Performance of high-order SVD approximation: reading the data twice is enough

Melven Röhrig-Zöllner, Jonas Thies and Achim Basermann
Institute for Software Technology, German Aerospace Center (DLR)

Categories of tensor decomposition methods

from the point of view of computing resources:. . .

1. data too large to process as a whole
   - "randomly" access part of the data
   - reconstruct approximation with some probability
2. data implicitly given by some high-dim. function with known low rank / smoothness
   - "black box" approximation, evaluate as few entries as possible
   - error bounds for special classes of functions
3. data large and sparse, feasible to access all entries
   - exploit (problem specific) sparsity
   - accurate up to a desired tolerance
4. data large and dense, feasible to access all entries
   - discussed here!
   - accurate up to a desired tolerance

Problem definition

Low-rank approximation in tensor-train format [Oseledets]

Given:

- large dense tensor $X \in \mathbf{R}^{n_1 \times n_2 \times \cdots \times n_d}$
- max. tensor-train rank r_{max}
- desired tolerance ϵ_{tol}

Calculate:

- tensor-train X_{TT} with:

$$\text{ranks}(X_{\text{TT}}) \leq r_{\text{max}} \quad \text{and} \quad \|X - X_{\text{TT}}\|_F \lesssim \epsilon_{\text{tol}}$$

Remarks:

- Focus on the tensor-train format; very similar approaches for some other formats
- Consider high-dimensional case ($d \gg 3$) and sufficiently small TT-ranks $r_1, \ldots r_{d-1}$

Roofline performance model

Consider 2 bottlenecks:

1. Peak performance: P_{max} [GFlop/s]
2. Memory bandwidth: b_s [GByte/s]

Analyze the algorithm:

1. Computations: n_{flops} [flop]
2. Data transfers: $V_{\text{read+write}}$ [byte]

$\Rightarrow$ Expected (ideal) runtime:

$$t = \max\left(\frac{n_{\text{flops}}}{P_{\text{max}}}, \frac{V_{\text{read+write}}}{b_s}\right) \text{ [s]}$$

Remark: growing memory gap P_{max}/b_s (e.g. $\sim$ 100 Flops per double on my CPU from 2017)

Standard TT-SVD algorithm

Algorithm

Input: Tensor X

$r_0 \leftarrow 1$

for $i = 1, \dots, d-1$ **do**

Reshape X to $\bar{n}_i \times (n_i r_{i-1})$

Calculate SVD: $USV^T = X$

Choose rank r_i

$T_i \leftarrow V^T_{1:r_i}$, reshape to $r_{i-1} \times n_i \times r_i$

$X \leftarrow U_{1:r_i} S_{1:r_i}$

end for

Reshape X to $(r_{d-1} \times n_d \times 1)$

Output: Tensor-train $(T_1, \dots, T_{d-1}, X)$

Standard TT-SVD algorithm

Algorithm

Input: Tensor X

$r_0 \leftarrow 1$

for $i = 1, \ldots, d-1$ **do**

Reshape X to $\bar{n}_i \times (n_i r_{i-1})$

Calculate SVD: $USV^T = X$

Choose rank r_i

$T_i \leftarrow V^T_{1:r_i}$, reshape to $r_{i-1} \times n_i \times r_i$

$X \leftarrow U_{1:r_i} S_{1:r_i}$

end for

Reshape X to $(r_{d-1} \times n_d \times 1)$

Output: Tensor-train $(T_1, \ldots, T_{d-1}, X)$

Observations

- Based on successive SVDs, reshapes and matrix-matrix multiplications (GEMM)
- Cheap operations are grayed out
- All large matrices are tall and skinny $\bar{n}_i := \prod_{j=i+1}^{d} n_i \gg n_i r_i$

→ Operations are likely memory-bound!

- Size of X ideally decreases in each step, not ensured in first steps for $r_i \ll r_{\max}$

Optimized TT-SVD algorithm

Algorithm

Input: Tensor X

Skip first $j-1$ iterations

Reshape X to $\bar{n}_j \times (n_1 \cdots n_j)$

for $i = j, \dots, d-1$ **do**

Calculate QR decomposition: $QR = X$

Calculate small SVD: $\bar{U}SV^T = R$

Choose rank r_i

$T_i \leftarrow V_{1:r_i}^T$, reshape to $r_{i-1} \times n_i \times r_i$

$X \leftarrow XV_{1:r_i}$, reshape to $\bar{n}_{i+1} \times (n_{i+1} r_i)$

end for

Recover $T_1, \dots, T_j$

Output: Tensor-train $(T_1, \dots, T_{d-1}, X)$

Optimized TT-SVD algorithm

Algorithm

Input: Tensor X

Skip first $j-1$ iterations

Reshape X to $\bar{n}_j \times (n_1 \cdots n_j)$

for $i = j, \dots, d-1$ **do**

Calculate QR decomposition: $QR = X$

Calculate small SVD: $\bar{U}SV^T = R$

Choose rank r_i

$T_i \leftarrow V^T_{1:r_i}$, reshape to $r_{i-1} \times n_i \times r_i$

$X \leftarrow XV_{1:r_i}$, reshape to $\bar{n}_{i+1} \times (n_{i+1} r_i)$

end for

Recover $T_1, \dots, T_j$

Output: Tensor-train $(T_1, \dots, T_{d-1}, X)$

Remarks

- Skip iterations that don't reduce the size of X
- Replaced costly SVD by tall-skinny QR
- Never use Q → Q-less TSQR
- Fused reshape and tall-skinny matrix-matrix multiplication ("TSMM"): $X \leftarrow XV_{1:r_i}$, reshape to ...

→ Reads the input data twice (1st iteration): (once for $QR = X$, once for $X \leftarrow XV_{1:r_1}$)

Performance analysis (1)

Building blocks

Q-less TSQR:
$(X \in \mathbf{R}^{n \times m})$

- $V_{\text{read}} = nm$
- $n_{\text{flops}} \approx 2nm^2$

TSMM+reshape:
$(X \leftarrow XM, \; M \in \mathbf{R}^{m \times k})$

- $V_{\text{read+write}} = n(m+k)$
- $n_{\text{flops}} = 2nmk$

$\Rightarrow$ One step of the TT-SVD iteration:

- $V_{\text{read+write}} = n(2m+k)$
- $n_{\text{flops}} \approx 2nm(m+k)$

Performance analysis (1)

Building blocks

Q-less TSQR:
($X \in \mathbf{R}^{n \times m}$)

- $V_{\text{read}} = nm$
- $n_{\text{flops}} \approx 2nm^2$

TSMM+reshape:
($X \leftarrow XM,\ M \in \mathbf{R}^{m \times k}$)

- $V_{\text{read+write}} = n(m+k)$
- $n_{\text{flops}} = 2nmk$

$\Rightarrow$ One step of the TT-SVD iteration:

- $V_{\text{read+write}} = n(2m+k)$
- $n_{\text{flops}} \approx 2nm(m+k)$

Complete TT-SVD algorithm

Assume size reduction factor $f < 1$ in each step with $k/m \leq f$.
$\Rightarrow$ upper bound from the geometric series:

- $V_{\text{read+write}} \leq \frac{2N}{1-f} + \frac{fN}{1-f}$
- $n_{\text{flops}} \lesssim 2Nr_{\max}\left(\frac{1}{f} + \frac{2}{1-f}\right) + O(r_{\max}^3)$

with $N := \prod_{i=1}^{d} n_i$.

Performance analysis (2)

Interpretation

Try to influence f by combining (splitting) dimensions!
Suitable choices for 2^d tensors:

- $f = 1/16$ (low rank): $V_{\text{read+write}} \lesssim 2.2N$ and $n_{\text{flops}} \lesssim 36Nr_{\text{max}}$
- $f = 1/2$ (medium rank): $V_{\text{read+write}} \lesssim 5N$ and $n_{\text{flops}} \lesssim 12Nr_{\text{max}}$

Performance analysis (2)

Interpretation

Try to influence f by combining (splitting) dimensions!
Suitable choices for 2^d tensors:

- $f = 1/16$ (low rank): $V_{\text{read+write}} \lesssim 2.2N$ and $n_{\text{flops}} \lesssim 36Nr_{\text{max}}$
- $f = 1/2$ (medium rank): $V_{\text{read+write}} \lesssim 5N$ and $n_{\text{flops}} \lesssim 12Nr_{\text{max}}$

Comparison with measurements (using CPU performance counters)

Decompose a double-precision 2^{30} tensor (8GB)

	r_{max}	operations (est.) [GFlop]	data transfers (est.) [GByte]
$f = 1/2$	1	14 (13)	43 (43)
$f = 1/16$	1	41 (39)	21 (19)
$f = 1/2$	31	417 (399)	43 (43)

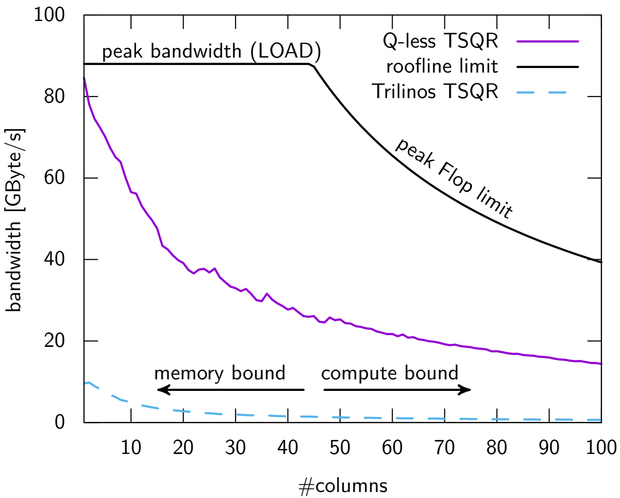
(in practice, as n_i and r_i are integers, only some discrete values for f possible)

Performance of building blocks: Q-less TSQR

- $(25 \cdot 10^6) \times m$ matrix, $m = 1, \ldots, m$ (double-precision)
- Data size: 200MB,. . . ,20GB
- 14-core Intel Skylake Gold 6132
- Bandwidth: $b_w := V_{\text{read}}/t$

→ Peak bandwidth for small m, $\sim 1/3$ peak Flop/s for larger m

- Significantly faster than other TSQR implementations I tried. . .
- BUT the Trilinos TSQR here calculates Q explicitly!

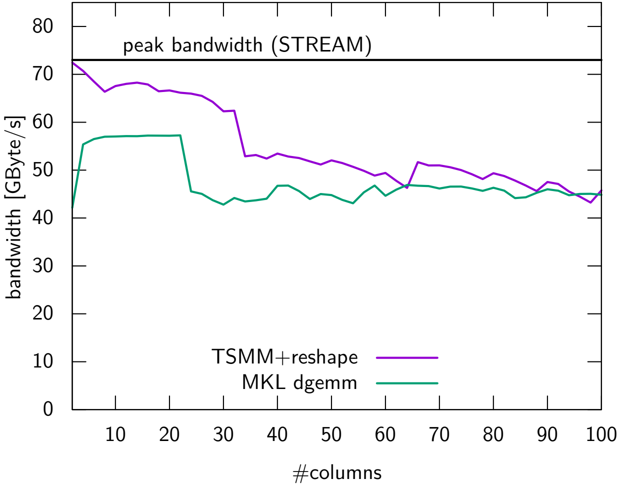


Performance of building blocks: fused tall-skinny GEMM+reshape

- For $X \in \mathbf{R}^{n \times 2m}$, $M \in \mathbf{R}^{2m \times m}$, $X \leftarrow XM$, reshape to $(n/2, 2m)$, $n = 25 \cdot 10^7$, $m = 1, \ldots, 50$
- Data size: 200MB,. . . ,20GB
- Bandwidth: $b_w := V_{\text{read+write}}/t$
- 14-core Intel Skylake Gold 6132

→ High bandwidth with fused reshape

- Similar performance as MKL, exploit known memory layout



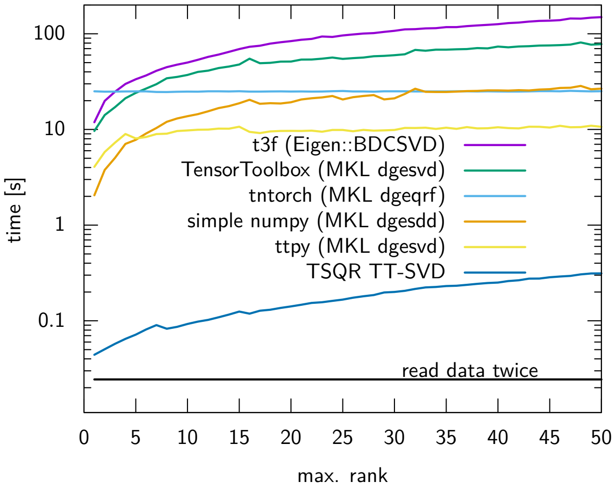
TT-SVD runtime: different implementations

- Decompose random 2^{27} tensor, $r_{\max} = 1, \ldots, 50$ (double precision)
- Data size: 1GB
- 14-core Intel Skylake Gold 6132

→ "Almost" optimal runtime

→ Existing software: >50x slower

- tntorch first constructs a full-rank TT, then truncates it.
- remark: my RNG is slower than the TT-SVD for $r_{\max} \lesssim 20$.

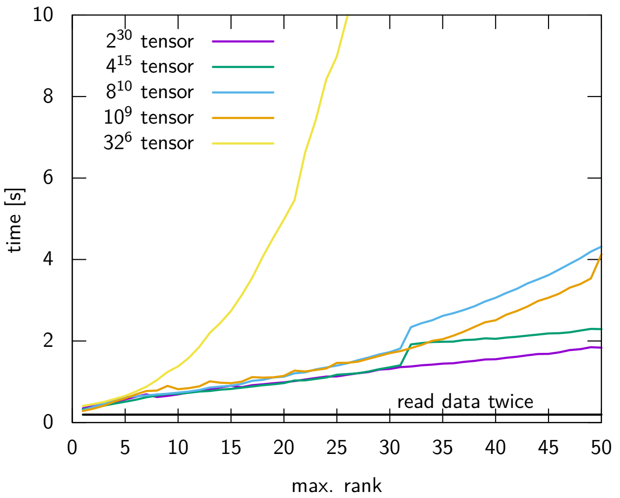


TT-SVD runtime: different tensor dimensions

- Decompose large random tensor, $r_{\max} = 1, \ldots, 50$ (double precision)
- Data size: $\sim$ 8GB
- Combine first dimensions only if beneficial
- 14-core Intel Skylake Gold 6132

→ Calculation more costly with fewer small dimensions!

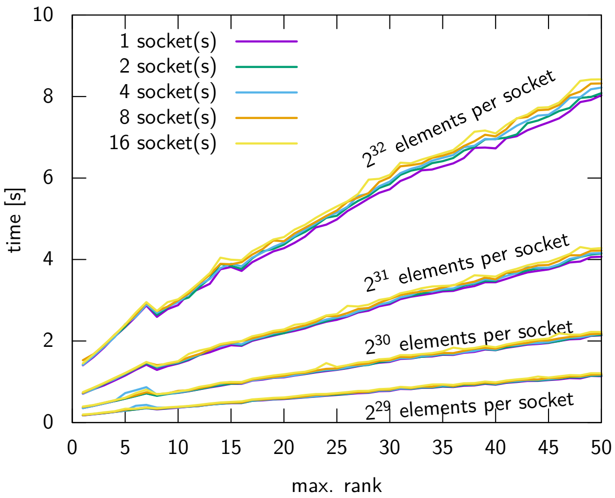
- Jumps in runtime: switch from e.g. $8^8 \times 8^2$ to $8^7 \times 8^3$ in the first tsqr step



TT-SVD runtime: distributed memory (MPI)

- Decompose random 2^d tensor, $d = 29, \ldots, 36$, $r_{\max} = 1, \ldots, 50$ (double precision)
- Data size: 4GB, ..., 550GB
- Distributed parallel (user-defined MPI reduction for TSQR)
- Up to 4 nodes with 4x14-core Intel Skylake Gold 6132

$\rightarrow$ Scales well onto multiple nodes



Conclusion

- Goal: compute a low-rank approximation of a large dense high-dimensional tensor ($d \geq 10$)
- Runtime lower bounds for the TT-SVD algorithm:
  - low rank: $\sim$ access data twice
  - medium rank: $O(r_{\max} \cdot N)$

  $\rightarrow$ Similar for some other tensor decompositions
- Almost optimal implementation:
  $\sim 50\times$ faster than others
  $\rightarrow$ Difficult to map tensor algorithms to efficient building blocks
- Future work: other tensor formats, performance of randomized decompositions (they can avoid this lower bound!), speed up algorithms from data analysis using TT-SVD

Literature

- Röhrig-Zöllner et.al.: "Performance of low-rank approximations in tensor train format (TT-SVD) for large dense tensors", submitted to SISC, arXiv:2102.00104, 2021
- Oseledets: "Tensor-Train Decomposition", SISC, 2011
- Demmel et.al.: "Communication-optimal Parallel and Sequential QR and LU Factorizations", SISC 2012
- Psarras et.al.: "The Linear Algebra Mapping Problem", preprint, arXiv:911.09421, 2019
- Demmel: "Communication avoiding algorithms", ENLA Seminar, `https://www.youtube.com/watch?v=42f0n0w2Nlg`
- Williams et.al.: "Roofline: An Insightful Visual Performance Model for Multicore Architectures", Comm. of the ACM, 2009

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm
  (sequential/cache optimized, parallel/comm. optimized)

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm
  (sequential/cache optimized, parallel/comm. optimized)

Householder QR algorithm

1. New block + previous block (already triangular)

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ * & * & * & * & * \end{pmatrix}$$

$$\begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{pmatrix}$$

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm (sequential/cache optimized, parallel/comm. optimized)

Householder QR algorithm

1. New block + previous block (already triangular)
2. Calculate reflection vector

$$\begin{pmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ * & * & * & * & * \\ * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{pmatrix}$$

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm
  (sequential/cache optimized, parallel/comm. optimized)

Householder QR algorithm

1. New block + previous block (already triangular)
2. Calculate reflection vector and apply reflection

$$\begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{pmatrix}$$

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm
  (sequential/cache optimized, parallel/comm. optimized)

Householder QR algorithm

1. New block + previous block (already triangular)
2. Calculate reflection vector and apply reflection
3. Repeat

$$\begin{pmatrix} * & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & * & * & * & * \\ 0 & 0 & * & * & * \\ 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * \end{pmatrix}$$

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm
  (sequential/cache optimized, parallel/comm. optimized)

Householder QR algorithm

1. New block + previous block (already triangular)
2. Calculate reflection vector and apply reflection
3. Repeat

$$
\begin{pmatrix}
* & * & * & * & * \\
0 & * & * & * & * \\
0 & 0 & * & * & * \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & * & * & * \\
0 & 0 & * & * & * \\
0 & 0 & * & * & * \\
0 & 0 & * & * & * \\
0 & 0 & 0 & * & * \\
0 & 0 & 0 & 0 & *
\end{pmatrix}
$$

Implementation details: TSQR SIMD optimization

Idea

- Combine a full and a triangular block
- Used for all reductions in the TSQR algorithm
  (sequential/cache optimized, parallel/comm. optimized)

Householder QR algorithm

1. New block + previous block (already triangular)
2. Calculate reflection vector and apply reflection
3. Repeat

$\rightarrow$ Works with vectors of fixed-size $b \cdot n_{\text{simd}}$ (multiple of SIMD width)

$$
b \cdot n_{\text{simd}} \left\{ \vphantom{\begin{matrix} 0\\0\\\vdots\\0\\0\\0\\0 \end{matrix}} \right.
\begin{pmatrix}
* & * & * & * & * \\
0 & * & * & * & * \\
0 & 0 & * & * & * \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & * & * & * \\
0 & 0 & * & * & * \\
0 & 0 & * & * & * \\
0 & 0 & * & * & * \\
0 & 0 & 0 & * & * \\
0 & 0 & 0 & 0 & *
\end{pmatrix}
$$

Implementation details: "rank preserving" TSQR

Goal

- Avoid data dependencies!
  $\rightarrow$ crucial for high performance,
  (the CPU is a big pipeline)
  $\rightarrow$ **No pivoting** $\lightning$
- Still handle rank-deficient blocks

Implementation details: "rank preserving" TSQR

Goal

- Avoid data dependencies!
  $\rightarrow$ crucial for high performance,
  (the CPU is a big pipeline)
  $\rightarrow$ **No pivoting** ↯
- Still handle rank-deficient blocks

Algorithm

Input: Input column $u \in \mathbf{R}^k$,
Smallest positive FP number ϵ_{fp} ($\approx 10^{-300}$)
Output: Householder reflection $(I - vv^T)$
with $\|v\|_2 = \sqrt{2}$

1: $t \leftarrow \|u\|_2^2 + \epsilon_{\text{fp}}$
2: $\alpha \leftarrow -\text{sign}(u_1)\sqrt{t + \epsilon_{\text{fp}}}$
3: $t \leftarrow t - \alpha u_1$
4: $u_1 \leftarrow u_1 - \alpha$
5: $\beta \leftarrow 1/\sqrt{t}$
6: $v \leftarrow \beta u$

Implementation details: "rank preserving" TSQR

Goal

- Avoid data dependencies!
  $\rightarrow$ crucial for high performance,
  (the CPU is a big pipeline)
  $\rightarrow$ **No pivoting** ↯
- Still handle rank-deficient blocks

Adjusted Householder reflection

- Add smallest representable number ϵ_{fp}
- Prevents break-down (no division by zero)
- Introduces an error of order $\sqrt{\epsilon_{\text{fp}}}$ if $u = 0$
- Ensures a valid reflection ($\|v\|_2 = \sqrt{2}$)

Algorithm

Input: Input column $u \in \mathbf{R}^k$,
Smallest positive FP number ϵ_{fp} ($\approx 10^{-300}$)
Output: Householder reflection $(I - vv^T)$
with $\|v\|_2 = \sqrt{2}$

$t \leftarrow \|u\|_2^2 + \epsilon_{\text{fp}}$
$\alpha \leftarrow -\text{sign}(u_1)\sqrt{t + \epsilon_{\text{fp}}}$
$t \leftarrow t - \alpha u_1$
$u_1 \leftarrow u_1 - \alpha$
$\beta \leftarrow 1/\sqrt{t}$
$v \leftarrow \beta u$