Assignment 4: Matrix Multiplication

Johannes Rostad Bertoft, Daniel Estehghari

December 16, 2021

1 Introduction

This report analyses the implementation of a number of different algorithms to solve Matrix Multiplication. In broad terms, Matrix Multiplication involves finding the product C of two operand matrices A and B. This report concerns square matrices only; in other words, matrices of shape $n \times n$. More specifically, the product $C = AB$ will satisfy

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

for all $i, j \in 1, 2, ..., n$.

The Matrix elements in this implementation uses double precision floating point numbers (64-bit), and in addition to being square matrices, the implementations here assume that side length n of any given matrix will be some power of 2.

Seven different algorithms are implemented to compute Matrix Multiplication: `Elementary, Elementary Transposed, Tiled, Recursive Copying, Recursive Write Through, Strassen,` and `Strassen Parallel`[1] (a variant of `Strassen` using parallelisation). The algorithms listed have an asymptotic running time of $\Theta(n^3)$ except for `Strassen` which has a running time of $\Theta(n^{2.81})$.

This report uses the Ideal Cache Model to discuss the behavior of the algorithms for varying sizes of n. The Ideal Cache Model is a simplified model of the memory hierarchy, consisting of two levels, the "faster" cache, and the "slower" main memory. The model assumes that the CPU reads

[1]This implementation is not based on any standard implementation and may therefore not be optimal in any way

from cache at 0 cost. If the data required is found in cache, it is a cache hit. Otherwise it is a cache miss, and data needs to be transported from main memory at a cost of 1 (per cache line). Cache in this model consists of Z words, and the cache line (the data moved from main memory to cache at one time) is of L words, with a total of Z/L cache lines.

Furthermore, the model assumes a **fully associative cache**, meaning memory blocks can be allocated to any cache line, **perfect replacement strategy**, and a **tall cache** indicating that $Z = \Omega(L^2)$. Performance analysis consist of analysing the work (number of operations), performed by each algorithm, denoted $W(n)$, as well as the cache complexity $Q(n; Z, L)$. Cache complexity is the number of cache misses incurred during execution, in terms of n, Z, L.

2 Implementation

The Matrix class implementation is based on a library stub written in native Python3 code, using standard library functions and a flat list in row-major order. All algorithms utilises this class to compute Matrix Multiplication, and similarly uses only standard library functions. Auxiliary functions for Matrix addition and subtraction, as well as `transpose`, `split` (splitting a Matrix into four equally sized submatrices), and `elementary multiplication in place` have been implemented. The full implementation can be found in `MatrixList.py`. `Strassen Parallel` is the only implementaion with parallelisation, using Python's multiprocessing module to split the problem into seven subproblems $M_i = P_iQ_i$ for $i \in 1, ..., 7$ and with a `Pool` of 4 workers (subprocesses), executing the subsequent `Strassen` function calls recursively before joining the result of the product C.

3 Input Generation

In order to avoid rounding-errors resulting from floating-point arithmetic, the algorithms were evaluated on input matrices consisting of $n \times n$ randomly generated integers (represented as floats), where each integer k was sampled according to $\{k \in \mathbb{Z} \mid -x < k < x\}$, $x = \sqrt{\frac{2^{53}}{n}}$ [2]

[2]the value for x is derived from the formula $nx^2 = 2^m$, where m denotes the number of bits in the mantissa for a double precision floating-point number (64) and nx^2 denotes the highest encounterable value when multiplying two matrices of $n \times n$ integers where, for each integer k, $k < |x|$

4 Correctness Testing

The `Elementary` algorithm was tested for correctness by tests provided by CodeJudge. After it had been verified correct, the subsequent algorithms were tested for correctness against the elementary algorithm. This was done by first generating two sets of 20 small ($n = 8$) matrices A and B, each matrix consisting of randomly generated integers according to section 3. The product $C_i = A_i \times B_i$ was then computed, for each matrix pair $\{A_1, B_1 \dots A_{20}, B_{20}\}$ with the `Elementary` algorithm and each result was compared element-by-element with the corresponding result yielded by each subsequent algorithm to assert equality. The procedure was then repeated with two sets of 20 larger ($n = 512$), similarly generated matrices A and B.

5 Experiments

Experiments are run on a MacBook Air 13 inch (2016 model) with a 1.6 GHz processor, Dual-Core Intel Core i5, with 8GB of RAM and Cache sizes L1 = 64 KB (32 KB instruction set, 32 KB data set), L2 = 512 KB (256 KB per Core), L3 = 3 MB, amounting to a total cache size of 3.544 MB. Experiments were performed using PyPy (7.3.2 with GCC Clang 10.0.1), an alternative to CPython, which uses a just-in-time compiler.

5.1 Choosing the optimal parameter

The first set of experiements concern the optimal parameter used for the `Tiled`, `Recursive Write Through` and `Strassen` algorithms. To deduce some cache behavior, sizes of the matrices to benchmark are chosen with regards to what in theory would fit in cache. Given the total cache size of 3.544 MB and assuming a word size of 8 Bytes (B), corresponding to a 64-bit double precision float (64-bit), this means that in the Ideal Cache Model scenario, we can calculate Z, the number of words that fit in cache,

$$\begin{aligned} Z &= 3.544 \text{ MB} \\ &= 3\,544\,000 \text{ Bytes} \\ &= \frac{3\,544\,000}{8} \text{ words} \\ &= 443\,000 \text{ words} \end{aligned}$$

This entails that $\sqrt{Z} \approx 666$ is the side-length of a Matrix operand that would fill the entire cache. Matrices for the experiment are therefore of sizes

$n \in \{256, 512, 1024, 2048\}$, to allow for both when operands do, and do not, fit in cache. The experiments tests parameters[3] in the range $2, ..., n/2$.

5.1.1 Tiled Algorithm

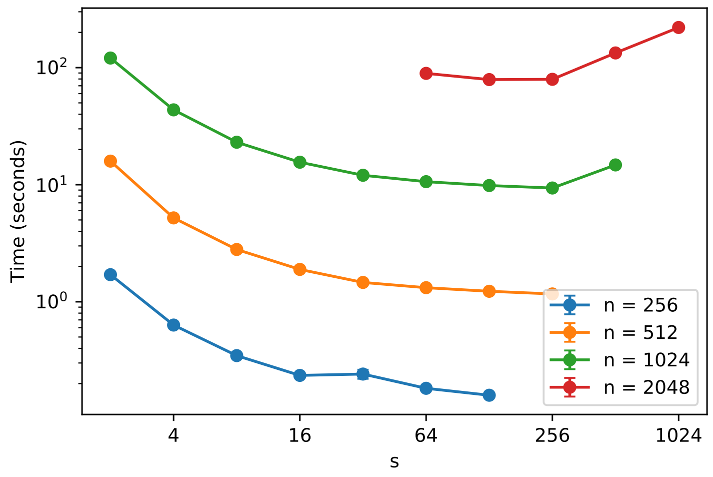

Figure 1: Average runtime (in seconds) for Tiled Matrix Multiplication with different values for parameter s.

The `Tiled` algorithm is cache-aware meaning it should be tuned according to the size of the cache, where the adjustable parameter s controls the size of the submatrices. Just like the `Elementary` algorithm, it has $W(n) = \Theta(n^3)$. However, we would expect that for larger values of n (where $n^2 > Z$), there will be an optimal value for the submatrix size $s \times s$ where submatrices just fits into cache, resulting in fewer cache misses and shorter running time in comparison. With optimal $s = \Theta\sqrt{Z}$ cache complexity for the `Tiled` algorithm is $Q(n) = \Theta(\frac{n^3}{\sqrt{Z}L})$.

As we can see, for $n = 256$ and $n = 512$, running time decreases steadily as s increases. For $n = 1024$ and $n = 2048$ however, running time reaches a minimum for $s = 256$ and then increases for $s = 512$.

Since the above analysis is based on the Ideal Cache Model but in reality the experiments are run on a multi-purpose computer, it is not surprising that the optimal s seems to be a bit lower in the experiments, as there can be

[3]Due to time limitations, $n = 2048$ was only evaluated for $s, m \geq 64$

several processes utilizing the cache simultaneously, and naturally a smaller part of the cache would be allocated to running the program.

5.1.2 Recursive Write Through Algorithm

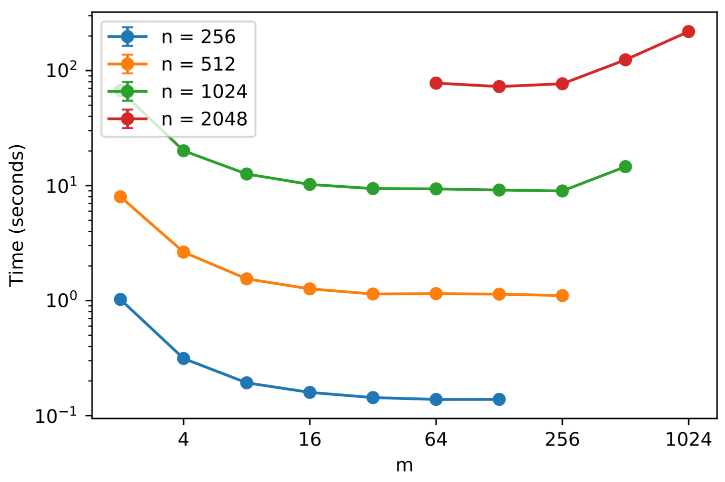

Figure 2: Average runtime (in seconds) for Recursive Matrix Multiplication with different values for parameter m.

For the `Recursive Write Through` algorithm, the adjustable parameter (m) controls the sub-problem size at which the recursion should stop and the algorithm should fall back to the elementary algorithm. As the recursive algorithm is cache oblivious in the Ideal Cache Model, we would – in theory – expect that m is irrelevant; setting $m = 0$ would mean that the parameter isn't used and that the algorithm recurses all the way down to the base case where scalar multiplication is performed when $n = 1$, which should be optimal regardless of cache size, with a cache complexity of $Q(n) = \Theta(\frac{n^3}{\sqrt{Z}L})$.

In practice however, we are not operating under these conditions, and overhead from recursion depth will have an effect on runtime. This entails that making a late fallback to the elementary algorithm (low m) becomes costly, and for $n^2 > Z$ optimal performance will thus rather be achieved by setting m so that the elementary algorithm is performed when the sub-operands just fit in cache. Setting m lower than this will just incur unnecessary overhead. This is a plausible explanation for why it is only for $m \geq 256$ that we start to see a decrease in performance for increasing m further.

5.1.3 Strassen's Algorithm

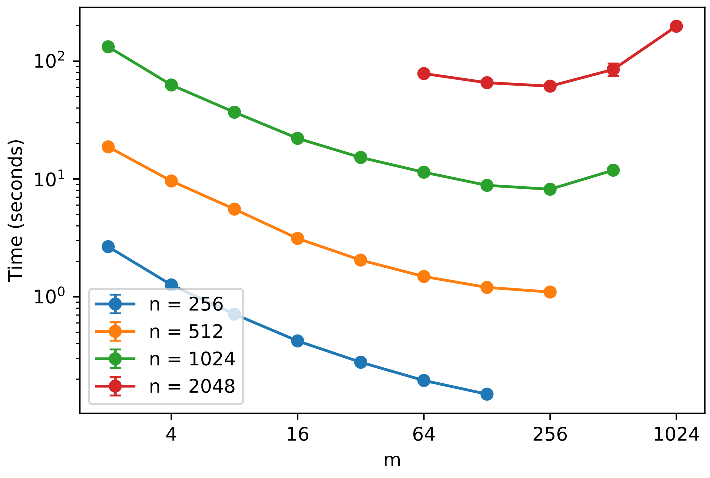

Figure 3: Average runtime (in seconds) for Strassen's Algorithm with different values for parameter m.

For `Strassen`, we see a similar pattern with respect to the adjustable parameter (m), which here also controls the subproblem size for which the algorithm should fall back to performing elementary multiplication. Again, the behavior predicted by the Ideal Cache Model only begins to partly show when n is sufficiently high (and the elementary algorithm cannot fit the operands of the full problem neatly in cache). Optimal performance thus again becomes a trade-off between the cost of recursion overhead and fitting operands in cache. The cost of recursion makes an early fallback ($m = n/2$) optimal for $n \leq 512$, whereas for $n \geq 1024$, cache considerations appear to yield minimal runtime when $m = 256$.

5.2 Horse Race

The results in Figure 4 from the final experiment shows the behavior of all algorithms for varying size $n \times n$ matrices, using the optimal parameter for the `Tiled`, `Recursive Write Through` and `Strassen` algorithms found in Section 5.1. For `Strassen Parallel` the same parameter was chosen. The optimal parameter in all cases was 256 for larger matrices. For the matrices where $n \leq 256$ parameters should be lower, and was therefore set to $\frac{n}{2}$ as this proved optimal in those cases. The results are normalized by dividing average time with n^3 to get runtime in terms of operations $W(n) = \Theta(n^3)$. Note that since the `Strassen` variants are $\Theta(n^{2.81})$ this is not completely accurate in terms of operations, but allows comparison of runtime against the other algorithms.

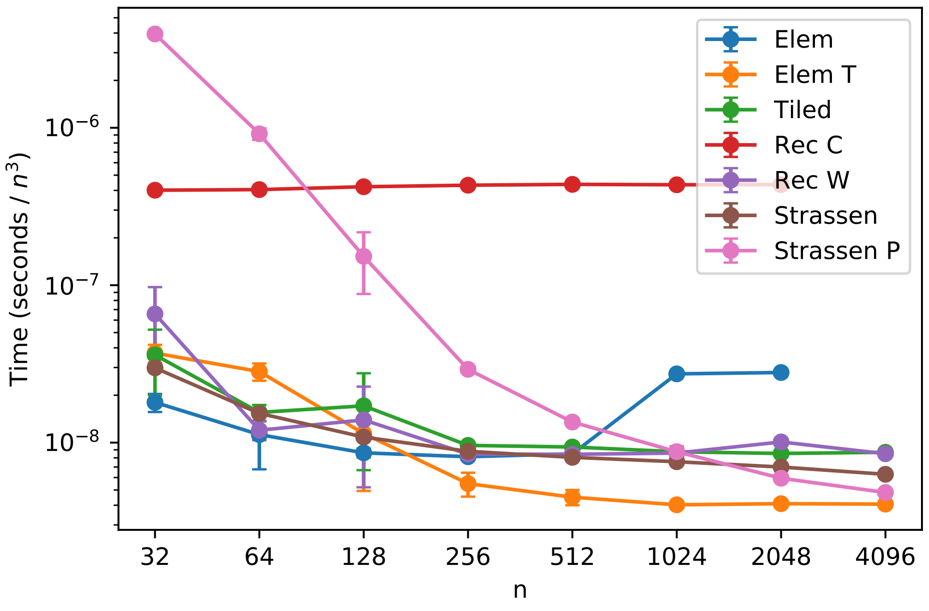

Figure 4: Average normalized time per operation (average time divided by n^3) for Matrices of varying sidelength n

The `Recursive Copying` algorithm is by far the slowest of the bunch. It also has the property that its time per operation is near constant across varying n, indicating that it is cache-oblivious, in the sense that it doesn't depend on cache size, compared to for instance the `Elementary` algorithm. However, it only performs scalar multiplication once $n = 1$ in the recursion, and likely, the resulting recursion call overhead makes it slow in comparison.

The `Elementary` algorithm clearly shows that when matrices get larger, it becomes slower per operation, in line with the Ideal Cache Model, indicating that it incurs more cache misses. Based on the results, it seems to be closely related to the optimal parameter discussed in Section 5.1 where the Matrix operands fit in cache. The `Tiled` and `Recursive Write Through` have similar patterns, with decreasing runtime per operation until $n = 256$ and subsequently evening out for larger matrices. However both also see a slight increase from $n = 64$ to $n = 128$, perhaps this could be due to cache in reality being multi-level and for rather small matrices they could fit in a faster level of cache, which is not accounted for in the Ideal Cache Model.

`Strassen` shows a trend with more steep decrease until $n = 256$ and onwards keep decreasing slightly. Since y-axis is normalized with based on $\Theta(n^3)$ it is natural that it doesn't even out in the same fashion as the previous two algorithms, because of its better asymptotic runtime. All these three algorithms tuned with an optimal parameter surpass the `Elementary` algorithm in speed approximately when $n \geq 512$. `Strassen Parallel` also show a steep downward trend, and actually manages to surpass all other algorithms except for `Elementary Transposed`, when matrices get large enough. It is not surprising that it has a slow start, since initializing subprocesses is costly. It is first when it is allowed to utilize the parallelization for a longer period that it manages to catch up with, and surpass, the other algorithms.

Finally, the "winner" turns out to be `Elementary Transposed`. In the Ideal Cache Model, it has cache complexity of $\Theta(n^2 + n^3/L)$, thus better than `Elementary` because of its data layout, showing how important that becomes in reality, as matrices grow in size. Its cache complexity is still worse than the recursive-style algorithms and `Tiled Multiplication`, but still performs better in the experiments - this probably is due to the recursion call overhead incurred in the other algorithms, and its comparatively better cache complexity than the `Elementary` algorithm gives it an edge also on larger matrices. The time recorded assumes an already transposed operand B. However Table 1 shows that transposing the operand B is not costly in comparison to runtime for the larger matrices [4] . However, if experiments were run for even larger matrices, the runtime and trend of the `Strassen` variants indicate that they would surpass the `Elementary Transposed` algorithm as matrices grow, in line with the theoretical assumptions.

[4]especially when using PyPy in contrast to CPython - see Appendix for transposing using CPython for this implementation

Table 1: Average runtime and standard deviation for transposing a Matrix operand of sidelength n

n	Mean	Standard Deviation
8	0.000285	0.000018
16	0.001144	0.000057
32	0.003900	0.003232
64	0.000600	0.000396
128	0.000098	0.000009
256	0.000415	0.000107
512	0.002097	0.000613
1024	0.013351	0.000782
2048	0.064354	0.002221
4096	0.302496	0.013115

6 Appendix

Table 2: Runtimes for Elementary Multiplication

n	Mean	Standard Deviation
32	0.000590	0.000078
64	0.002948	0.001175
128	0.018056	0.001182
256	0.136474	0.004228
512	1.138603	0.012668
1024	29.349494	0.175043
2048	239.539230	2.830636

Table 3: Runtimes for Elementary Transposed Multiplication

n	Mean	Standard Deviation
32	0.001213	0.000156
64	0.007411	0.000930
128	0.023968	0.013627
256	0.092080	0.015948
512	0.604014	0.067130
1024	4.323075	0.083054
2048	35.162160	0.563749
4096	279.432624	1.109959

Table 4: Runtimes for Recursive Copying Multiplication

n	Mean	Standard Deviation
32	0.013151	0.000071
64	0.106156	0.000503
128	0.885370	0.007910
256	7.243781	0.019545
512	58.679600	1.697099
1024	466.591530	0.090333
2048	3743.662680	5.292884

Table 5: Runtimes for Recursive Write Through Multiplication

n	Mean	Standard Deviation
32	0.002153	0.001034
64	0.003142	0.000427
128	0.029237	0.018328
256	0.141794	0.003253
512	1.131118	0.010071
1024	9.216727	0.370256
2048	86.723540	2.188852
4096	583.942368	0.685978

Table 6: Runtimes for Strassen

n	Mean	Standard Deviation
32	0.000978	0.000054
64	0.004022	0.000021
128	0.022729	0.000051
256	0.147755	0.000615
512	1.081802	0.004742
1024	8.119082	0.050294
2048	60.105895	0.671711
4096	432.261520	0.722904

Table 7: Runtimes for Strassen Parallel

n	Mean	Standard Deviation
32	0.129245	0.004717
64	0.240067	0.020482
128	0.319663	0.135193
256	0.490070	0.013809
512	1.813897	0.042868
1024	9.394948	0.805498
2048	51.016935	0.279285
4096	331.247326	9.133646

Table 8: Runtimes for Tiled Multiplication

n	Mean	Standard Deviation
32	0.001182	0.000528
64	0.004076	0.000469
128	0.035927	0.021914
256	0.161077	0.001176
512	1.258675	0.033025
1024	9.394447	0.504462
2048	73.265981	0.376958
4096	596.659477	6.191921

Table 9: Runtimes for transposing in CPython

n	Mean	Standard Deviation
8	0.000114	0.000018
16	0.000892	0.000190
32	0.002191	0.000504
64	0.009977	0.002219
128	0.040031	0.008406
256	0.146337	0.009647
512	0.532341	0.030577
1024	1.942888	0.034796
2048	8.215103	0.318853
4096	33.981679	2.566492