Part 4: Blocked Right-looking LU Factorization with Partial Pivoting

Lukas Gosch
01326060

May 7, 2018

Contents

1 Preface

The following subsections except BLAS and Makefile are mainly the same as for part three.

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 LVL I, II and III functions are called. Therefore the compile option *-lblas* in the clang compiler is set.

1.2.2 Makefile

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

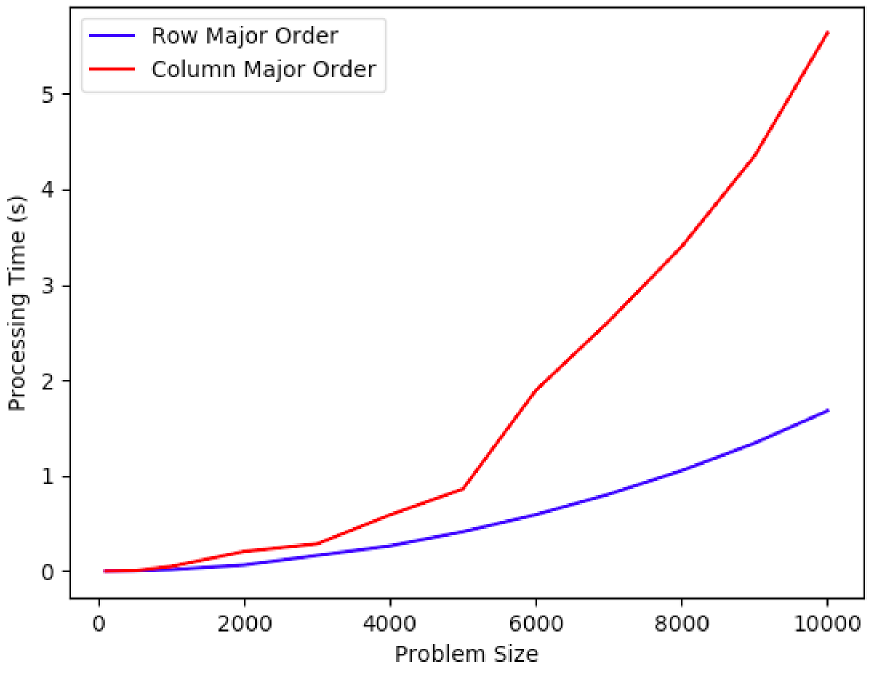

Figure 1: Performance difference if iterating a matrix in row major order or column major order.

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 blocked right-looking LU-Factorization algorithm from the *BLAS and Blocked LU Factorization*-slides was used and extended by partial pivoting as stated on the last page of the slides.

To effectively make use of the BLAS III functions for the blocked algorithm, the double-vector representation of a matrix used in the previous parts has to be changed to a single C-style array. Therefore, the unblocked LU-Factorization with partial pivoting is reimplemented using only C-style arrays and BLAS LVL I and II functions. In the array, a column major storing order is chosen.

Very good run times could be achieved by setting the block parameter b to a value between 30 and 80. For this report, b was chosen to be 40. For a b smaller 30 or bigger 80, significant runtime increases could be observed.

3 Results

For the following sections, the results of the unblocked LU-Factorization algorithm with partial pivoting from part three are compared with the results of the blocked 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 (blocked and unblocked) 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 blocked and unblocked algorithms given a problem size n.

Compared to the unblocked LU-Factorization, the blocked algorithm shows a longer period of high variance till a problem size of around 2500 and then stabilises around an efficiency of ∼0.4. Till n=5000, it does not show such a stabilised efficiency behaviour as the unblocked algorithm. The blocked LU-Factorization by far outperforms the unblocked LU-Factorization in terms of the efficiency. This can be explained by the use of the BLAS LVL III functions which are the fastest of the BLAS functions.

The blocked LU-Factorization achieves peak efficiency of ∼0.627 at n=1200 and has an average efficiency of ∼0.417. This beats the average efficiency of ∼0.086 of the unblocked LU-Factorization by nearly a factor of five.

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

For n=5000, the blocked algorithm needs ∼7.53s to compute the LU-Factorization and in that is ∼33.8s faster then the blocked algorithm.

Again it is to note, that the time complexity (work) of both algorithms is exactly the same but that matrix multiplications using BLAS can be computed

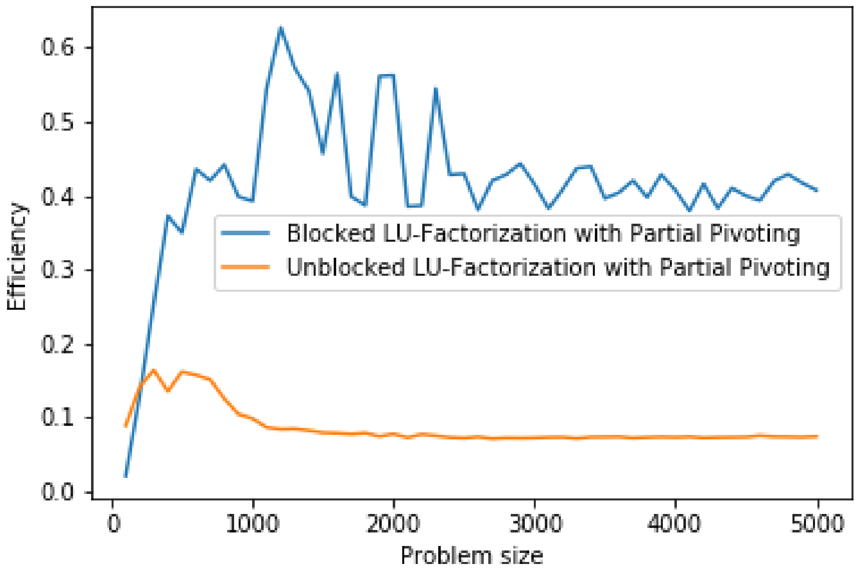

Figure 2: Efficiency of the blocked and unblocked LU-Factorization with partial pivoting given a certain problem size

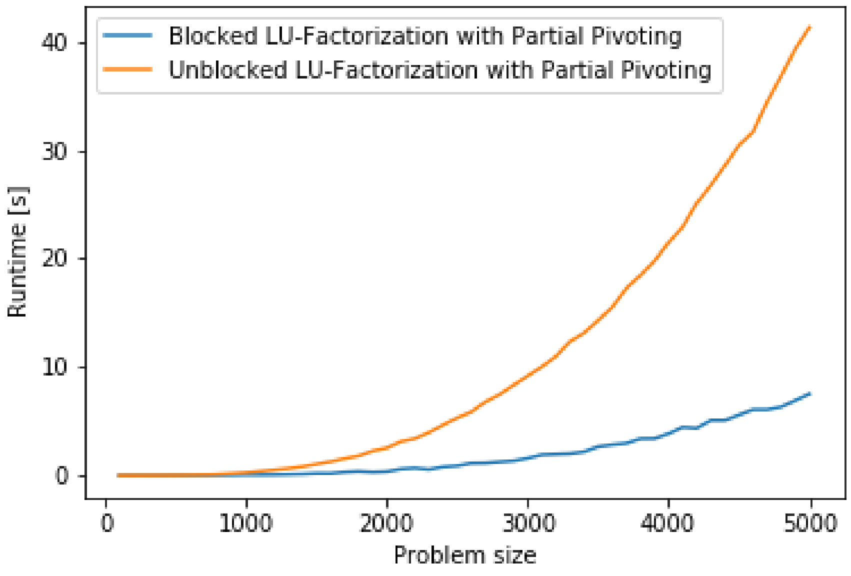

Figure 3: Runtime of the blocked and unblocked LU-Factorization with partial pivoting given a certain problem size

extremely efficiently marking the blocked algorithm superior over the unblocked algorithm in terms of practical performance.

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 the unblocked LU-Factorization with partial pivoting with the blocked one, the system matrix A is randomly generated using the same mechanism as in part three (including multiplying every diagonal element by a small factor).

Graph 4 shows that the relative factorization error of the algorithms (logarithmically transformed with base 10). Both factorization algorithm show a very low factorization error and the factorization error of both algorithms shows the same systematic of slowly increasing and a small oscillatory behaviour. To note is that the blocked LU-Factorization consistently shows a little bit of a smaller error then the unblocked algorithm.

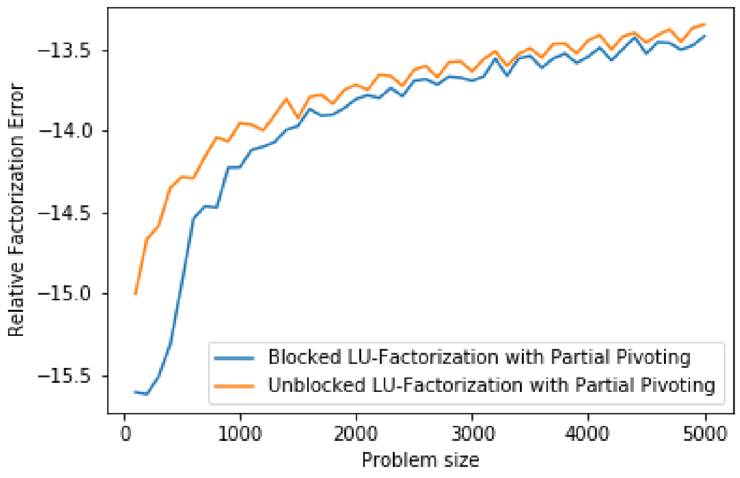

Figure 4: Relative factorization error (logarithmically transformed with base ten) of the blocked and unblocked LU-Factorization with partial pivoting given a certain problem size.

3.3 Conclusion

Given the nearly same relative factorization error of the blocked and unblocked LU-Factorization with partial pivoting but the extreme performance increase of using BLAS III functions in the blocked algorithm, the blocked algorithm seems way superior over the unblocked algorithm for practical applications.