Assignment 1: High Performance Dense Linear Solver

Christian Wiskott, 01505394.

April 29, 2020

Part 4: Blocked Right-looking LU Factorization

In this part of the assignment, the goal was to write an efficient algorithm for calculating a blocked right-looking LU-Factorization for a randomly generated, non-singular matrix $A \in \mathbb{R}^{n \times n}$. With this factorization, A is decomposed into an upper triangular matrix U and a lower triangular matrix L, such that $A = LU$. The main idea is similar to the unblocked LU-Factorization, but instead of iterating trough every single column of A, the matrix is divided into block matrices of block-size b, which are then factorized, to solve a system of equations in order to obtain the fully-factorized matrix LU. The advantage of this method, is that it uses matrix-matrix multiplications (*Blas Level 3*) instead of matrix-vector multiplications (*Blas Level 2*), which should result in a higher performance with respect to the run-time.

This paper seeks to prove this claim, by showcasing a direct comparison of unblocked and blocked LU-Factorization, as well as an evaluation of the optimal block-size b.

Performance Evaluation

In order to evaluate the performance of the written code, the run-time was measured, such that only the actual computation of the resulting matrices L and U were included in the measurement. Similar to the unblocked LU-Factorization, the computation required three explicit for-loops, in order to factorize the $b \times n$ - matrices below the main diagonal, as well as one for-loop to iterate through the columns of A with step-size b and the subsequent solving of the system of equations outlined in rows 47-50 of the provided code, which consist of matrix multiplications. This

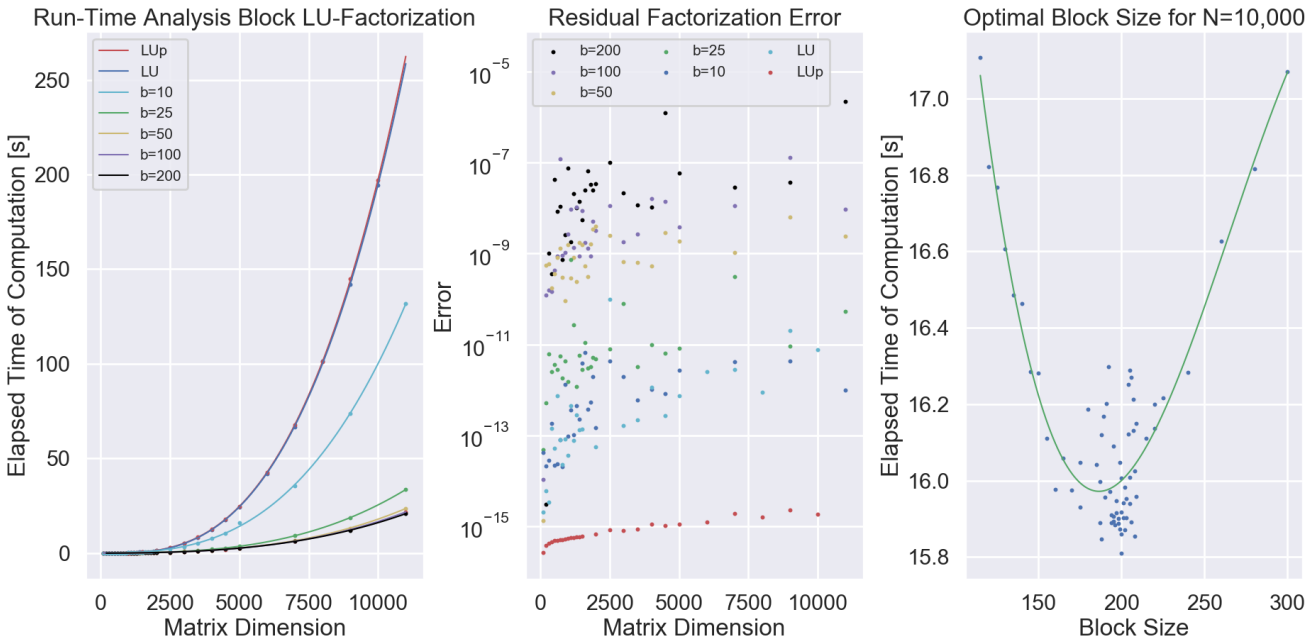

Figure 1: Comparison of run-times and errors for unblocked and blocked methods using different block-sizes b. *LUp* refers to LU-Factorization with pivoting. The optimal block-size was determined as b=200.

implementation utilized the same method as in Part 2&3 of the assignment and constructed only one matrix for the computation of LU, which required less memory for storing purposes.

To minimize the run-time of the algorithm, the Python module *numba* was used in a similar manner as in Part 2 of the Assignment (please refer to the respective document for more details on the parallelization of the code).

Fig. 1 shows the run-time analysis of the blocked and unblocked LU-factorizations for problem sizes within the range of [100, 11000]. The data has been fitted with cubic functions, which show (with high correlation) a cubic increase w.r.t the matrix dimension for unblocked- as well as blocked methods. The optimal block-size was obtained by increasing b in steps of 25 and plotting the computation time for the dimensions using the entire range. The left most subplot in Fig. 1 shows the decreasing run-time, until the minimum was reached at b=200. This was further examined and proven by calculating the LU-Factorization for the fixed dimension of n=10000 using different block-sizes between [100,300]. The right-most subplot demonstrates,

Method	T_{comp} [s]	T_{rel}	Avg. Rel. Error
With pivot	196.88	1	5.8×10^{-16}
Without pivot	194.44	0.988	2.7×10^{-13}
b=50	17.94	0.091	8.1×10^{-10}
b=100	16.67	0.085	2.7×10^{-9}
b=200	16.08	0.082	2.1×10^{-8}

Table 1: T_{comp} for N=10k. T_{rel} features the relative run-time compared to the slowest method, i.e LU-*without pivoting*. The average errors were obtained by using the median to get a more stable value, despite the outliers.

that the computation-time reached a minimum at exactly b=200. Therefore, b=200 was accepted for the optimal block-size.

The analysis of the residual factorization errors of the different block-sizes, revealed that the errors increase with increasing block-size. Table 1 showcases the comparison of the median of the errors, seen in the middle plot of fig. 1.

By examining fig. 1, the comparison of run-time and accuracy of the different LU-factorizations clearly shows, that the blocked variants completed the same task much faster the the unblocked variants. Using the optimal block-size of b=200, the task with n=10000 was completed approx. 12-times faster than the unblocked factorizations, which is a significant performance-increase. However, the increased performance does not come without its cost. The blocked methods generally lead to greater errors compared to the unblocked methods, with unblocked LU-*with pivoting*, being the most accurate of all the algorithms.

Given the apparent advantages and disadvantages of the blocked variants, it cannot be said, that they are generally superior to the unblocked methods. If the main objective, is to increase performance (i.e. minimize run-time) the blocked algorithms are a better choice. If however the accuracy of the results is of higher importance, the unblocked LU-*with pivoting* should be preferred. For a compromise of performance and accuracy, the best choice would be a blocked method with a relatively small block-size to obtain acceptable errors, while still maintaining better performance. Another option would be to use of a blocked method *with pivoting*, which should improve the errors, but since this was not included in this case study, it's not possible to give any concrete recommendations.

In order to provide a reference value for the run-time, the code was executed on the server the university provided. Fig. 2 showcases the comparison of all previous variants of the LU-Factorization. Since the server doesn't feature the module *numba*, it had to be disabled and therefore needed significantly more time to complete the

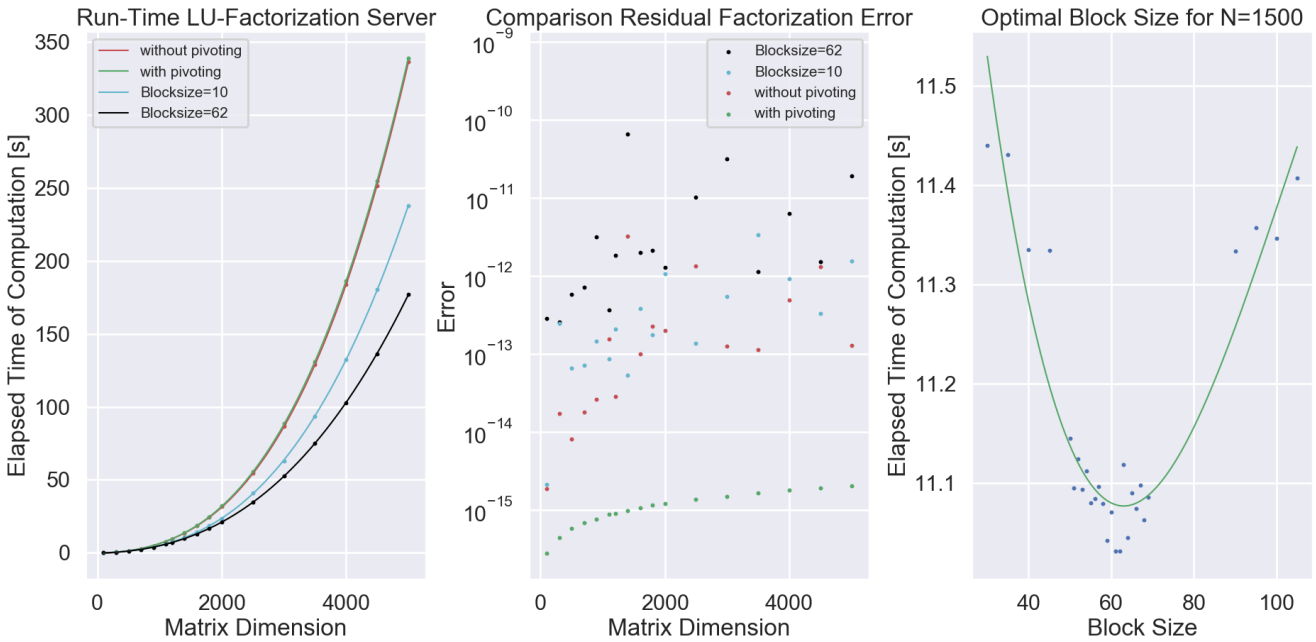

Figure 2: Run-time and error-analysis using the provided university server

tasks.

The same strategy was used to find the optimal block-size. Fig. 2 shows the minimum run-time at b=62 for the fixed dimension of n=1500. The two different optimal block-sizes are probably due to the use of *numba*, which optimizes the code automatically and therefore could result in unpredictable behaviour. Table 2 showcases the comparison of results for the different methods using the server. The blocked LU with optimal block-size completed the same task at n=5000 approx. twice as fast as the unblocked method without pivoting. The errors show the same behaviour, such that increasing block-size leads to more sever errors.

Efficiency

The efficiency was determined using the provided material from Prof. Gansterer. Efficiency is defined as:

$$E = \frac{P_s}{P_p} = \frac{W}{RP_p} \tag{1}$$

Method	T_{comp} [s]	T_{rel}	Avg. Rel. Error
Without pivot	336.39	1	1.21×10^{-13}
With pivot	338.90	1.01	1.06×10^{-15}
b=10	237.64	0.71	2.06×10^{-13}
b=62	177.01	0.53	1.82×10^{-12}

Table 2: T_{comp} for N=5000. The average errors were obtained by using the median to get a more stable value, despite the outliers.

with P_s as the sustained performance, P_p as the peak performance (8.8 GFlops using the university server), W as the amount of work in Flops and t as the runtime.

Fig. 3 shows the efficiency for all computations in assignment 1 in comparison to each other. From this illustration it's visible, that *Forward-* is a bit more efficient than *Backward Substitution*. From the comparison of blocked and unblocked LU-methods, it's obvious that the blocked LU (using the optimal block size) is substantially more efficient than the unblocked methods.

Figure 3: Analysis of the efficiency of all results from assignment 1, using the provided university server. To determine the efficiency of the blocked LU, the optimal block size b=62 was used.