Final Project: Matrix Multiplication

Jonas Balin (jbal@itu.dk)
Luna Berthelsen (lube@itu.dk)
Emily Crome (ecro@itu.dk)

KSAPALG1KU

2021-12-17

1 Introduction

Matrix multiplication is a binary operation which produces a new matrix from two other matrices. Matrix multiplication is a basic tool in linear algebra, and has many applications in mathematics [1].

The definition of a (square) matrix product $C = AB$ is as follows, given two matrices A and B of size n x n, such that

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1}$$

for all $i, j \in \{1, 2, ..., n\}$ [3].

In this report, several different versions of matrix multiplication algorithms are implemented and compared.

2 Implementation

The algorithms were implemented in Java openjdk version 11.0.12, using the provided `Matrix.java` library stub, the data type `double`, and the build tool Gradle version 7.1.1.

All algorithms were tested for correctness using the unit testing framework `JUnit Jupiter`. Two sizes of input matrices were used to test most algorithms, with side length $n = 256$ and $n = 1024$. Input arrays were created with random integers, using the Java library `Random`, in the range of -10 to 10. Four different types of matrices were used; only positive integers, only negative integers, the same integer and mixed positive and negative integers. The implemented algorithms were tested against the elementary matrix multiplication algorithm using the `equals` method provided in `Matrix.java`.

The following subsections report on which algorithms were implemented. It was assumed for this project that all matrices are square, with a side length n a power of 2.

2.1 Task 1 - Elementary Matrix Multiplication

The elementary matrix multiplication algorithm was adapted from a method which passed on CodeJudge, using three nested for-loops and the `get` and `set` methods in `Matrix.java`. There are two implementations of this algorithm, one copying and one in-place.

2.2 Task 2 - Matrix Transposition

The transposition method was created using two for-loops. One for columns and one for rows. Within the inner loop, the values at A_{ij} and A_{ji} were switched with each other, effectively transposing the matrix.

2.3 Task 3 - Transposed Matrix Multiplication

The assumption of one input-matrix transposed was achieved by a call of the `transpose()` method (2.2) on the given matrix B. Following that, the elementary multiplication algorithm (2.1) was used, with the slight difference that both matrices are accessed sequentially.

2.4 Task 4 - Tiled Matrix Multiplication

The tiled matrix multiplication method was developed following pseudo-code presented in the lectures [4]. In practice, this manifested in three nested for-loops. In the inner loop, the two sub-matrices of inputs A and B are defined, and a slightly modified version of the in-place matrix multiplication is called. The modification served the purpose of making the in-place multiplication additive, such that the values in the sub-matrices of C would not get overwritten.

2.5 Task 7 - Recursive Matrix Multiplication, Copying

The copying variant of the recursive algorithm was implemented by explicitly following Equations (3) and (4) from [3]. In other words, each intermediate product was computed and eight recursive calls performed, before the products were merged and the result matrix C returned.

2.6 Task 8 - Recursive Matrix Multiplication, Write-through

As the intermediate results of the copying variant of the recursive matrix multiplication (2.5) might make the algorithm unnecessarily slow, a write-through version of the recursive matrix multiplication was implemented as well. The algorithm was defined to fall back to the in-place version of the elementary algorithm (2.1) at a particular subproblem size, m.

2.7 Task 10 - Strassen's Algorithm

The implementation of Strassen's algorithm was made with the copying variant of the recursive matrix multiplication (2.5) as the starting point, and altering it according to [5]. A parameter m was included, so that the algorithm falls back to the elementary multiplication in-place (2.1) when the sub-matrix reaches the size of m.

3 Experiments

All the experiments were run on a Lenovo Yoga Slim 7, using an AMD Ryzen 7 4700U, 2.0 GHz CPU w/ 8 cores. 3200Mhz DDR4 RAM, running on Windows 10 OS. The Java code was executed in the terminal running openjdk version 11.0.12.

3.1 Task 5 - Tiled Matrix Multiplication Experiments

In the ideal cache model, the CPU cache has a total amount of storage, M (measured in words). This storage is divided into lines of length B, such that there are $\frac{M}{B}$ lines in total. The ideal cache model makes some simplifying assumptions, namely optimal replacement, full associativity, and tall cache.

The ideal cache model will be used as the theoretical foundation to fix n to a suitable, sufficiently large power of 2 that does not fit within the cache. To find the lower bound for n, we must first identify the side-length of our sub-matrices, s, that should fit within the cache. As the sub-matrix size is $s \times s$, the largest s that would fit within the cache is $\sqrt{M}$.

To calculate our choice of n, the following variables were used[1]:

$$\begin{aligned}
\text{Total Cache Size (bytes): } C &= L1 + L2 + L3 \\
\text{Total words: } W &= C/8 \\
\text{Sub-matrix area : } M &= W/3 \\
s &= \sqrt{M} \\
n &= 2s
\end{aligned}$$

[1]In this interpretation of the ideal cache model, it was chosen to add the three CPU caches together and assume a uniform R/W speed of the L3 cache. Furthermore, the sub-matrix area was divided by 3, as storage for three sub-matrices in the cache A, B and C, was needed.

Based on the aforementioned machine specifications, this lead to the following calculation:

$$\begin{aligned} C &= 512000 + (4e+6) + (8e+6) = 12512000 \\ W &= \frac{12512000}{8} = 1564000 \\ M &= \frac{1564000}{3} = 521333.33 \\ S &= \sqrt{521333.33} = 722.03 \end{aligned}$$

The ideal s was therefore found to be roughly 722. However, given the square matrix assumption, s must be a power of 2, and the power of 2 that comes closest to 722 (without exceeding it) is 512. This was therefore found to be the theoretical optimal s based on the ideal cache model for the tiled matrix algorithm and the given machine. In other words, three sub-matrices of s^2 area should fit within the CPU's cache, following the ideal cache model.

Following from this, as n must be at least $2 \times s$, the suitable, sufficiently large n was found to be $512 \times 2 = 1024$. As n can grow in proportion with powers of 2, it was concluded that the optimal size of s allows for $n = 2^x$, where $x \geq 10$.

However, an experiment was performed to test this theoretically optimal s. n was set to 1024 and the running times of the tiling algorithm with s ranging from 2 to $\frac{n}{2}$ were measured. The quickest running time was not found to be $s = 512$, but $s = 8$. The slowest running time was at $s = 2$. Generally, the running times varied greatly depending on the size of s (all running times are found in Table 1). Thus while the ideal cache model might be a useful abstraction, in practice many variables can influence the outcome. A discussion of this will follow in the next subsection.

3.2 Task 6 - Tiled Matrix Multiplication, Cont.

The experiment from 3.1 were also run with n size 512 and 2048 (half and double the suitable n found in 3.1). It was found that no matter the size of n, $s = 8$ produced the fastest running times. The different running times for varying sizes s can be found in Table 1.

Given that the running times indicate the ideal s is 8 rather than 512, it can naturally be concluded that there is a discrepancy between the theoretical calculations based on the ideal cache model and practice. The ideal cache model, as it name implies, belies an idealized model of the CPU cache. The discrepancy can perhaps be caused by modern multitasking systems complicating matters. The ideal sub-matrix cache calculations assume that the CPU cache is 100% available, but this is an idealized assumption that may not hold. An OS like Windows 10 will always have background processes running to ensure operability of the computer, and it is unclear to what extent these processes occupy cache space. Other explanations could point to the specifics of the CPU used. The CPU cache is not one level but three different levels of varying sizes and orders of magnitude differing R/W speeds. Attempts to account for the levels of caches do not fully explain the discrepancy either. Conducting the calculation for s with only the L1 cache gives $s = 64$, still several times larger than $s = 8$. In conclusion, due to the wealth of unknown and (in our experiments) uncontrollable factors, it cannot be safely determined why the practical results differ from the theoretical expectation.

Table 1: Different running times (in milliseconds) obtained with varying submatrix sizes, s, for the tiled matrix multiplication algorithm with three values of n.

	n		
s	512	1024	2048
2	345	19060	161658
4	162	3659	31859
8	151	1577	12230
16	274	2339	17203
32	364	2927	26936
64	384	3511	39467
124	425	4918	40101
256	606	5110	41355
512	-	5318	97998
1024	-	-	87932

3.3 Task 9 - Recursive Matrix Multiplication Experiments

Following the calculations from 3.1, n was set to 1024. Similar experiments to the ones done for 3.1 were then performed and it was found that the algorithm should fall back to the elementary algorithm at $m = 8$. The results are presented in Table 2. This corresponds to the experiments with the tiled matrix multiplication in which the subproblem size of 8 was similarly the most effective. There can be different reasons for this, but the intuitive answer is that a subproblem size of 8 may be contained within the cache entirely, such that the speed increases significantly from subproblem sizes with higher rates of cache-misses. This aligns nicely with expectations drawn from the tiling multiplication results, as the cache complexity for the recursive algorithm is the same as for the tiling algorithm [4].

Table 2: Different running times (in milliseconds) obtained by different subproblem sizes, m, for the write-through variant of the recursive algorithm with $n = 1024$.

m	running time
0	33974
2	8544
4	2754
8	1465
16	2328
32	2972
64	3309
128	4731
256	5068
512	5289

3.4 Task 11 - Strassen's Algorithm Experiments

For the implementation of Strassen's algorithm, n was again set to 1024 following the calculations from 3.1. It was found that the fastest running time was obtained at $m = 32$. All running times can be found in Table 3. Compared to the two other algorithms, tiling matrix multiplication and write-through recursive matrix multiplication, Strassen's algorithm store many intermediate results which require auxiliary space. This might explain why Strassen's needs to fall back to an elementary algorithm sooner than e.g. the recursive write-through algorithm. The overhead from the increased number of additions dominates the workload and might make Strassen's inefficient for submatrices smaller than 32.

Table 3: Different running times (in milliseconds) obtained with different subproblem sizes, m, for Strassen's algorithm with different sizes of n.

	n		
m	512	1024	2048
0	3627	20457	140168
2	1129	5547	38490
4	346	1934	13457
8	160	1074	7171
16	109	778	5169
32	111	629	4391
64	101	694	4834
128	147	1049	7241
256	324	2149	15591
512	-	4654	35840
1024	-	-	76908

3.5 Task 12 - Horse Race

For task 12, a horse race was performed between all the different algorithms implemented: the elementary matrix multiplication; the elementary multiplication with a transposed operand (B); the tiled algorithm with $s = 8$; the copying recursive algorithm; the write-through recursive algorithm with $m = 8$; and Strassen's algorithm with $m = 32$.

For the benchmarking, the toolchain from the 3SUM assignment [2] was adapted. All algorithms to be benchmarked were overloaded to make them fit the single-argument requirement of the `Consumer<T>` given as parameter to the benchmarking algorithm. Then each algorithm was benchmarked with a total of three runs per input size to ensure that the cache was initialized the same way in each experimental evaluation. The results of the final run were plotted in Figure 1.

The only deviation from this scheme was the benchmarking of the transposed variant of the elementary algorithm. Here, first the time to transpose the operand was measured, then the running time of the multiplication itself. In Figure 1 only the running time of the multiplication itself is reported. The running times of transposing and multiplying can be found in Table 4.

An algorithm to generate input for the benchmarking was computed as well. This algorithm would compute matrices with the sidelength n given as parameter. In order to avoid rounding errors, all matrices were filled with integers in the range -500 to 500. These were generated using a random

Table 4: Time required for the transpose of the operand and time required for the multiplication. Both for the transposed matrix multiplication.

n	multiplication	transpose
16	0.000000000	0.0000000
32	0.000000000	0.0000000
64	0.000000000	0.0000000
128	0.000000000	0.0000000
256	0.012516400	0.0000000
512	0.093820400	0.0000000
1024	0.869620700	0.0080568
2048	6.936553300	0.0402641

number generator (the Java library `Random`) with seed 10. In theory, smaller matrices would be able to hold larger integers and still avoid rounding errors, but it was decided to run with very small integer values to be on the safe side.

All in all, Strassen's algorithm performed the best for the larger sizes of n. The copying version of the recursive algorithm performed the worst on all input. The elementary algorithm performed decently on smaller input, but did not scale well. While the write-through version of the recursive algorithm performed worse than the tiling algorithm, the two ended up with almost identical running times for input larger than $n = 1024$. Perhaps surprisingly, the algorithm assuming a transposed operand scaled well. As this was the algorithm with the best performance apart from Strassen, it would be an interesting future experiment to compare their running times on even larger inputs. Furthermore, as transposing takes time in itself, it might also be interesting to compare the two where the time to transpose is considered part of the actual running time.

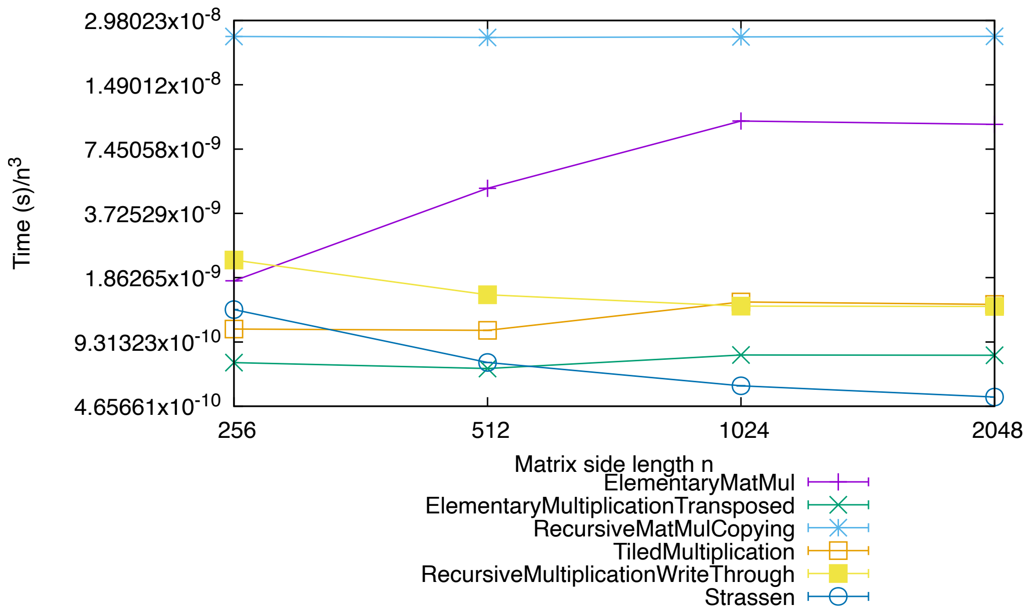

Figure 1: Horse race. Running times of all algorithms normalized by n^3.

References

[1] Matrix multiplication. https://en.wikipedia.org/wiki/Matrix_multiplication.

[2] Rico Jacob and Matti Karppa. *Instructions, Assignment 1: 3SUM*, 2021.

[3] Rico Jacob and Matti Karppa. *Project instructions, Final project: Matrix Multiplication*, 2021.

[4] Matti Karppa. Applied algorithms 9: Matrix multiplication 2. ITU Lecture, 11 2021.

[5] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math*, 13, 1969.

A Appendix

The code as well as all CSV files, plots and tables generated for this project are submitted along with this report. To run the code, uncomment the sections corresponding to the task you want to run in `main` (these are clearly marked).