# Matrix Multiplication: Practical Use of a Strassen Like Algorithm

Rozman, Mitja and Eleršič, Miha

**Abstract:** ***In this paper we present and experimentally evaluate the performance of several practical algorithms for the matrix multiplication problem. We explain and implement classical algorithms, recursive algorithms as well as the Strassen algorithm. Additionally, we also present and evaluate several code tuning techniques. Our comparison includes our implementations of the algorithms as well as the algorithms from the uBLAS standard library. Finally, we also examine numerical stability of the algorithms.***

**Index Terms**: ***matrix multiplication, algorithm, Strassen, experiment, performance***

## 1. Introduction

Matrix multiplication is one of the basic operations used in many different fields such as machine learning, engineering and physics simulations. This makes implementing efficient and numerically stable algorithms for the operation important. In this article we compare different methods of matrix multiplication and their implementations. The main goal is to compare execution time of matrix multiplication algorithms when the size of the matrix grows.

Matrix multiplication is a binary operation on two matrices $A$ and $B$ resulting in matrix $C = A \cdot B$. Each entry in the new matrix is defined as scalar product of the corresponding row in the first matrix and the corresponding column in the second, i.e.,

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}.$$

Manuscript received Nov, 2017.

The authors are with the Faculty of Computer and Information Science, University of Ljubljana, Slovenia (e-mail: mitja.rozman@student.fmf.uni-lj.si).

For matrix $A$ of size $m \times n$ and matrix $B$ of size $n \times k$ we can therefore compute matrix $C$ of size $m \times k$.

Most known matrix multiplication algorithms are derived directly from this definition and only change the order of addition operations. Examples of this kind of algorithms are classic and recursive algorithm as well as block matrix multiplication. They all have asymptotic time complexity $\Theta(n^3)$ for two square $n \times n$ matrices.

There also exist algorithms with sub-cubic time complexity. These algorithms are of great theoretical importance but are usually not implemented. Authors who worked on this field are Strassen [17], Pan [14], Bini [5], Schonhage [16], Romani [15], Coppersmith [7], Winograd [7], Stothers [8], Williams [19].

Unfortunately most of the sub-cubic algorithms are very elaborate and sometimes come with a much larger constant, meaning that even in theory they would not be faster on practically sized matrices. This article tries to promote a Strassen-like algorithm, which is straightforward to understand and fairly simple to implement. We hope to demonstrate its practical value.

For comparison evaluation of our results we used uBLAS from the C++ library boost [2]. We used Boost because of its popularity and ease of use. Possible alternative libraries are Armadilo [1], Eigen [3] and Intel Math Kernel Library [4].

During the course of development of the algorithms presented in this paper we followed the systematic approach to generation of new ideas in research in computing [9]. Additionally, we also followed the process and methods of the algorithm engineering approach to development of practically efficient algorithms [6, 12, 13].

In the next section we describe the classi-

cal matrix multiplication method. In Section 3 we describe the divide and conquer approach to matrix multiplication. In section 4 we describe the Strassen method of matrix multiplication along with our own method similar to the Strassen method. In Section 5 we compare the numerical stability of the implemented algorithms. Finally, in Section 6 we compare the performance of our implemented algorithms.

## 2. Classical Algorithm

The most basic algorithm for matrix multiplication is derived directly from the definition of matrix multiplication using three for loops. In this basic algorithm we iterate thorough the rows of the first matrix and multiply each row with every column of the second matrix. For every row and column we compute the sum of the products of corresponding row and column entires.

### 2.1 Transposed Matrix

The standard way of representing dense matrices in computer memory are two dimensional arrays. Since the 2D array has to be stored in memory which is one dimensional, it needs to be linearized. This can be done in a row-major order or column-major order [11]. The ordering has a significant effect on the algorithm performance, because accessing consecutive elements in memory is faster on modern CPUs because of caching. However classic multiplication accesses one matrix consecutively by rows and the other consecutively by columns. Hence, regardless of the order we choose to store the matrices in, one of the matrices is accessed inefficiently.

Fortunately this can be fixed by transposing one of the matrices before multiplication. The question is, whether the overhead of first transposing one of the matrices is larger than the potential gains of better memory access pattern during multiplication.

In Figure 1 we compare the performance of Classic algorithm and a version where the second matrix is transposed. From the Figure we observe that the transposed version is faster. We can conclude that the overhead of transposing the matrix is lower than the speedup resulting from better use of the processor cache.

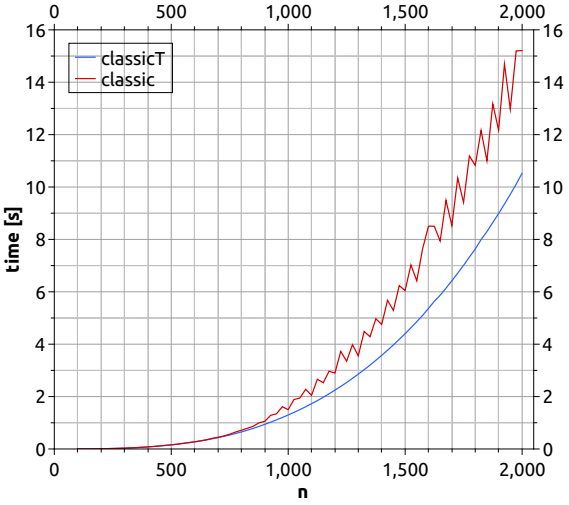


Figure 1: Running-time comparison of classic multiplication algorithms: without (classic) and with (classicT) transposition.

We can also observe higher variation in the running times of the non-transposed version. Although the results may look noisy, running more tests revealed the variation is consistent. In Figure 2 we can see the results of running $10$ tests for $10$ consecutive values of $n$. We can see that for each $n$ we get consistent results, but changing $n$ by one can change the results significantly.

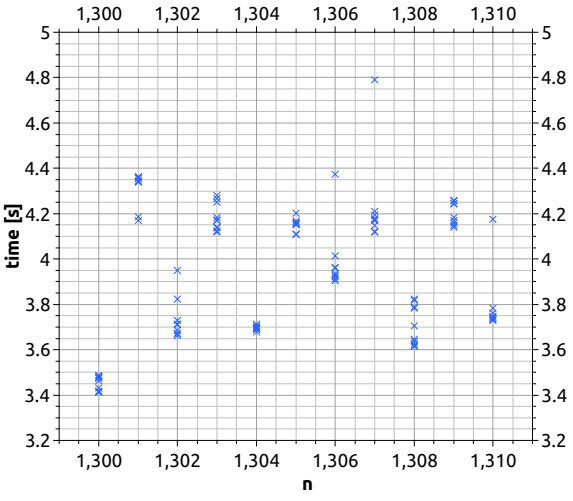


Figure 2: Running-time of untransposed classic algorithm on consecutive $n$. Each cross represents one test run without averaging.

### 2.2 Implementation with Vectors

For comparison we also implemented the classic algorithms with a nested vector implementation in C++ (`vector<vector<double>>`). This implementation allocates each row of the matrix separately, and the outer vector stores only references to the rows. In Figure 3 we can see the same comparison as in Figure 1, but using the nested vector data structure. Again the transposed version is faster, but this time the difference is more significant. If we compare the nested vector implementations to the linearized row-major implementations we observe that the untransposed classic multiplication is slower, but the transposed version is not.

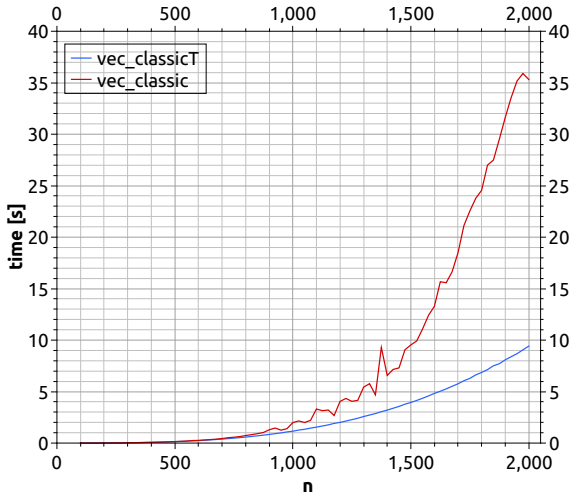


Figure 3: Running-time comparison of classic multiplication algorithms using a nested vector implementation: without (vec_classic) and with (vec_classicT) transposition.

## 3. Recursive Algorithm

Matrix multiplication can also be implemented with a divide and conquer approach[18]. We divide the matrices from the multiplication $C = AB$ into four blocks

$$\begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix} = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}.$$

Each block can be computed as follows.

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{1,2}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Each recursive step reduces the problem size and eventually the matrices are small enough that additional recursive steps are no longer optimal and it is more efficient to use the classic algorithm for such matrices. The minimal size of the matrix for which additional recursive steps are taken is analogous to the block size in the block algorithm. Additionally, such scheme also enables straightforward parallelization since the subproblems can be computed by multiple threads.

Since we use the classic algorithm in the final stages of recursion, we can also try transposing the matrix first as in Section 2.1. This can be also written in the form of $C = A \times X$ where $X$ represents transposed matrix $B$ and $\times$ corresponding transposed product. At this point we write block partition in different style to avoid confusion with untransposed version of algorithm. So we can partition matrices $A$ and $X$ on blocks as

$$A = \begin{bmatrix} A_1 & A_2 \\ B_1 & B_2 \end{bmatrix}, \quad X = \begin{bmatrix} X_1 & X_2 \\ Y_1 & Y_2 \end{bmatrix}.$$

And the resulting matrix $C$ can be computed as

$$\begin{aligned} C_{1,1} &= A_1 \times X_1 + A_2 \times X_2 \\ C_{1,2} &= A_1 \times Y_1 + A_2 \times Y_2 \\ C_{2,1} &= B_1 \times X_1 + B_2 \times X_2 \\ C_{2,2} &= B_1 \times Y_1 + B_2 \times Y_2. \end{aligned}$$

In Figure 4 we compare these two versions. The blocks now fit in the cache and the gains from memory access patterns withing blocks are not as significant as in the completely classical approach.

### 3.1 Implementation Details

It is important to note that recursion in this algorithm is only used to control the flow of computation. No matrices are copied or created

in the recursive step as all operations happen in-place.

To select the point where the algorithm switches from recursive to classic multiplication we parametrized the switch point and ran scripted tests to find the optimal block size. This number was different across computers. The computer we used for testing performed best if the switch to classic algorithm happened for blocks smaller than $22$ in any dimension.

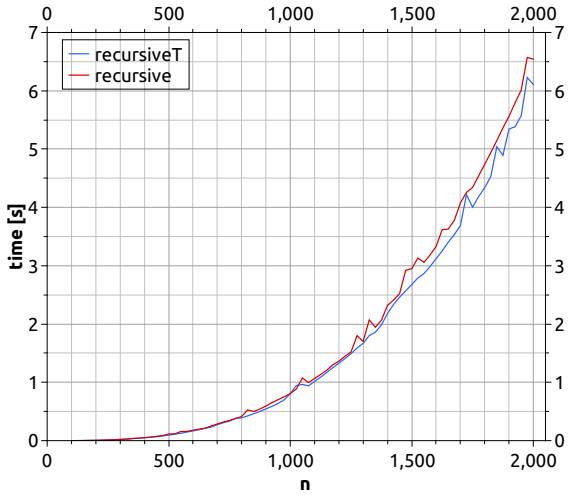


Figure 4: Running-time comparison of recrusive multiplication algorithms: without (recursive) and with (recursiveT) transposition.

## 4. Strassen Algorithm

The Strassen algorithm can be derived from the recursive algorithm, as it is only an additional idea on the top of the recursive algorithm. The idea is to get the block matrices of matrix $C$ with less block products then in the recursive algorithm i.e., less than 8.

So for recursive multiplication $C = AB$ we can define auxiliary matrices as

$$
\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}
$$

and compute sub-matrices of C as

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

Which yields the same result as the recursive algorithm. In recursive algorithm we need $8$ multiplication of submatrices of $A$ and $B$ but in the Strassen algorithm we need only $7$ multiplications of submatrices of the same size. Additionally, recursive algorithm requires $4$ matrix additions and Strassen requires $12$ matrix additions, and $6$ matrix subtractions. Since multiplication is asymptotically harder than summation or subtraction this drastically improves the theoretical time complexity of the strassen algorithm. Strassen showed that his method has time complexity $\Theta(n^{\log_2(7)})$ or $\mathrm{O}(n^{2.8074})$ [14].

### 4.1 Transposed Strassen-like Algorithm

Our implementation of a subcubic algorithm can be derived from recursive transposed algorithm similarly to how the Strassen algorithm can be derived from the recursive algorithm. Since we did the derivation from recursive transposed algorithm on our own, its blocks are different from the ones in the Strassen algorithm. There exists many Strassen-like algorithms that take the same idea as the Strassen algorithm, but come up with different block organizations. Our goal was to combine the idea of transposing one of the matrices before multiplication with the subcubic nature of Strassen algorithm.

For the transposed recursive multiplication $C = A \times X$ we can define submatrices of $C$ as

$$
\begin{aligned}
C_{1,1} &= M_4 + M_5 \\
C_{1,2} &= M_2 - M_6 + M_3 - M_5 \\
C_{2,1} &= M_1 - M_4 - M_3 - M_7 \\
C_{2,2} &= M_6 + M_7
\end{aligned}
$$

where auxiliary matrices $M_i$, where $i \leq 7$, are

defined as

$$
\begin{aligned}
M_1 &= (A_1 + B_1) \times (X_1 + Y_1) \\
M_2 &= (A_2 + B_2) \times (X_2 + Y_2) \\
M_3 &= (A_1 - B_2) \times (X_2 + Y_1) \\
M_4 &= A_1 \times (X_1 - X_2) \\
M_5 &= (A_2 + A_1) \times X_2 \\
M_6 &= B_2 \times (Y_2 - Y_1) \\
M_7 &= (B_1 + B_2) \times Y_1
\end{aligned}
$$

In Figure 5 we compare our implementations of the Strassen algorithm and its recursive counterpart. As with the recursive algorithm the time gain is not significant. This is because the algorithms switch to the recursive algorithm when block size is lower than 96. As before this threshold was determined experimentally. Since small blocks fit in the cache, memory access patterns do not affect performance much.

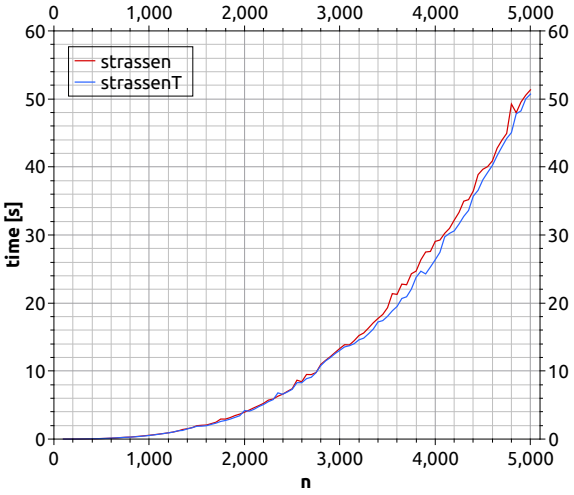


Figure 5: Running-time comparison of Strassen multiplication algorithms: without (strassen) and with (strassenT) transposition.

## 5. Numerical Stability

We also compared numerical stability of our implemented algorithms. For this test we changed our algorithms to perform calculations using single-precision floating-point arithmetic (float). We generated two random matrices and multiplied them. We also multiplied the same matrices using double-precision in uBLAS. We then rounded the uBLAS result back to a float and calculated the error estimate as follows

$$\max_{i,j} |(C_{\text{blas}} - C_{\text{test}})_{i,j}|.$$

Here we assume that the result computed with double precision is correct for all the digits representable in single precision floating point numbers.

The results in Figure 6 show that the recursive algorithm is the most numerically stable algorithm, followed by classic algorithm and the Strassen algorithm. The transposed Strassen algorithm performs the worst in this test. Although the test shown in Figure 6 shows values only for one particular pair of random matrices, testing with different random matrices produces similar results.

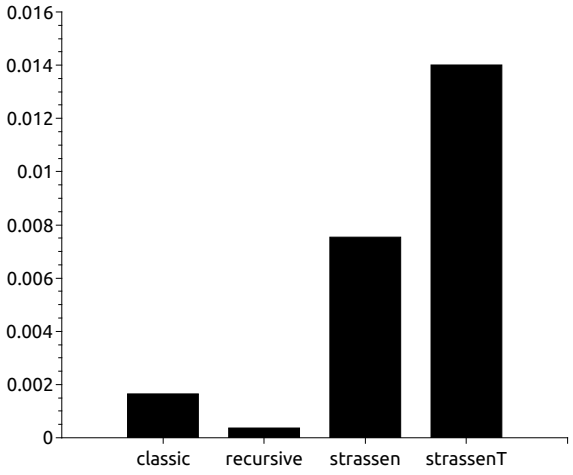


Figure 6: Maximum absolute error comparing single-precision computation for a $2000 \times 2000$ matrix multiplication with a double-precision computation on the same matrix.

The Classic algorithm is not perfectly numerically stable, because adding arbitrary floating point numbers will usually give some rounding error, which will be proportional to the size of the result. We can accomplish better results by applying some smart and expensive numerically stable algorithm like Kahan summation [10] or using a recursive strategy where we first sum pair-wise elements of the vector and continue the procedure until only one element remains.

The recursive algorithm's way of summation is most similar to recursive summation,

however our implementation violates the recursive algorithm scheme by:

- stopping the recursion early and performing the classic multiplication algorithm for sufficiently small blocks and
- adding the values from the small blocks calculated with the classical algorithm directly to the corresponding matrix entries and not first adding the blocks together pair-wise.

This two differences both lower the numerical stability compared to a more strictly implemented recursive algorithm.

Strassen and Strassen-like algorithms are less numerically stable, because if we expand the formula for one matrix entry we see that in the Strassen algorithm the formula is much larger. Some elements are first added and then subtracted, which reduces numerical stability.

## 6. Experimental Comparison

### 6.1 Implementation

We implemented the algorithms in C++. We tried to optimize each algorithm as much as possible. Although all our comparisons were performed on square matrices we also tested correctness on non-square matrices. Implementation of our algorithms is available on Github at `https://github.com/mihic/matrixmul`.

### 6.2 uBLAS Library

In order to compare our implementations with an existing state-of-the-art implementations we chose uBLAS, the implementation of matrix multiplication from the Boost library in the C++ programming language. We chose it for its ease of use and simplicity. Other libraries such as Eigen [3] or Intel Math Kernel Library [4] are vigorously optimized in order to be able to use multiple cores and advanced vector instructions of modern processors. We chose uBLAS since our goal was to compare generic optimization techniques, not specific to particular processor architecture.

We used the `block_prod` function from `boost::numeric::ublas` with a block size of 32. As before we determined the optimal block size experimentally.

### 6.3 Testing Environment

All our tests were run on a computer with 16 GBs of RAM on Intel's Core i7-6700HQ locked to 3.1GHz. The operating system was Arch Linux using kernel version 4.13. We used *gcc* version `7.2.0`, with the following compilation flags: `-O3 -march=native -DNDEBUG -DBOOST_UBLAS_NDEBUG`. `O3` and `march=native` flags enable compiler optimizations. The other two flags disable checking for errors inside the uBLAS library. Without those flags the uBLAS library checks the numbers in the calculation and warns the user if it finds that the computation is loosing too much precision. The flags disable this overhead and significantly improve the performance of uBLAS.

All the compared algorithms used a single thread of execution and calculated a product of two $n \times n$ randomly generated matrices.

### 6.4 Results

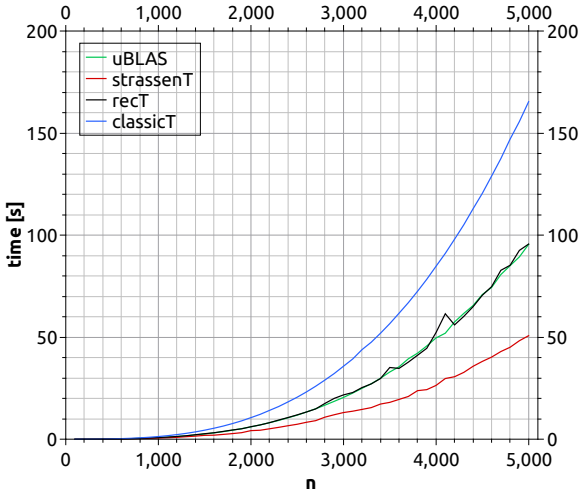


Figure 7: Running-time comparison of the faster version of discussed algorithms, including the uBLAS library for comparison.

In Figure 7 we compare the faster versions of all described algorithms. As expected transposed classic algorithm is the slowest, followed by transposed recursive and uBLAS. Since uBLAS uses a block algorithm we can

confirm our prediction, that recursive algorithm should behave similarly to the block multiplication algorithm. The best algorithm in this comparison is transposed strassen. Although this was expected for large $n$, these results show that the transposed Strassen algorithm is faster even for smaller $n$.

## 7. Conclusion

We described several practical matrix multiplication methods and discussed their implementation details, numerical stability and performance. We can conclude that the Strassen-like algorithm does not have a significant constant hindering its performance for practically sized matrices. This would suggest that this method could be implemented even in general purpose libraries with the only downside being lower numerical stability. The library could provide a fast Strassen based algorithm and a fall-back recursive algorithm for cases when numerical stability is very important.

Our work could be extended to include processor specific optimizations such as SIMD (Single Instruction Multiple Data) instructions. This would enable a comparison of the Strassen algorithm with highly optimized libraries such as Eigen or Intel MKL.

## Acknowledgements

The authors would like to thank Jurij Mihelič, because our work began as an assignment during his MSc course *Algorithm Engineering* and he encouraged and helped us expanding it into an article.

## References

[1] Armadillo C++ linear algebra library, October 2017. http://arma.sourceforge.net/.

[2] Boost C++ libraries. http://www.boost.org/, October 2017.

[3] Eigen C++ template library for linear algebra, October 2017. http://eigen.tuxfamily.org.

[4] Intel math kernel library, October 2017. https://software.intel.com/en-us/mkl.

[5] Dario Bini, Milvio Capovani, Francesco Romani, and Grazia Lotti. $O(n^{2.7799})$ complexity for n× n approximate matrix multiplication. *Information processing letters*, 8(5):234–235, 1979.

[6] Jurij Mihelič and Uroš Čibej. Experimental algorithmics for the dataflow architecture: Guidelines and issues. *IPSI BgD Transactions on Advanced Research*, 13:1–8, 2017.

[7] Don Coppersmith and Shmuel Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[8] Alexander M. Davie and Andrew James Stothers. Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh Section A: Mathematics*, 143(2):351–369, 2013.

[9] Blagojevic V. et al. A systematic approach to generation of new ideas for phd research in computing. *Advances in Computers*, 104:1–19, 2016.

[10] William Kahan. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40, 1965.

[11] Donald E Knuth. The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2, 1997.

[12] Catherine C. McGeoch. A guide to experimental algorithmics. 2012.

[13] M. Müller-Hannemann and S. Schirra. Algorithm engineering: bridging the gap between algorithm theory and practice. *Lecture Notes in Computer Science*, 2010.

[14] V. Pan. Strassen algorithm is not optimal. trilinear technique of aggregating, uniting, and canceling for constructing fast algorithms for matrix multiplication. In *Proc. 19th Annual Symposium on the Foundations of Computer Science*, pages 166–176, 1978.

[15] Francesco Romani. Some properties of disjoint sums of tensors related to matrix multiplication. *SIAM Journal on Computing*, 11(2):263–267, 1982.

[16] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3-4):281–292, 1971.

[17] Volker Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[18] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[19] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd [extended abstract], STOC 12 — Proceedings of the 2012 ACM Symposium on Theory of Computing, ACM, New York, 2012.