# Part 2: Unblocked Right-looking LU Factorization

Lukas Gosch
01326060

May 13, 2018

## Contents



# 1 Preface

The following subsections except BLAS and Makefile are mainly the same as for task 1.

## 1.1 Processor

The (C++) code is executed on an Intel Core i7 - 4650U CPU from the 4. Generation (Haswell architecture). The specifications are given in table 1.

| | |
|---|---|
| Base Frequency | 1.7 GHz |
| # of Cores | 2 |
| L2 Cache (per Core) | 256 KB |
| L3 Cache (per Core) | 4 MB |
| Double Precision (DP) Operations | 16 FLOP/cycle (two 4-wide FMA[1]instructions) |
| Single Precision (SP) Operations | 32 FLOP/cycle (two 8-wide FMA instructions) |

Table 1: Processor Specifications

The resulting theoretical peak performance is given in table 2.

| # of Cores | DP Operations | SP Operations |
|---|---|---|
| Single Core | 27.2 GFLOPS | 54.4 GFLOPS |
| Dual Core | 54.4 GFLOPS | 108.8 GFLOPS |

Table 2: Theoretical Peak Performance

To note is, that the theoretical peak performances are calculated using the FMA execution units.

## 1.2 Compile Options

The C++ code is compiled using clang-700.1.81.

FMA is supported by the AVX2 instruction set. To enable this processor-specific optimization, the compile option *-march=core-avx2* is set. In the makefile this option is changed to *-march=native* to make it processor independent. Furthermore the optimization flag *-O3* is set as it results in a significant performance boost. *-ffast-math* is enabled too, as it does not seem to have much effect on the accuracy but fasten the calculations. An important note is, that *-ffast-math* is an unsafe optimisation (changes the order of calculations and therefore changes the solution), but for our problem the errors stay in the same order of magnitude, therefore it seems safe to use.

### 1.2.1 BLAS

To speed up linear algebra operations, the CBLAS library is used. From this library the functions *cblas_daxpy* and *cblas_dgemm* are called. Therefore the compile option *-lblas* in the clang compiler is set.

### 1.2.2 Makefile

The *C++* source file is called *LU_gosch.cpp* and can be compiled with the above listed commands with the given Makefile. The Makefile creates an executable output file *LU_gosch.out*.

## 1.3 Evaluation of Runtime Performance

For runtime evaluations *std::clock* defined in header $< ctime >$ is used. It measures processor time used by the relevant process (programm) in clock ticks. A so calculated interval is divided by the constant *CLOCKS_PER_SEC* beeing one million on my system. Therefore precision seems good enough for the purpose of our task.

An important distinction has to be made between processor time and wall time. If the CPU is shared by other processes or the measured process is multi-threaded and executed on multiple cores, wall time and processor time can differ significantly[2].

[1] FMA stands for fused multiply and add

[2] See: http://en.cppreference.com/w/cpp/chrono/c/clock

## 1.4 Row Major Order vs Column Major Order

To represent vectors and matrices, C-style arrays or C++ wrappers around C-style arrays (e.g. vectors) are used. If the matrix is stored in row-major order, an array corresponds to one row of a matrix and the matrix then is an array of arrays. As an array object consists of contiguously allocated objects of a certain type, iterating over a so defined matrix in column major order is expected to perform worse as iterating over the matrix in row major order. (This expectation is reversed, if an array is interpreted as one column of matrix.)

Testing this hypothesis for different problem sizes indeed confirms our expectation and shows that the (relative) performance gap between the different orders even increases with increased problem size. This is shown in figure 1.

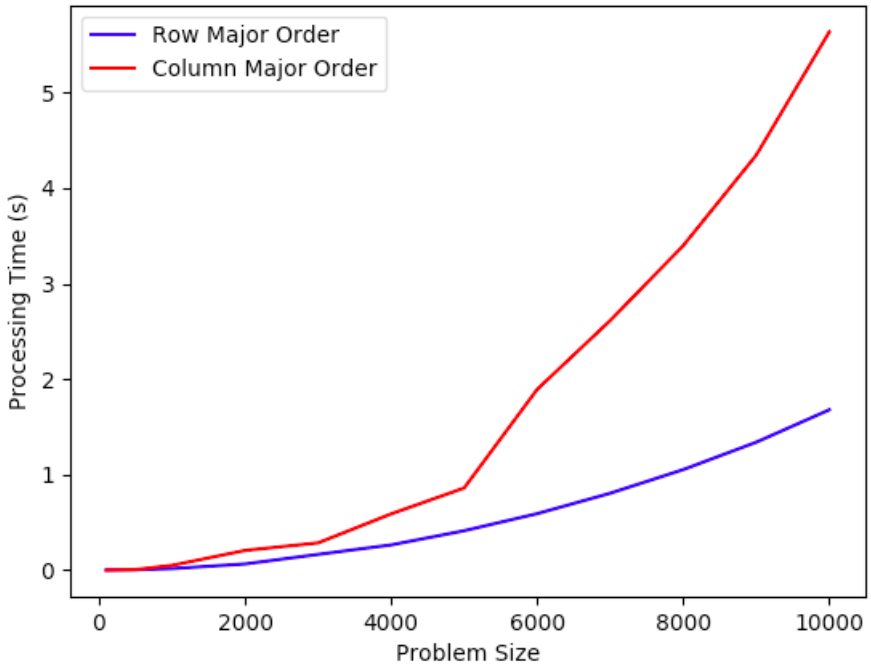


Figure 1: Performance difference if iterating a matrix in row major order or column major order.

To compare performances, two identical random matrices of size n stored in row-major order where created and iterated over in row-major or column-major order. Then for each element in the respective matrix, one floating point multiplication and one floating point addition was performed (exactly the same operations for both iteration orders).

Therefore, for the following algorithm, attention was payed to storing and iterating a matrix in the most efficient order.

# 2 Implementation

For the implementation, the unblocked (right-looking) LU Factorization algorithm, as can be found in the lecture slides[3], was used. As the most-inner loop loops over all the row entries, the matrix which should be factorized is stored in row-major order. Furthermore, the inner loop over j was optimized using the CBLAS routine *cblas_daxpy* leading to a huge performance boost.

# 3 Results

## 3.1 Efficiency

To calculate the relative peak performance, the runtime of the algorithm is measured as explained in section 1.3. The work for both algorithms is $O(\frac{2}{3}n^3)$. Combined with the peak performance listed in section 1 and the following formula

$$Runtime = \frac{Work}{(Peak\ Performance\ \times Efficiency)} \tag{1}$$

the efficiency (relative performance) of the implementation could be calculated.

Graph 2 shows the relative performance (relp) of the algorithm given a problem size n.

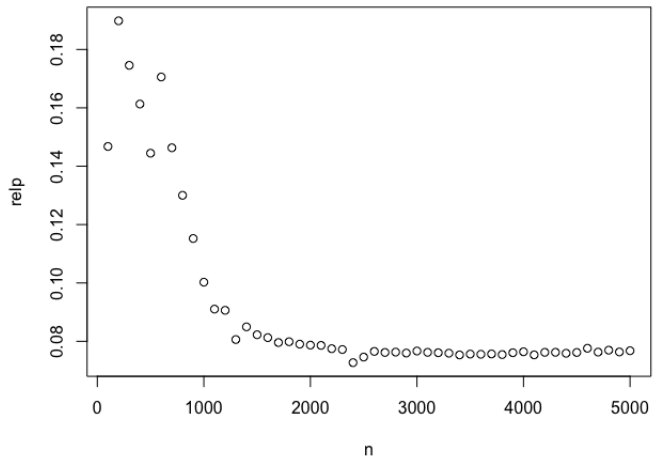


Figure 2: Relative performance (relp) of the unblocked right-looking LU Factorization given a certain problem size n.

It shows high efficiency and variance till a problem size of around 1000. From there on it seems to stabilise at an efficiency of ∼0.076. The average efficiency

[3]BLAS and blocked LU Factorization Slide 25

is ∼0.092 with a maximum of ∼0.190 for n=200. For n=5000 the efficiency is ∼0.077 and the runtime ∼39.90s.

Graph 3 shows the runtime of the algorithm given a certain problem size n.

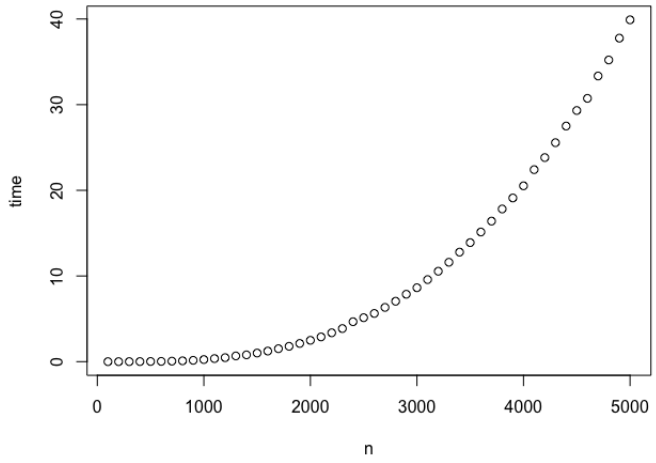


Figure 3: Runtime of the unblocked right-looking LU Factorization given a certain problem size n.

Graph 4 confirms that the runtime of the algorithm nearly perfectly scales with $\frac{2}{3}n^3$.

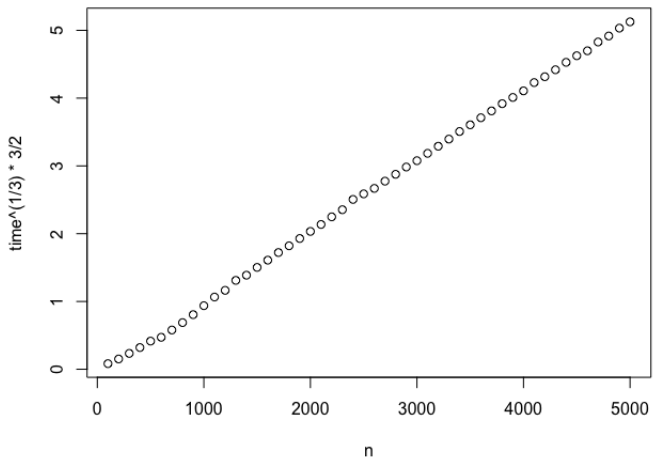


Figure 4: Transformed runtime of the unblocked right-looking LU Factorization given a certain problem size n.

## 3.2 Accuracy

To avoid using external libraries, the random triangular matrices are generated manually using the *mersenne_twister_engine* from the standard library. Random matrices are filled with values drawn from a uniform real distribution with $p(x) = \frac{1}{b-a}$ where $a = 1$ and $b = 1.5$.

Accuracy should not depend as extremely on the condition number as in forward or backward substitution but again the matrix is not allowed to be singular. As the condition number of the randomly generated matrix depends on how correlated the column vectors of the randomly generated matrix are (and is an indicator for singular/near singular matrices), a good conditioned matrix with a low condition number should be generated. Therefore, to lower the condition number, the diagonal entries are again multiplied by a factor of 20. Furthermore, the sign of each entry is chosen randomly. This forces a "more linearly independent column space" leading to a better condition number.

Graph 5 shows that the relative factorization error (rfacte) of the algorithm seems very low. It shows a slight upward trend and an increase in variance for increased n. For n=5000 the relative factorization error is $\sim 2.33 \cdot 10^{-11}$.

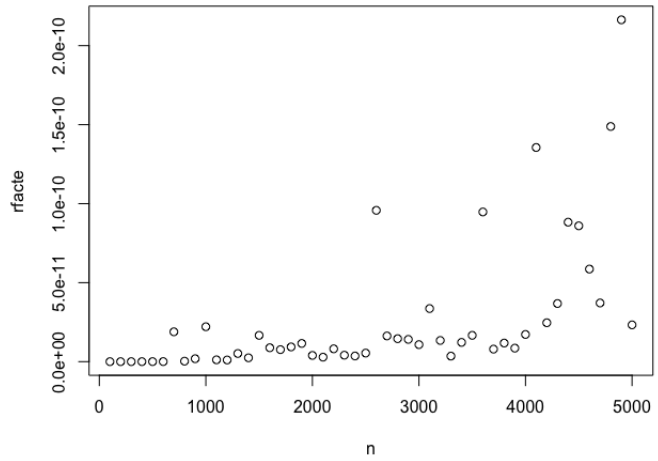


Figure 5: Relative factorization error (rfacte) of the unblocked right-looking LU Factorization given a certain problem size n.