Part 3: Unblocked Right-looking LU Factorization with Partial Pivoting

Lukas Gosch
01326060

May 13, 2018

Contents

1 Preface

The following subsections except Makefile are mainly the same as for part two.

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_pp_gosch.cpp* and can be compiled with the above listed commands with the given Makefile. The Makefile creates an executable output file *LU_pp_gosch.out*. As the random matrix A is generated a little different compared to part two (as explained in section 3.2) , the modified source file from part two *LU_gosch.cpp* is also submitted and can be executed if the name of the source file is changed in the makefile.

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.

[1] FMA stands for fused multiply and add

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 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.

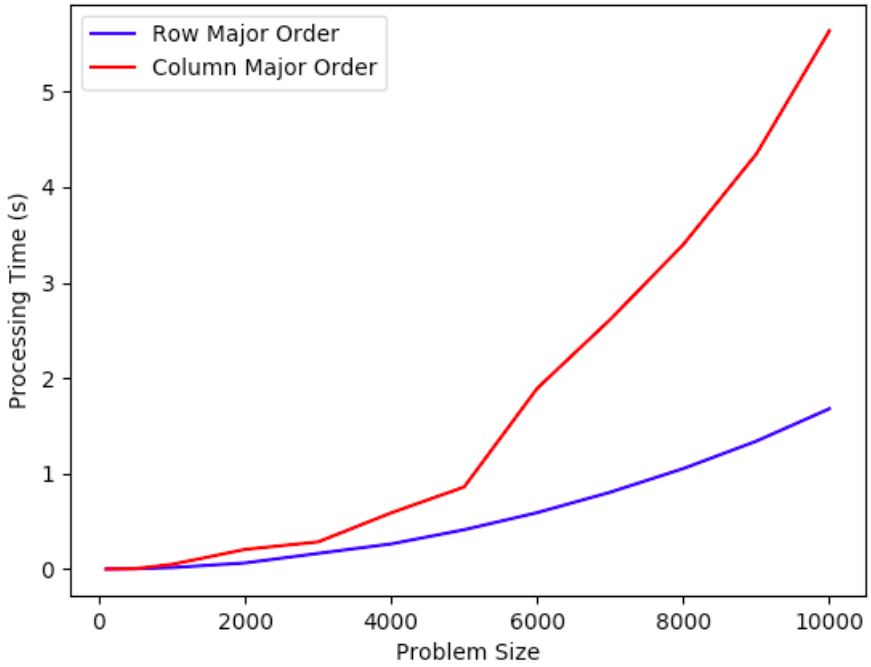

Figure 1: Performance difference if iterating a matrix in row major order or column major order.

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

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

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 with partial pivoting algorithm, as can be found in the lecture slides[3], was used.

The permutation matrix is implemented as a permutation vector to minimize storage complexity and row swapping of the system matrix was realized using the *std::swap* function on two rows stored as vectors. This leads to a constant time complexity for the swap process as only pointers are reset and no element-wise swap has to be performed.

On the row-swapped system matrix, the unblocked right-looking LU Factorization of the previous task is applied. Therefore the matrix is again stored in row-major order. Furthermore, the inner loop over j was again optimized using the CBLAS routine *cblas_daxpy*.

3 Results

For the following sections, the results of the LU-Factorization algorithm of part 2 are compared with the results of the LU-Factorization algorithm with partial pivoting.

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 LU-Factorization algorithms (with and without partial pivoting) 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 efficiency of the algorithms given a problem size n.

Both algorithms show high efficiency and variance till a problem size of around 1000. From there on both algorithms seem to stabilise. The average efficiency of the LU-Factorization without partial pivoting is ∼0.089 and therefore slightly better as the LU-Factorization with partial pivoting which has an average efficiency of ∼0.084. The peak efficiency of the LU-Factorization without partial pivoting is ∼0.186 at n=500 and of the LU-Factorization with partial pivoting is ∼0.164 at n=300.

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

[3]BLAS and blocked LU Factorization Slide 43

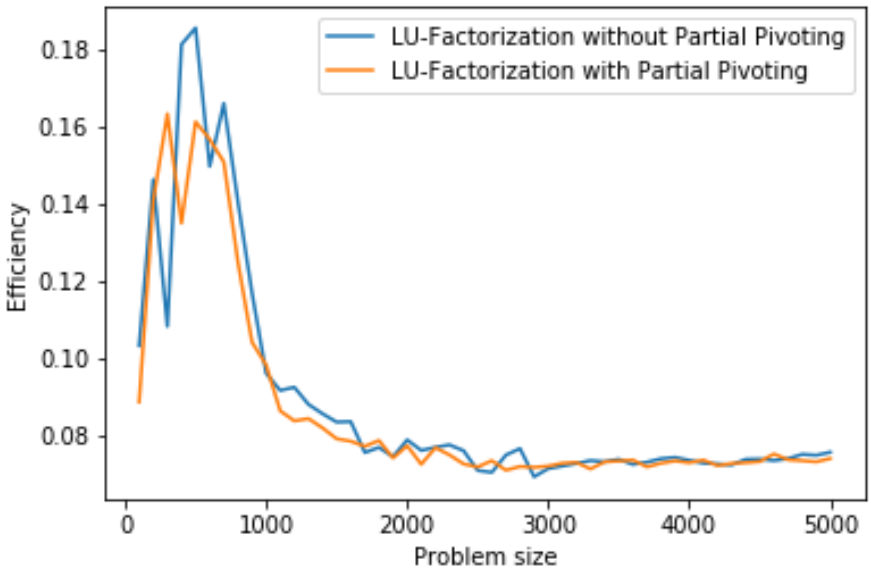

Figure 2: Efficiency of LU-Factorization with and without partial pivoting given a certain problem size

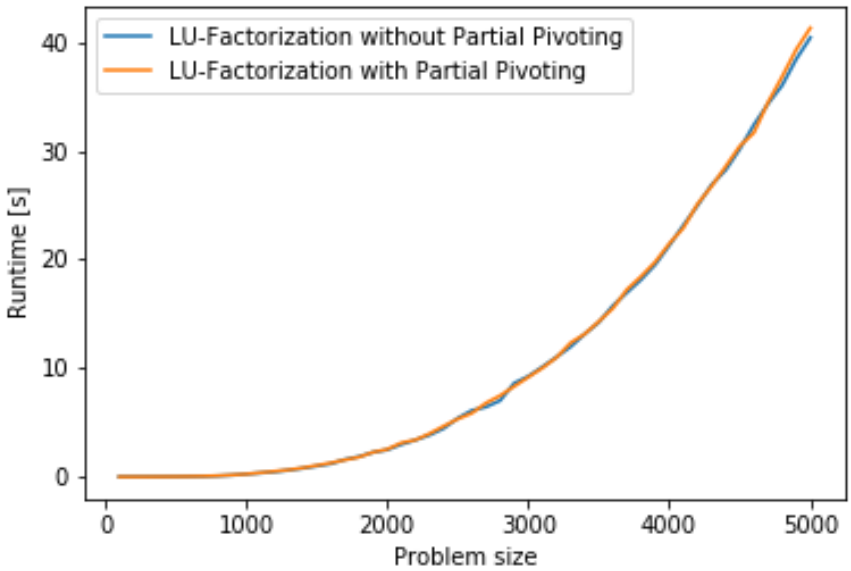

Figure 3: Runtime of LU-Factorization with and without partial pivoting given a certain problem size

For the biggest problem of size n=5000, LU-Factorization without partial pivoting has a runtime of ∼40.44s whereas the LU-Factorization with partial pivoting has a runtime of ∼41.33s beeing nearly a second slower.

The slightly worse performance of the LU-Factorization with partial pivoting regarding runtime and efficiency can be explained by the extra work needed for finding a (maximal) pivot. Therefore the time complexity of both algorithms is the same with regards to the leading order n^3 but differs in n^2.

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 $a = 0$ and $b = 1$.

To compare the accuracy of LU-Factorization with and without partial pivoting and to highlight their differences, the system matrix A is randomly generated as in part two, but instead of generating a good conditioned matrix with large diagonal entries, A is generated with very small diagonal entries compared to the rest of the entries in A. This is realized by multiplying every diagonal element by a factor of 10^{-14}.

Therefore it is expected that the LU-Factorization without partial pivoting performs significantly worse then the LU-Factorization with partial pivoting.

Graph 4 shows that the relative factorization error of the algorithms (logarithmically transformed to the base 10). The error of the LU-Factorization with partial pivoting is consistently lower as the LU-Factorization without partial pivoting by around twelve orders of magnitudes marking it as way more accurate. The error of the LU-Factorization without partial pivoting shows oscillatory behaviour of constant sized amplitudes whereas the LU-Factorization with partial pivoting shows a very slow increase problem size and much more less pronounced oscillatory behaviour.

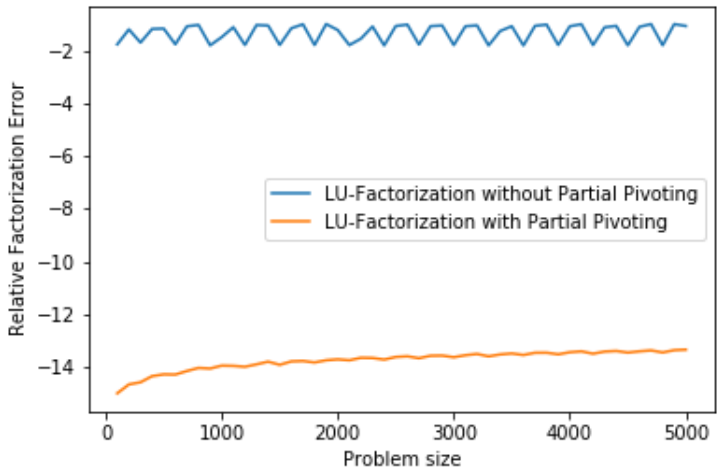

Figure 4: Relative factorization error (logarithmically transformed to base ten) of LU-Factorization with and without partial pivoting given a certain problem size.

3.3 Conclusion

Given the nearly same runtime performance of the LU-Factorization with and without partial pivoting but the way more stable behaviour of the LU-Factorization with partial pivoting, LU-Factorization with partial pivoting seems preferable in most cases compared to without partial pivoting.