Vít Břichňáč; Jakub Šístek

Performance of parallel QR factorization methods on the NVIDIA Grace CPU Superchip

In: Jan Chleboun and Jan Papež and Karel Segeth and Jakub Šístek and Tomáš Vejchodský (eds.): Programs and Algorithms of Numerical Mathematics, Proceedings of Seminar. Hejnice, June 23-28, 2024. Institute of Mathematics, Czech Academy of Sciences, Prague, 2025. pp. 29–40.

Persistent URL: http://dml.cz/dmlcz/703227

Terms of use:

Institute of Mathematics of the Czech Academy of Sciences provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This document has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* http://dml.cz

Programs and Algorithms of Numerical Mathematics 22
J. Chleboun, J. Papež, K. Segeth, J. Šístek, T. Vejchodský (Eds.)
Institute of Mathematics CAS, Prague 2025

PERFORMANCE OF PARALLEL QR FACTORIZATION METHODS ON THE NVIDIA GRACE CPU SUPERCHIP

Vít Břichňáč[1], Jakub Šístek[1,2]

[1] Czech Technical University in Prague
Jugoslávských partyzánů 1580/3, 160 00 Prague 6 - Dejvice, Czech Republic
brichvit@cvut.cz
[2] Institute of Mathematics of the Czech Academy of Sciences
Žitná 25, 115 67 Prague 1 - Nové Město, Czech Republic
sistek@math.cas.cz

Abstract: This article studies several algorithms for QR factorization based on hierarchical Householder reflectors organized into elimination trees, which are particularly suited for tall-and-skinny matrices and allow parallelization. We examine the effect of various parameters on the performance of the tree-based algorithms. The work is accompanied with a custom implementation that utilizes a task-based runtime system (OpenMP or StarPU). The same algorithm is implemented in the PLASMA library. The performance evaluation is done on the recent NVIDIA Grace CPU Superchip.

Keywords: QR factorization, task-based programming, NVIDIA Grace CPU
MSC: 65F05

1. Introduction

The need for computing the QR factorization of dense matrices with substantially more rows than columns (so-called tall-and-skinny matrices) arises in a number of applications, for example, when solving overdetermined systems of linear equations by the least-squares method or as a preprocessing step for the SVD algorithm used in reduced order modeling.

Modern algorithms for computing the QR factorization of a matrix using orthogonal triangularization by Householder reflectors split the matrix into blocks and then perform operations on those blocks. Importantly, the parallel TSQR and CAQR algorithms of [7] opened the way to parallelizing the panel factorization and hence for deriving parallel algorithms for tall-and-skinny matrices. These algorithms are implemented for example in the ScaLAPACK[1] [3] and PLASMA[2] [5] libraries. Other recent approaches include numerically stable variants of triangular orthogonalization using Cholesky QR [11, 12] or randomized QR factorization methods, see, e.g., [13].

DOI: 10.21136/panm.2024.03
[1] http://www.netlib.org/scalapack
[2] https://icl.utk.edu/plasma

The algorithms work with several parameters (e.g., the block size, inner block size, etc.) that do not influence the result but have an impact on the computation time [4, 10]. In our paper [6], we present a new version of the algorithm for QR factorization based on tasks implemented in the OpenMP version of PLASMA [9] and perform a study of the effect of the main algorithmic parameters on performance on several multicore CPU architectures by Intel, AMD, and Arm. In light of [7], the algorithm can be seen as a combination of the parallel and sequential versions of the Communication-avoiding QR (CAQR) algorithm, with the latter performed on the leaves of the tree arising from the former.

The main purpose of the present article is to complement the experiments from [6] with performance measurements on the NVIDIA Grace CPU Superchip, another recent multicore chip based on the Arm architecture. The reader is referred to [6] for a more detailed description of the algorithm.

2. QR factorization

QR factorization is a matrix decomposition of a matrix $A \in \mathbb{R}^{m,n}$ into a product QR, where $Q \in \mathbb{R}^{m,m}$ is an orthogonal matrix and $R^{m,n}$ is an upper trapezoidal matrix. Many different methods may be used for computing the QR factorization of a matrix, but for developing parallel algorithms for QR computation, the Householder reflector method is of particular interest.

It works by applying a series of orthogonal transformations $Q_1, Q_2, \ldots, Q_k$ for $k = \min(m, n)$ on an arbitrary matrix $A \in \mathbb{R}^{m,n}$, where each of the transformations Q_i:

- zeros out the vector $A_{(i+1):m,i}$ using the entry $A_{i,i}$ by a reflection in a subspace corresponding to the last $m - i + 1$ rows of A, and
- functions as the identity transformation in the subspace corresponding to the first $m - 1$ rows.

As a result, the matrix $R = Q_k Q_{k-1} \ldots Q_1 A$ is upper triangular, and the QR decomposition of A can be formed as $A = QR$, where $Q = Q_1^T Q_2^T \ldots Q_k^T$.

3. Elimination schemes

3.1. Column block Householder reflector algorithm

To promote BLAS Level 3 operations in the application of the Householder reflectors, matrix columns can be grouped into column blocks as in the LAPACK library[3] [1]. This column-blocking also opens a way to parallelize the algorithm, since each block column of the updated matrix can be updated independently. In particular, the algorithm performs the following steps for each column block:

1. Factorize the column block into upper triangular form using a **block Householder reflector**.
2. Apply the calculated reflector to subsequent column blocks (potentially in parallel).

[3] http://www.netlib.org/lapack

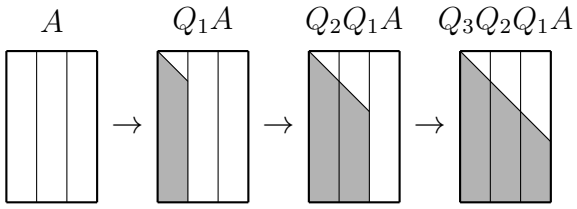

Figure 1: Example of the column block algorithm for a matrix with 3 column blocks.

The algorithm is visualized in Fig. 1.

In LAPACK, the routine for factorization using block Householder reflectors is named **GEQRT**. The routine that applies the calculated factor Q on an arbitrary matrix of appropriate size is labeled **GEMQRT**. These two routines will be referred to as **general QR kernels** in the rest of this article.

If a multithreaded implementation of the BLAS library is employed, parallelism is exploited in the second step of the algorithm. This approach, however, only offers enough parallelism if the matrix has a sufficient number of block columns, as only the update in Step 2 can be parallelized. Hence, for matrices $A \in \mathbb{R}^{m,n}$ with $m >> n$, this algorithm exploits parallelism insufficiently and offers subpar performance.

3.2. TS kernels & TS flat tree elimination scheme

In order to develop a parallel algorithm with good performance for tall-and-skinny matrices, it is necessary to split the matrix into row blocks as well as column blocks.

For a blocked matrix $A = \begin{pmatrix} A_1^T & A_2^T \end{pmatrix}^T$, where the block A_1 has a QR factorization $A_1 = Q_1 R_1$ and the matrix $\begin{pmatrix} R_1^T & A_2^T \end{pmatrix}^T$ has a QR decomposition $\widehat{Q}R$, it holds (see [7]) that

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = \begin{pmatrix} Q_1 R_1 \\ A_2 \end{pmatrix} = \begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} R_1 \\ A_2 \end{pmatrix} = \underbrace{\begin{pmatrix} Q_1 & 0 \\ 0 & I \end{pmatrix} \widehat{Q}}_{\bar{Q}} R = \bar{Q}R.$$

Since $\bar{Q}$ is a product of two orthogonal matrices, it is orthogonal as well. As a result, QR factorizations of the matrices A and $\begin{pmatrix} R_1^T & A_2^T \end{pmatrix}^T$ have the same factor R.

Consequently, we can calculate the QR factorization of A by factorizing its block A_1 using GEQRT (so that the matrix A_1 gets replaced with R_1) followed by factorizing the triangle-on-top-of-square matrix $\begin{pmatrix} R_1^T & A_2^T \end{pmatrix}^T$. The factorization and subsequent Q application of triangle-on-top-of square matrices is performed using the so-called **TS kernels**:

TSQRT performs factorization of a triangle-on-top-of-square matrix
TSMQR applies the transformation from **TSQRT** on an arbitrary block matrix made up of two row/column blocks

A scheme of these functions is visualized in Fig. 2.

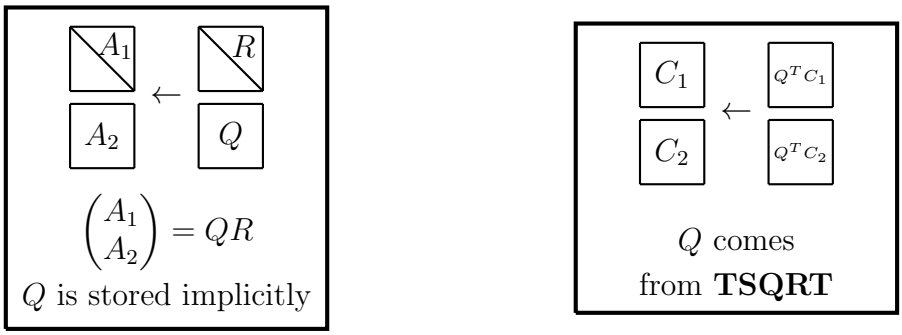

Figure 2: Scheme of the **TSQRT** kernel (left) and of the **TSMQR** kernel (right) for parameter values `SIDE='R'` and `TRANS='T'`.

Expanding on the observations made above, we can calculate the QR factorization of A using the **TS flat tree** elimination scheme:

1. Factorize the diagonal block using general QR kernels
2. Use it to eliminate the blocks below the main diagonal with TS kernels.

Similarly to the column block algorithm, only the Q application kernels can be parallelized in this procedure. This algorithm is called sequential CAQR in [7].

3.3. TT kernels & TT binary tree elimination scheme

We now further examine the QR factorization of a blocked matrix $A = \begin{pmatrix} A_1^T & A_2^T \end{pmatrix}^T$. Let $A_1 = Q_1 R_1$ and $A_2 = Q_2 R_2$ be the QR factorizations of A_1 and A_2, in their respective order. Let then $\widehat{Q}R$ denote the QR factorization of the blocked matrix $\begin{pmatrix} R_1^T & R_2^T \end{pmatrix}^T$. Then (cf. [7]):

$$A = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = \begin{pmatrix} Q_1 R_1 \\ Q_2 R_2 \end{pmatrix} = \begin{pmatrix} Q_1 & 1 \\ 0 & Q_2 \end{pmatrix} \begin{pmatrix} R_1 \\ R_2 \end{pmatrix} = \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \widehat{Q} R.$$

Instead of using the GEQRT kernel on the upper block followed by the TSQRT kernel to factorize the blocked matrix $A = \begin{pmatrix} A_1^T & A_2^T \end{pmatrix}^T$, we could first factorize both blocks using the GEQRT kernel (so that the upper triangular parts of the blocks get replaced with R_1 and R_2) and then factorize the obtained triangle-on-top-of-triangle matrix. The last step is done using the **TTQRT factorization kernel**, whose scheme in visualized in Fig. 3.

Based on the procedure presented above, we can define the **TT binary tree scheme** for decomposing a general blocked matrix A:

1. Factorize all blocks on & below the main diagonal using `GEQRT`.
2. Eliminate blocks below the (block) diagonal using `TTQRT` in a binary tree fashion.

The application kernels TTMQR can be parallelized, but now the factorization kernels in both steps can be performed in parallel, too (provided that they occur on the same level of the binary tree). This approach is called parallel CAQR in [7].

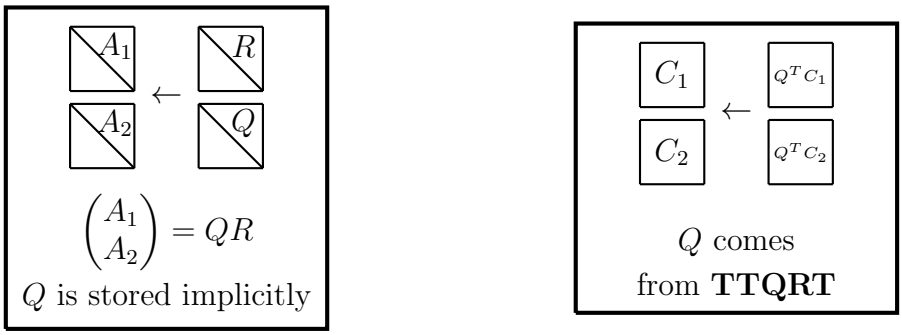

Figure 3: Scheme of the **TTQRT** kernel (left) and of the **TTMQR** kernel (right) for parameter values `SIDE='R'` and `TRANS='T'`.

3.4. Superblock-based elimination schemes

By comparing the TS flat tree and the TT binary tree elimination schemes, we can see that:

- The TS flat tree scheme requires fewer kernel calls with only the application kernels being parallelizable.
- The TT binary tree scheme requires more kernel calls with both the factorization and Q application kernels being parallelizable.

In this respect, the two schemes can be seen as block versions of the Householder reflector and the Givens rotation methods, respectively.

To balance out the effects of the two schemes, we may divide each block column into **superblocks**, where all superblocks in each column contain the same number of blocks (with the possible exception of the last superblock in a column). In other words, each superblock is composed of a fixed number of subsequent blocks. This number of blocks is called the **superblock size** b. We then factorize each column block of the matrix in the following manner:

1. Eliminate all blocks in an individual superblock using the TS flat tree scheme.
2. Eliminate first blocks of all superblocks in this column block using the TT binary tree scheme.

To select the superblock size b, we may utilize the formula [6]

$$b = \frac{m_t(n_t^2/2 + n_t/2)}{\gamma p}, \tag{1}$$

where

- m_t is the number of block rows,
- n_t is the number of block columns,
- p is the number of available threads,
- γ is a scaling factor (the default selection is $\gamma = 4$ as in the PLASMA library).

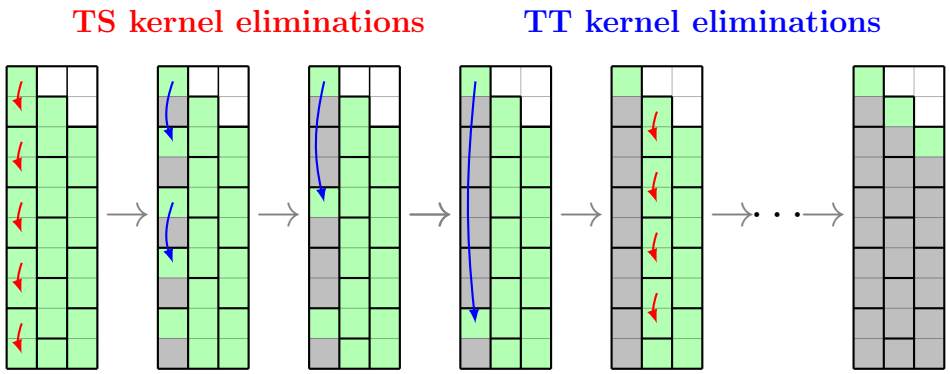

Figure 4: Example of the binary tree elimination scheme for 10 row blocks and 3 column blocks.

Formula (1) takes into account both the shape of the matrix and the number of CPU cores available for parallelization. As such, we obtain elimination schemes similar to the TT binary tree scheme for tall-and-skinny matrices, while for square-like matrices, we obtain elimination schemes very similar to the TS flat tree scheme.

This parameterized scheme is called the **superblock binary tree** scheme, see Fig. 4 for an example on a matrix with 10×3 blocks. By selecting a different scheme for eliminating the first blocks of each superblock in step 2, we may create different superblock-based schemes (other examples include the **superblock greedy** and **superblock Fibonacci** schemes [10], which are later user in Fig. 9). This idea can be seen as composing a hierarchical elimination tree [8] with different elimination trees on the top level and flat trees on the leaves (bottom level within superblocks).

4. Task-based runtime systems

The data dependencies between individual kernels can be represented by a **directed acyclic graph** (DAG), see an example in Fig. 5. From the DAG, we can see that the amount of available parallelism varies throughout the computation. As a result, **dynamic scheduling** is a powerful approach to implement a parallel TS flat tree scheme as well as the other schemes presented.

To ease the implementation, we use a **task-based runtime system**. These are systems that let us split the code into sections called **tasks**, and then execute the tasks in parallel while making sure that the data dependencies of the tasks are satisfied.

The runtime systems used in the implementation are **OpenMP** and **StarPU**. OpenMP[4] is a widely used standard for shared memory multicore programming, while StarPU [2] is a library created mainly with the intention of being used in heterogeneous systems (systems with multiple types of computing units), mainly targeting GPU accelerators.

[4] https://www.openmp.org/

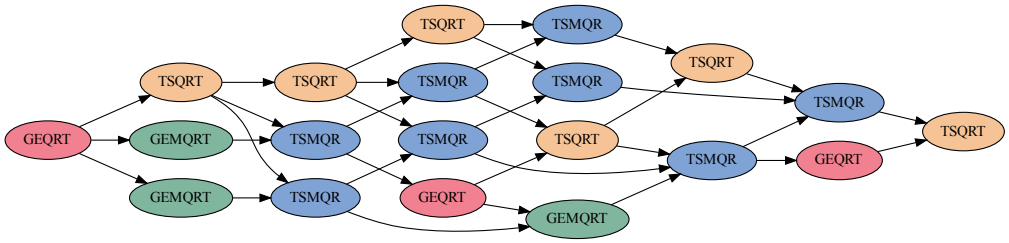

Figure 5: An example of the directed acyclic graph (DAG) of data dependencies. The TS flat tree scheme was used on a matrix with 4 row blocks and 3 column blocks.

5. Results

In this section, we evaluate the effect of different parameters on performance of the algorithm. We also compare the performance with the Arm Performance Libraries (ArmPL).

In each of the following figures, every point represents the best performance out of five consecutive runs. All experiments were performed using $m \times n$ matrices of sizes $m = 90000$ and $n \in \{250, 500, \ldots, 2750, 3000, 4000, \ldots, 15000\}$.

The performance was evaluated at the IT4Innovations National Supercomputing Center on a node with the following specifications:

- CPU: 1× NVIDIA Grace CPU Superchip
- CPU architecture: Arm64
- CPU cores: 144
- Base CPU frequency: 3.1 GHz
- All-code SIMD frequency: 3.0 GHz
- Instruction set extensions: Scalable Vector Extension 2 (SVE2)

The following library versions were used during the evaluation process:

- GCC 11.3.0
- Hwloc 2.7.1
- StarPU 1.3.10
- Arm Performance Libraries 22.1
- PLASMA 22.9.29

5.1. Block size comparison

In this section, we compare the effect of varying block sizes (nb) on the computational performance. In PLASMA, the matrix is first copied from column-major format to the tile layout before the QR factorization starts. After the end, the matrix

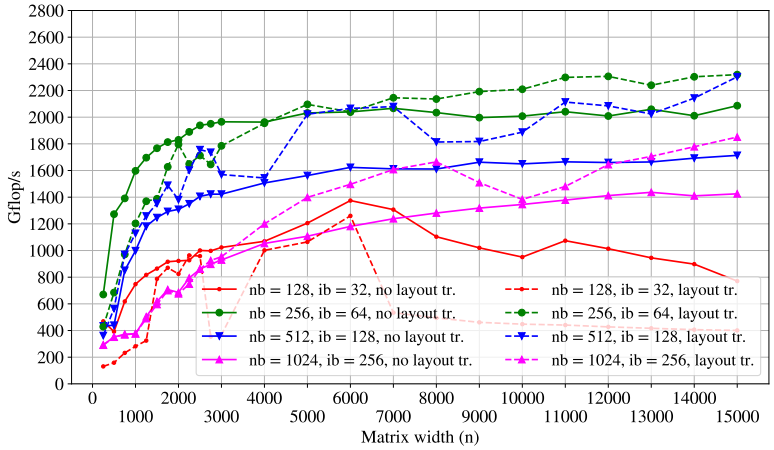

Figure 6: Performance for different block sizes (nb). SuperblockGreedy scheme, inner block size (ib) equal to $nb/4$, and $\gamma = 4$.

is copied back to the column-major format. We refer to these pre-/post-processing steps as *layout translation*. While the cost of the layout translation becomes outweighed by the faster processing of the matrix in the tile layout for wider matrices, we show in [6] that for very skinny matrices, it can be considerably faster to avoid the layout translation. Hence, for each tested block size in this section, we consider two variants – with and without layout translation.

We can see from Fig. 6 that the block size of 256 offers the best performance on the NVIDIA Grace node. Layout translation can boost the performance for wider matrices, while it can hinder the performance for skinnier matrices.

5.2. Inner block size comparison

In this section, we take a look at the effects of different inner block (ib) size values. The square blocks of size nb are divided into smaller block columns to perform the block-local operations by column-block oriented functions. Hence, the inner block size ib (the number of columns within these inner blocks) is always less than or equal to the selected block size; details can be found again in [6].

As can be seen from Fig. 7, the inner block size choice of 64 is best for the examined compute node.

5.3. Elimination schemes comparison

In this section, we compare the performance of the different elimination schemes presented earlier. We also include a performance curve for the Arm Performance Libraries (ArmPL) for comparison. The results are shown in Fig. 8.

In accordance with the observations in Section 3, the TsFlatTree scheme delivers a better performance for wider matrices, but it gets outperformed by the TtGreedy scheme for skinnier matrices. The SuperblockGreedy scheme combines the advan-

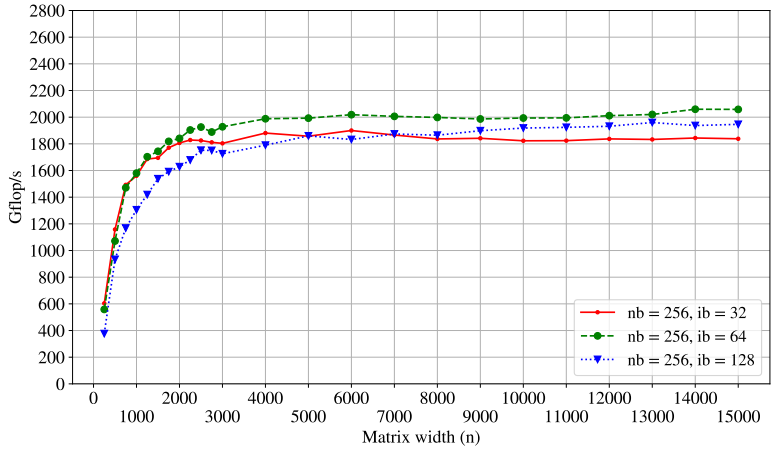

Figure 7: Performance for different inner block sizes. Tile size $nb = 256$, Superblock-Greedy scheme, $\gamma = 4$, and layout translation disabled.

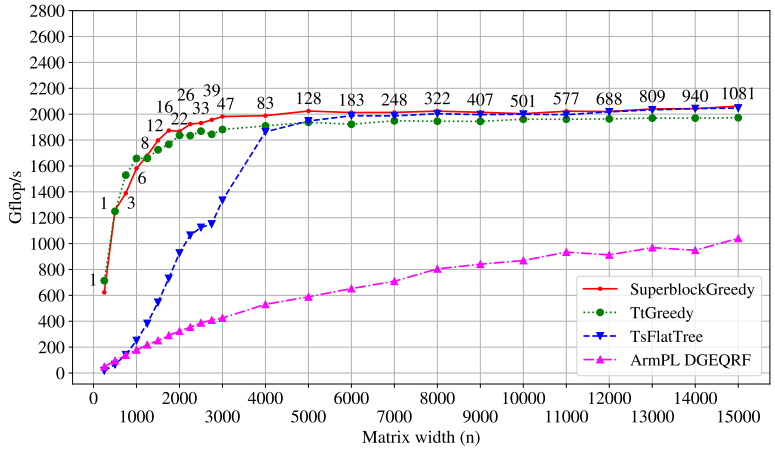

Figure 8: Performance of different elimination schemes. Block size $nb = 256$, inner blocks of size $ib = 64$, layout translation disabled, and $\gamma = 4$ (for the Superblock-Greedy elimination scheme). The numbers in the plot denote the used superblock sizes.

tages of both schemes to provide a good performance for both skinny and wide matrices. Interestingly, the performance of the TtGreedy scheme is only marginally lower on this architecture. All the schemes outperform the ArmPL implementation for matrices with more than 500 columns, while the latter slightly outperforms the TsGreedy scheme for skinnier matrices.

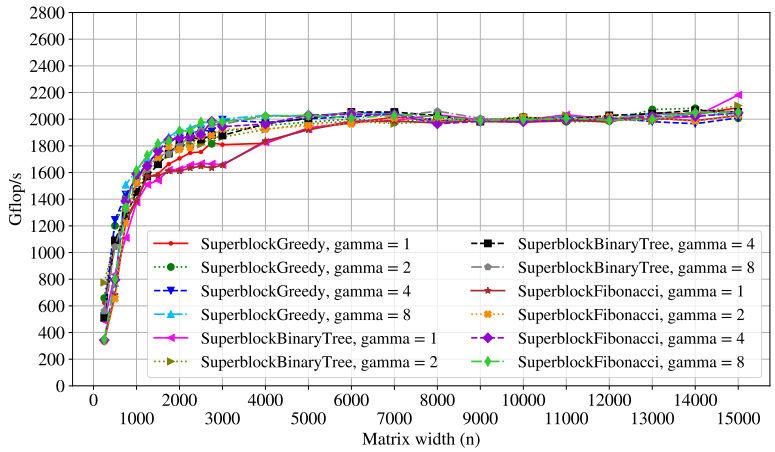

Figure 9: Performance of various superblock size factors (γ values). Block size $nb = 256$, inner block size $ib = 64$, and layout translation disabled.

5.4. Superblock size factor comparison

Next, we visualize the effects of different choices of the values for the γ parameter presented in Section 3.4. We compare four different values of $\gamma \in \{1, 2, 4, 8\}$ for three different superblock-based elimination schemes (SuperblockGreedy, SuperblockBinaryTree and SuperblockFibonacci).

In Fig. 9, we can see similar performances exhibited for all elimination trees and all γ values except for $\gamma = 1$. In the case of $\gamma = 1$, the SuperblockGreedy scheme performs better than the other two schemes for matrices with 1750-3000 columns, despite still not reaching the performance of the other tested γ values.

5.5. Runtime systems comparison

Finally, we examine the differences between the two presented runtime systems. Figure 10 shows that the performance of both runtime systems is very similar for wider matrices, while the OpenMP runtime system performs slightly better for skinnier matrices for both tested parameter sets.

6. Conclusions

The results of experiments with the NVIDIA Grace CPU Superchip bring us mostly to similar conclusions as the results from nodes tested in [6]. Nevertheless, the Grace node results have a few distinct features:

- The performance drop of the TtGreedy scheme for wider matrices is much less significant.
- There is a difference in performances of the SuperblockGreedy scheme and the other two superblock-based schemes for $\gamma = 1$. More specifically, the

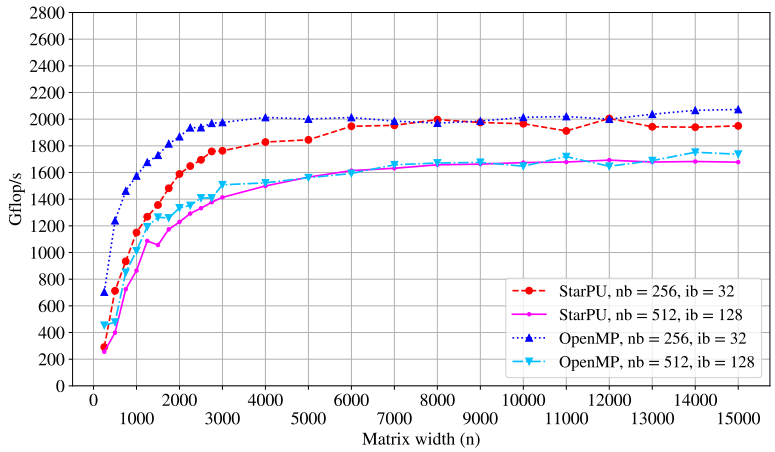

Figure 10: Performance of the OpenMP and StarPU runtime systems. The SuperblockGreedy scheme with inner blocks of size 32 and 128 (for block sizes of 256 and 512, respectively), layout translation disabled, and $\gamma = 4$.

SuperblockGreedy scheme performs better for certain matrix sizes. Hence, $\gamma \geq 2$ can be recommended also for this architecture.

- The difference between the OpenMP and StarPU runtime systems is small for skinny matrices and even smaller for wider matrices. Nevertheless, OpenMP still seems as a good choice for implementing this algorithm.

Details of the algorithm and results for different architectures will appear in [6].

Acknowledgments

This work was supported by the Student Summer Research Program 2023 of FIT CTU in Prague, by the Czech Science Foundation under project GA ČR 23-06159S, and by the Institute of Mathematics of the Czech Academy of Sciences (RVO:67985840). The computational time on the systems at IT4Innovations was provided thanks to the support by the Ministry of Education, Youth and Sports of the Czech Republic through the e-INFRA CZ (ID:90254).

References

[1] Anderson, E. et al.: *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1999, Third edn.

[2] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23** (2011), 187–198.

[3] Blackford, S.L. et al.: *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[4] Bouwmeester, H., Jacquelin, M., Langou, J., and Robert, Y.: Tiled QR factorization algorithms. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11, Association for Computing Machinery, 2011 .

[5] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Computing **35** (2009), 38–53.

[6] Břichňáč, V., Šístek, J., and Langou, J.: Effect of different elimination schemes on task-based implementation of qr factorization for multicore architectures. In preparation, 2025.

[7] Demmel, J., Grigori, L., Hoemmen, M., and Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. SIAM Journal on Scientific Computing **34** (2012), A206–A239.

[8] Dongarra, J. et al.: Hierarchical QR factorization algorithms for multi-core clusters. Parallel Computing **39** (2013), 212–232.

[9] Dongarra, J. et al.: PLASMA: Parallel linear algebra software for multicore using OpenMP. ACM Trans. Math. Softw. **45** (2019), 16:1–16:35.

[10] Faverge, M., Langou, J., Robert, Y., and Dongarra, J.: Bidiagonalization with parallel tiled algorithms. Tech. Rep. R-8969, INRIA, 2016.

[11] Fukaya, T., Kannan, R., Nakatsukasa, Y., Yamamoto, Y., and Yanagisawa, Y.: Shifted Cholesky QR for computing the QR factorization of ill-conditioned matrices. SIAM Journal on Scientific Computing **42** (2020), A477–A503.

[12] Fukaya, T., Nakatsukasa, Y., Yanagisawa, Y., and Yamamoto, Y.: CholeskyQR2: A simple and communication-avoiding algorithm for computing a tall-skinny QR factorization on a large-scale parallel system. In: *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. 2014 pp. 31–38.

[13] Higgins, A.J., Szyld, D.B., Boman, E.G., and Yamazaki, I.: Analysis of randomized Householder-Cholesky QR factorization with multisketching. arXiv:2309.05868v2, 2024.