><em>matrix inversion QDWH, shared-memory systems</em>.</td>
</tr>
<tr>
<td>2010</td>
<td>Nakatsukasa et al.</td>
<td><em>SIAM</em></td>
<td><em>inverse-free QDWH, theoretical accuracy study</em>.</td>
</tr>
<tr>
<td>2013</td>
<td>Nakatsukasa and Higham</td>
<td><em>SIAM</em></td>
<td><em>QDWH-EIG, QDWH-SVD, theoretical accuracy study</em>.</td>
</tr>
<tr>
<td>2014</td>
<td>Nakatsukasa</td>
<td><em>SIAM</em></td>
<td><em>ZOLO-PD, ZOLO-SVD, ZOLO-EIG, theoretical accuracy study</em>.</td>
</tr>
<tr>
<td>2016</td>
<td>Sukkari, Ltaief and Keyes</td>
<td><em>TOMS</em></td>
<td><em>QDWH-SVD, block algorithm, shared-memory system equipped with multiple GPUs</em>.</td>
</tr>
<tr>
<td>2016</td>
<td>Sukkari, Ltaief and Keyes</td>
<td><em>Euro-Par</em></td>
<td><em>QDWH, QDWH-SVD, block algorithm, distributed-memory system</em>.</td>
</tr>
<tr>
<td>2017</td>
<td>Sukkari, Ltaief, Faverge and Keyes</td>
<td><em>TPDS</em></td>
<td><em>QDWH, task-based, shared-memory system equipped with multiple GPUs</em>.</td>
</tr>
</tbody>
</table>
Parametrized Task Graph: PaRSEC pseudo-code with Cholesky factorization (POTRF and TRSM)

**potrf** \((k)\)

// Execution space
\(k = 0 \ldots NT-1\)

// Parallel partitioning
:A\((k, k)\)

RW \(T \leftarrow (k == 0) ? A(k, k)\)
[U]
\(<- (k != 0) ? T \text{ syrk}(k -1, k)\) [U]

\(- \rightarrow T \text{ trsm}(k, k+1..NT-1)\) [U]

\(- \rightarrow A(k, k)\) [U]

**trsm** \((k, n)\)

// Execution space
\(k = 0 \ldots NT-2\)
\(n = k+1 \ldots NT-1\)

// Parallel partitioning
:A\((k, n)\)

READ \(T \leftarrow T \text{ potrf}(k)\)
[U]

RW \(C \leftarrow (k == 0) ? A(k, n)\)
\(<- (k != 0) ? C \text{ gemm}(k -1, n, k)\)

\(- \rightarrow A \text{ syrk}(k, n)\)
\(- \rightarrow A \text{ gemm}(k, n, n+1..NT-1)\)

\(- \rightarrow B \text{ gemm}(k, k+1..n-1, n)\)
\(- \rightarrow A(k, n)\)
**Parametrized Task Graph: PaRSEC pseudo-code with Cholesky factorization (SYRK and GEMM)**

```
**syrk** (k, m)
  // Execution space
  k = 0 .. NT-2
  m = k+1 .. NT-1
  // Parallel partitioning
  : A(m, m)

  READ A <- C trsm(k, m)

  RW T <- (k == 0) ? A(m, m) [U]
      <- (k != 0) ? T syrk(k-1, m) [U]
      -> (m == k+1) ? T potrf(m) [U]
      -> (m != k+1) ? T syrk(k+1, m) [U]

**gemm** (k, m, n)
  // Execution space
  k = 0 .. NT-3
  m = k+1 .. NT-1
  n = m+1 .. NT-1
  // Parallel partitioning
  : A(m, n)

  READ A <- C trsm(k, m)
  READ B <- C trsm(k, n)

  RW C <- (k == 0) ? A(m, n)
      <- (k != 0) ? C gemm(k-1, m, n)
      -> (m == k+1) ? C trsm(m, n)
      -> (m != k+1) ? C gemm(k+1, m, n)
```
(1) Task-based Design of the Matrix Two-Norm Estimation

Local GEMV $y_i = Ax$

AllReduce $y = \sum y_i$

Local GEMV $x_i = A^T y$

AllReduce $x = \sum x_i$

Loc. Norm

AllRed. $||y||$

Loc. Norm

AllRed. $||x||$

Stopping criteria

M. Faverge – Large Scale SVD

32/42
(2) Scalable Universal Matrix Multiplication Algorithm (SUMMA)

- SUMMA replaces standard broadcasts with pipelined rings of communication
- Already implemented in ScaLAPACK for distributed-memory GEMM operations
- Fine-grained computations expose a low-level control of communications, which provide more flexibility for scheduling of computational tasks and communications.
- For instance: overlapping, network congestion, communication load balancing, etc.
Performance Impact in TFlop/s on 288 nodes w/ SUMMA for Matrix-Matrix Multiplication
(3) Hierarchical QR Factorization Using Tree Reduction: Flat Tree \( Flat(0) \) (or Domino)

Long critical path 😞
High communication volume 😞
(3) Hierarchical QR Factorization Using Tree Reduction: 
\[ \text{Flat}(k) \]

Short critical path 😊
High communication volume 😞
(3) Hierarchical QR Factorization Using Tree Reduction: Greedy

Short critical path 😊
Low communication volume 😊
Low kernels’ arithmetic intensity 😞
(3) Hierarchical QR Factorization Using Tree Reduction: Mixing Greedy + flat

Short critical path 😊
Low communication volume 😊
High kernels’ arithmetic intensity 😊
Performance Impact in TFlop/s on 288 nodes w/ HQR for the QR Factorization

![Graph showing performance impact in TFlop/s on 288 nodes with HQR for the QR Factorization. The graph compares DPLASMA - HQR, ScaLAPACK, DPLASMA - Flat(k), and DPLASMA - Flat(0).]
(4) Composing Directed Acyclic Graphs
### (4) Composing Directed Acyclic Graphs

The image contains a directed acyclic graph with nodes and edges representing computational operations. The graph is labeled with various nodes and edges, indicating the sequence of operations, such as `POTRF`, `TRSM`, `GEMM`, `HERK`, `TRTRI`, and `LAUUM`. The operations are interconnected to depict the flow of computations.

The graph is distributed across multiple levels, with operations at different stages.

**Key Operations**:
- `POTRF`: Performs a Cholesky factorization of a symmetric positive definite matrix.
- `TRSM`: Solves a triangular system of equations.
- `GEMM`: Performs a general matrix multiplication.
- `HERK`: Performs a Hermitian rank-1 update.
- `TRTRI`: Performs an LU factorization of a triangular matrix.
- `LAUUM`: Computes the Cholesky factorization of a symmetric matrix.

The graph is complex, showing a hierarchical structure with operations at different levels and stages, indicating the flow and dependencies between computations.
Performance Impact in TFlop/s on 288 nodes w/ DAG Composition for Cholesky-based Linear Solvers

![Graph showing performance impact in TFlop/s on 288 nodes with DAG composition for Cholesky-based linear solvers. The graph compares theoretical peak, DPLASMA (POSV), DPLASMA (POTRF+TRSM+TRSM), and ScaLAPACK performance.]