Part 1: Triangular Solver

Lukas Gosch
01326060

May 13, 2018

Contents

1 Preface

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 Makefile

The *C++* source file is called *task1_gosch.cpp* and can be compiled with the above listed commands with the given Makefile. The Makefile creates an executable output file *task1_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].

1.4 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

[1]FMA stands for fused multiply and add
[2]See: http://en.cppreference.com/w/cpp/chrono/c/clock

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.

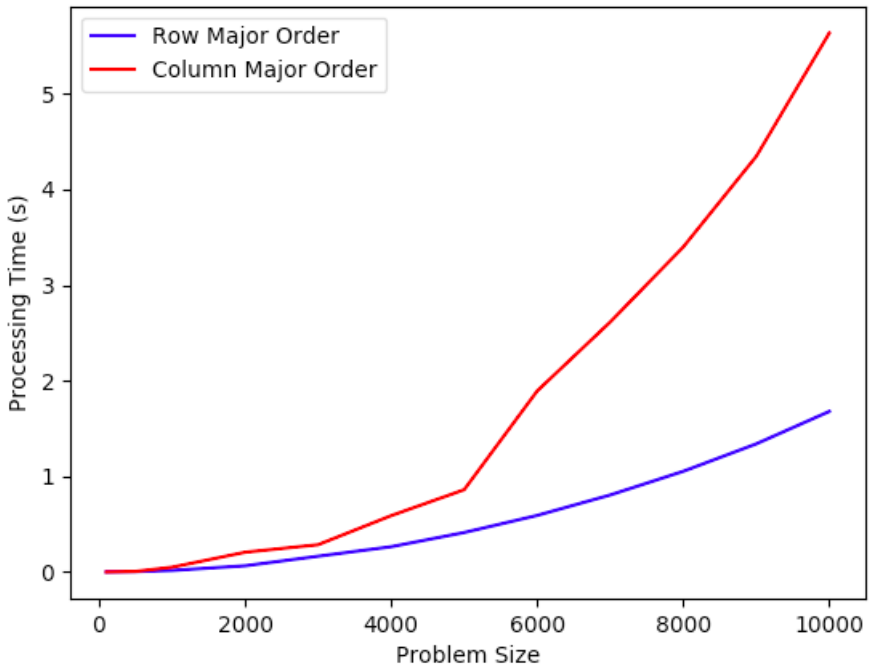

Figure 1: Performance difference if iterating a matrix in row major order or column major order.

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 algorithms, attention was payed to storing and iterating a matrix in the most efficient order.

2 Implementation

2.1 Forward Substitution

For the implementation of the forward substitution algorithm, the algorithm presented in the lecture was used. As each inner loop loops over a subset of column entries, the lower triangular matrix is stored in column major order to speed up column element access.

2.2 Backward Substitution

For the implementation of the backward substitution algorithm, the algorithm presented in the lecture was used. As each inner loop loops over a subset of column entries, the upper triangular matrix is stored in column major order to speed up column element access.

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(n^2). 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.

3.1.1 Forward Substitution

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

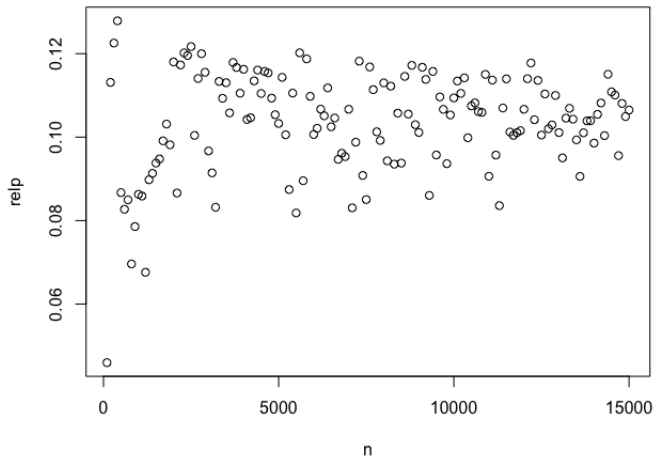

Figure 2: Relative performance (relp) of forward substitution given a certain problem size n.

It shows that independent of problem size, the efficiency seems to vary around some mean of around ∼0.10. The average efficiency is calculated to

be ∼0.1036, the maximum efficiency is ∼0.1279 at a problem size of 400. The efficiency for n=15000 is ∼0.1065 with a total runtime of ∼0.0777s.

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

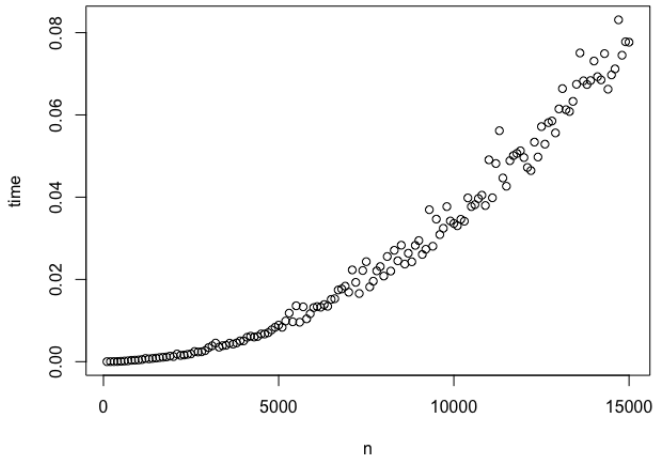

Figure 3: Runtime of forward substitution given a certain problem size n.

Graph 4 confirms that the runtime of the algorithm nearly perfectly scales with n^2.

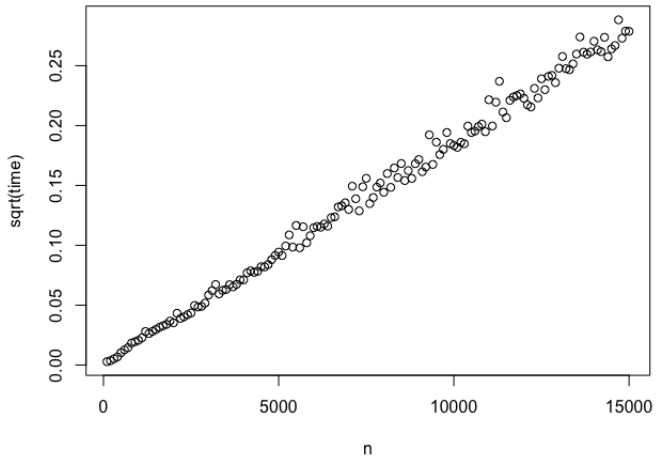

Figure 4: Transformed runtime of forward substitution given a certain problem size n.

3.1.2 Backward Substitution

Graph 5 shows the relative performance (relp) of the algorithm given a problem size n.

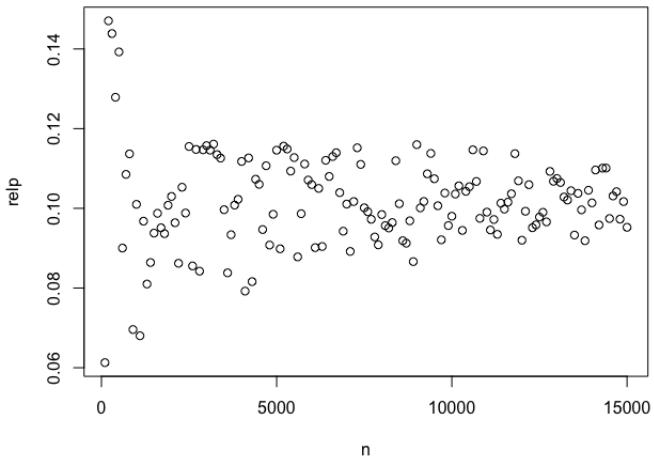

Figure 5: Relative performance (relp) of backward substitution given a certain problem size n.

It shows that for increasing problem sizes, the efficiency varies less and less around a mean of around ∼0.10. The average efficiency is ∼0.1018, the maximum efficiency is ∼0.1471 at a problem size of 200. The efficiency for n=15000 is ∼0.0952 with a total runtime of ∼0.0869s.

Graph 6 shows the runtime of the algorithm given a certain problem size n.

Graph 7 confirms that the runtime of the algorithm nearly perfectly scales with n^2.

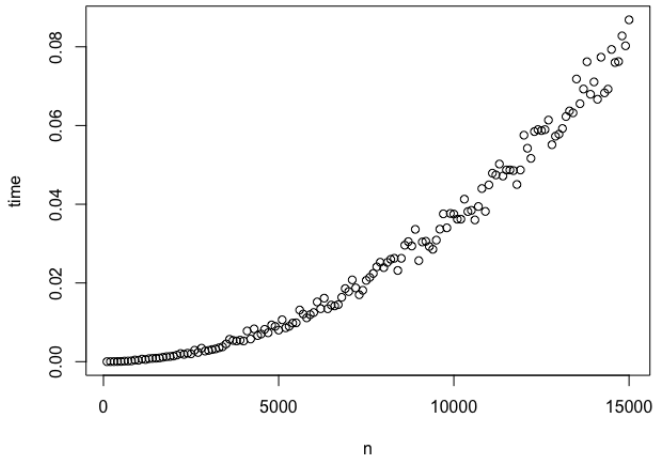

Figure 6: Runtime of backward substitution given a certain problem size n.

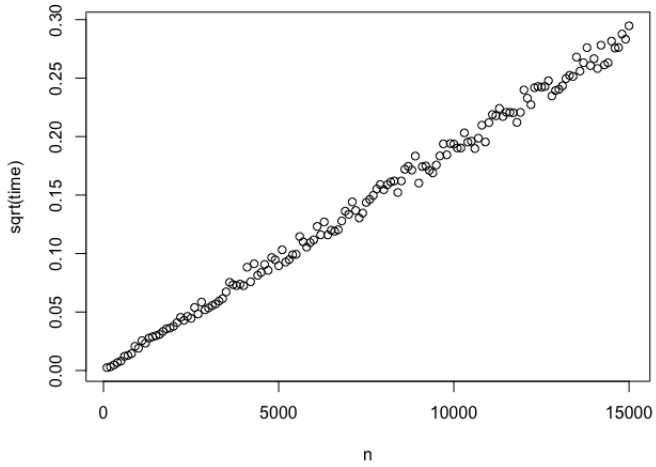

Figure 7: Transformed runtime of backward substitution given a certain problem size n.

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 hugely depends on the condition number of the randomly generated triangular matrix which itself depends on how correlated the column

vectors of the randomly generated matrix are. Therefore, to lower the condition number, the diagonal entries are multiplied by a factor of 20. Furthermore, the sign of each entry is chosen randomly. Therefore a "more linearly independent column space" is forced which leads to a better condition number.

3.2.1 Forward Substitution

The relative forward error (rfe) of the algorithm scales logarithmically with the problem size n as seen in graph 8 and is very low for the good! conditioned triangular input matrix. The relative forward error for n=15000 is $\sim 1.98 \cdot 10^{-8}$.

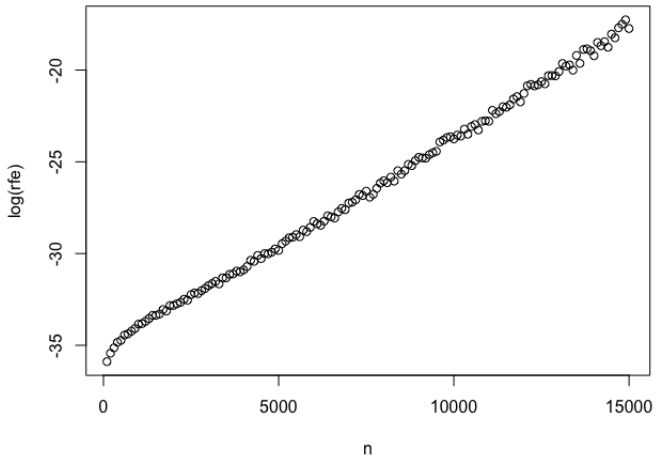

Figure 8: Logarithmic transform of relative forward error (rfe) of forward substitution given a certain problem size n.

The relative residual norm shows a very strange pattern as seen in graph 9 but is again very low for the good condition triangular input matrix. The relative residual norm for n=15000 is $\sim 3.78 \cdot 10^{-11}$.

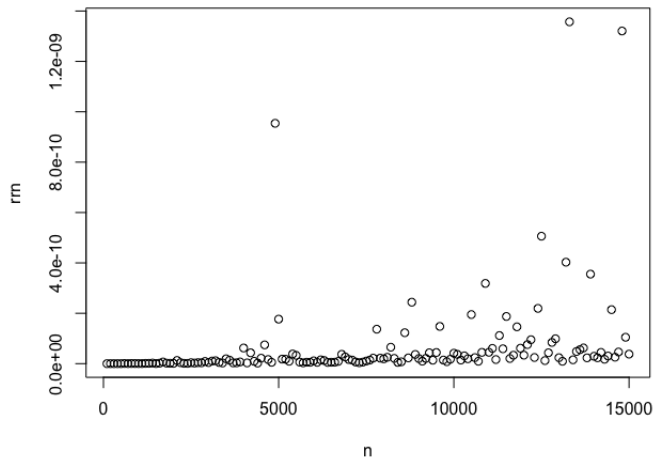

Figure 9: Relative residual norm (rrn) of forward substitution given a certain problem size n.

3.2.2 Backward Substitution

The relative forward error (rfe) of the algorithm again scales logarithmically with the problem size n as seen in graph 10 and is very low for the good! conditioned triangular input matrix. The relative forward error for n=15000 is $\sim 1.97 \cdot 10^{-8}$.

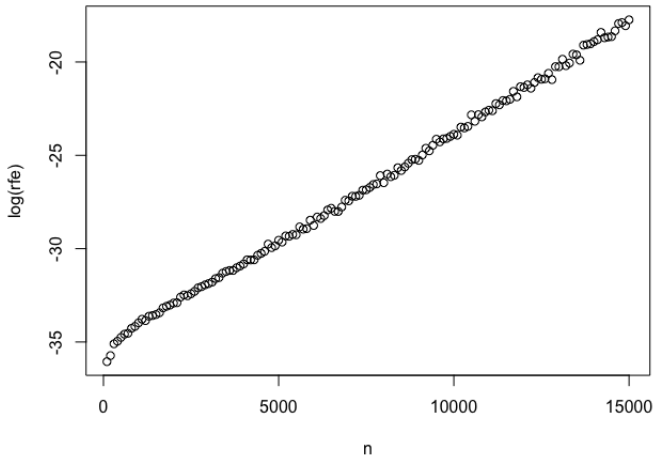

Figure 10: Logarithmic transform of relative forward error (rfe) of backward substitution given a certain problem size n.

The relative residual norm, different from forward substitution, scales polynomially with the problem size n as seen in graph 11 and is again very low for the good condition triangular input matrix. The relative residual norm for n=15000 is $\sim 3.53 \cdot 10^{-11}$.

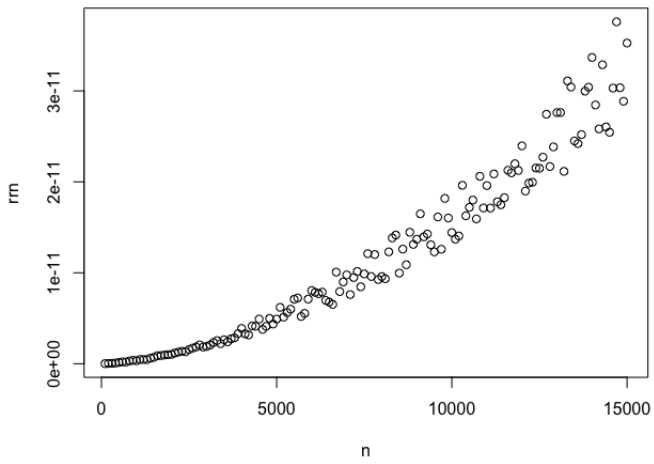

Figure 11: Relative residual norm (rrn) of backward substitution given a certain problem size n.