Exam Report Applied Algorithms

Kristof Gasior and Lasse Bach Andersen

16-12-2021

1 Introduction

Matrix multiplication plays an important role in scientific and real-life applications. It is used for computations in different areas such as image processing, seismic analysis or even financial analysis [3]. When the size of matrices grows, the need for efficient algorithms becomes apparent. Hence, we focus on the evaluation of different matrix multiplication algorithms in this work, namely an elementary implementation, tiling and recursive variants and the well-known Strassen's algorithm.

In section 3 we describe the implementations and improvements we have conducted. In section 4 we then describe the results of run-time experiments and conclude our findings in section 5.

2 Specifications, Testing and Input Generation

2.1 Implementation

For implementing the matrix multiplications algorithms we used the Java library provided as a template. Our algorithms operates on Java doubles, i.e. we perform matrix multiplication on the double-precision floating-point format.

For benchmarking we used our toolchain, and updated it for the Matrix multiplications, using a `TriConsumer<Matrix,Matrix,Integer>`[1] inteferface

All experiments were run in OpenJDK Runtime Environment AdoptOpenJDK-11.0.11+9 (build 11.0.11+9) on a Lenovo ThinkPad X230, 2.9 GHz Intel Core i7, 8 GB 1600 MHz DDR3 with cache sizes 32 KiB; 256 KiB; 4 MiB of *L1d*, *L2* and *L3* respectively.

2.2 Testing

The implementations were tested for correctness using the `JUnit` framework. Tests were performed against the EJML[1] library for correctness. The `add` method was tested for associativity and commutativity, and likewise the `subtract` method for the opposite. Further the `transpose` method was tested for being equal to the input matrix when transposed twice, i.e. $A = \left(A^\top\right)^\top$, and the different matrix multiplication implementations were tested for associativity and non-commutativity, for different sizes of n, m (and s).

2.3 Input Matrix Generation

To generate input matrices a random number generator from the Java standard library was used. During testing the random number generator had a seed value of 100 (to ensure the ability to reproduce possible bugs), but during bench marking, random doubles were generated using the `System.currentTimeMillis()` as seed. The generated doubles were limited to a certain range to prevent overflow. Using Eq. 1 the maximum allowable value of a matrice element d_{max} can be calculated depending on the side length of the matrix, n.

$$d_{max}(n) < \sqrt{2^{53}/n} \tag{1}$$

3 Implementation of the Algorithms

3.1 Elementary Matrix Multiplication

The naive implementation of matrix multiplication is, as known, three nested loops with a running time of $\mathcal{O}\left(n^3\right)$. However, due to *spatial locality* we may observe that the running time can be decreased by interchanging the loop order from `i, j, k` to e.g. `i, k, j` as stated in [2, pg. 88]. I.e. the program can be written as seen in Listing 1, of which the last version results in better spatial localitty as depicted in Table 1.

3.2 The Tiling Matrix Multiplication

Another variant is the Tiling matrix multiplication, very similar to the elementary algorithm, but instead it divides the matrix into $(n/s)^2$ sub-

[1] `http://ejml.org/wiki/index.php?title=Main_Page`

	Matrix	*block*				*block*				*block*			
Loop order (i, j, k)	A	+	+	+	+	+	+	+	+	+	+	+	+
	B	-	+	-	-	-	+	-	-	-	+	-	-
	C			×									
Loop order (i, k, j)	A			×									
	B	+	+	+	+	+	+	+	+	+	+	+	+
	C	+	+	+	+	+	+	+	+	+	+	+	+

Table 1: Optimizing spatial locality by interchanging loop order

```
(* loop order i, j, k *)
for (int i = 0 ; i < n ; i++)
    for (int j = 0 ; j < n ; j++)
        for (int k = 0 ; k < n ; k++)
            C[i,j] = C[i,j] + A[i,k]*B[k,j]

(* loop order i, k, j *)
for (int i = 0 ; i < n ; i++)
    for (int k = 0 ; k < n ; k++)
        for (int j = 0 ; j < n ; j++)
            C[i,j] = C[i,j] + A[i,k]*B[k,j]
```

Figure 1: Interchanging loop order to optimize cache locality

matrices. To reason about the optimal tiling size s we might consider two approaches, i.e. *calculation* and *experimentation*.

Cache	Size	Unit	Est. s
L1	32	KiB	32
L2	256	KiB	64
L3	4	MiB	256

Table 2: Cache sizes and estimations on the maximum s value suitable to perform matrix multiplication within the memory boundary of the caches.

Calculation

Since L3 is the slowest cache, lets try to estimate how big matrices we can multiply based on its size. To perform efficient matrix multiplication, it is needed to fit three matrices into the cache memory. A calculation that roughly indicates the maximum side length, n, of a $n \times n$ matrix is shown in Eq. 2. This is an indication that if we want to stay within $L3$ we should probably not choose tiles that are larger than $s = 256$. The same indications can be calculated for $L1$ and $L2$. See Table 2 for the cache sizes of this specific machine and the corresponding estimations for the tiling size, s, such that a $s \times s$ tile fits within the specified cache.

$$n = 2^k \text{ where, } k = \left\lfloor \log_2 \left(\sqrt{\delta/3} \right) \right\rfloor \text{ and } \delta = L3/d \tag{2}$$

Variabel	Value	Unit	Description
d	8	B	Size of a double
$L3$	4	MiB	Size of slowest cache
δ	524288		Amount of doubles fitting in L3
k	8		$n = 2^k$
n	256		Three $n \times n$ matrices that fits in L3

Table 3: Variables of Eq. 2

Experimental

Another approach is to experimentally determine the cache size. To do this a small program that performs 10^8 random look-ups in a single dimensional

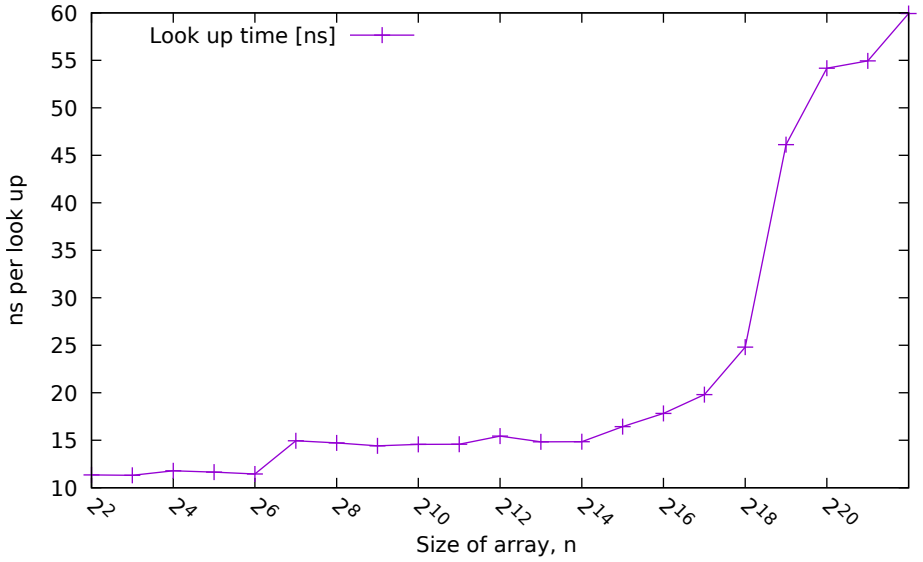

Figure 2: Time in [ns] for single lookup for different sizes of single dimensional arrays

array of doubles was implemented. The program runs with input arrays of sizes $n = \{2, 4, 8, 16, \dots\}$ and measures the time, T, it took to perform the lookups for each array. This time was then divided by 10^8 to get the time spend per lookup, i.e. $t = T/10^8$. The lookup times can be seen in Fig. 2. Notice, the radical increment in time pr. lookup from $n = 2^{18}$ to $n = 2^{19}$. Recall that the L3 cache size has 4 MiB of memory and notice that 2^{19} doubles corresponds to 4 MiB of data. We argue that the radical increase in time per look up is a direct indication of crossing the memory boundary of the $L3$ cache. To stay withing $L3$ we choose 2^{18} as the limit. Now, reuse Eq. 2, with $\delta = 2^{18}$ to calculate the biggest side length n such that the matrices fit within the $L3$ cache, which gives us the result of $n = 256$.

Determine the optimal tiling size s

The preliminary work, indicates that the optimal tiling size should be smaller than 256. However, it does not indicate the optimal tiling size, which we need to determine. In order to determine the tiling size s we run the implementation with matrix dimensions of $n = \{4, 8, 16, ..., 2048\}$ and varying sizes of $s = \{2, 4, 8, ..., n/2\}$. We set $n \leq 2048$ to stay within reasonable running times. Running the experiments, the the average running time and standard deviation was calculated by repeating the process 20

times for each size of s. Our findings, which are depicted in Fig. 3 indicate that the optimal tiling size of s is between 4 and 8, best at 4 so we set it to 4. One might find the result quite surprising when taking the initial estimation into account.

3.3 The Recursive Matrix Multiplication

Finding the optimal size of m for the recursive implementation a bench mark on sizes of $n = \{4, 8, 16, ..., 2048\}$ were run, with different sizes of $m = \{2, 4, 8, ..., n/2\}$. We set $n \leq 2048$ to stay within reasonable running times. For both the *copying* and *write through* implementation the optimal value of m to was around 64, since run times start to grow from around 64 and down. A bit larger m could also be argued for, but this would force the implementations to always perform elementary multiplication for relatively big matrices.

3.4 Strassen's Algortihm

In Fig. 5, we can evaluate that Strassen's algorithm performs best with an m between 32 and 256. By the same argument as above we consider the optimal size of m to be 64.

Strassen in parallel experiment

We used the Fork/Join Framework for parallelizing Strassen's algorithm, building a task pool from recursively created subtasks. This framework is an implementation of the `ExecutorService` interface, making it possible to distribute the work over different processors.

We observe it to be approximately 3 times faster than the single threaded Strassen implementation. Parallelizing Strassens each recursive task will follow each recursion from strassen, and interleavings are avoided since each worker thread will automatically work an an exclusive part of the matrix.

This experiment was made to investigate improvements in runtime, and we are not including a deeper concurrency analysis. The run times can be seen in Fig. 6

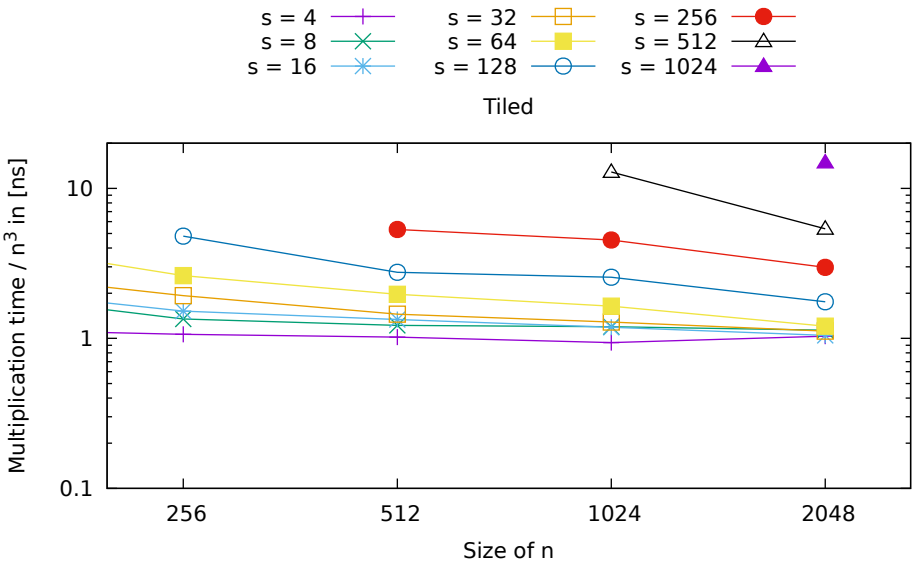

Figure 3: Run times for tiled matrix multiplication, with matrices of side length n and tile size s.

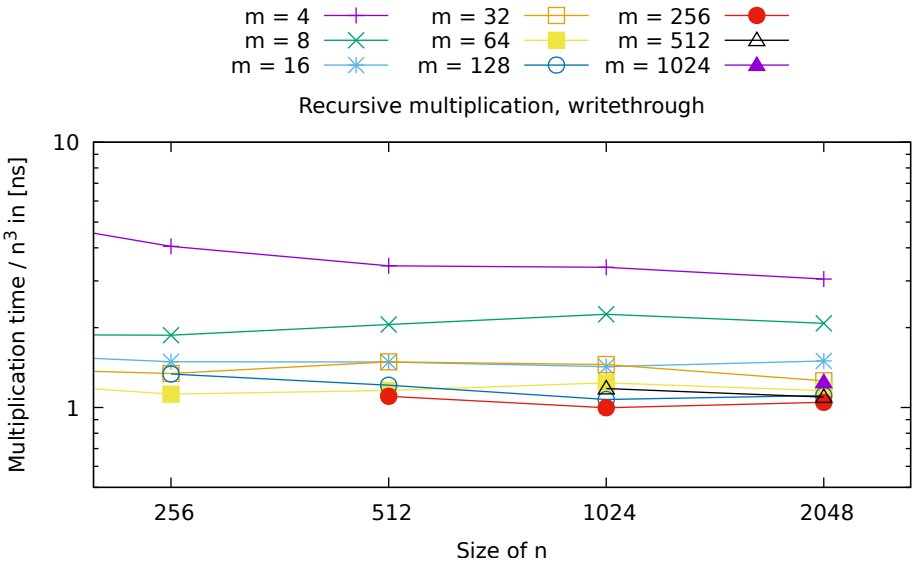

Figure 4: Run times for recursive writethrough matrix multiplication, with matrices of side length n and stop condition m.

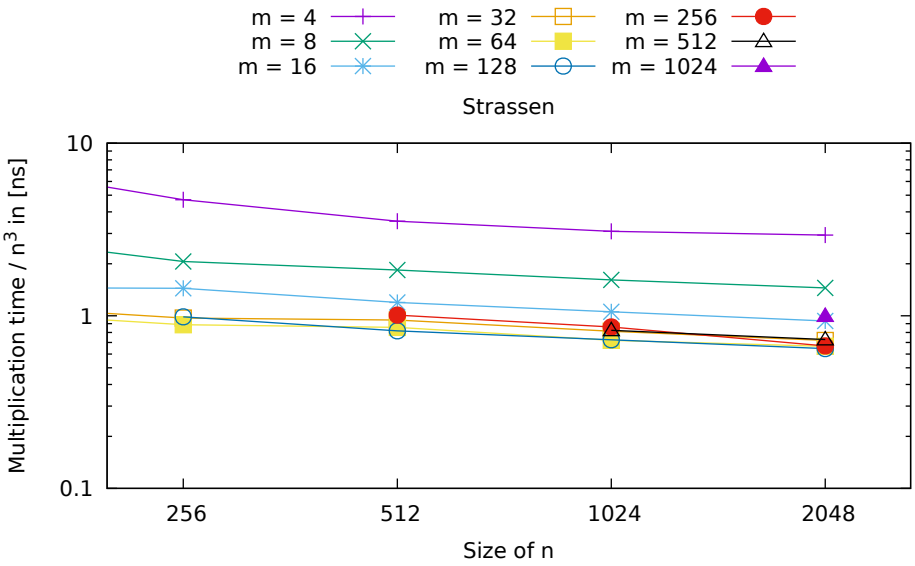

Figure 5: Run times for Strassen's matrix multiplication, with matrices of side length n and stop condition m.

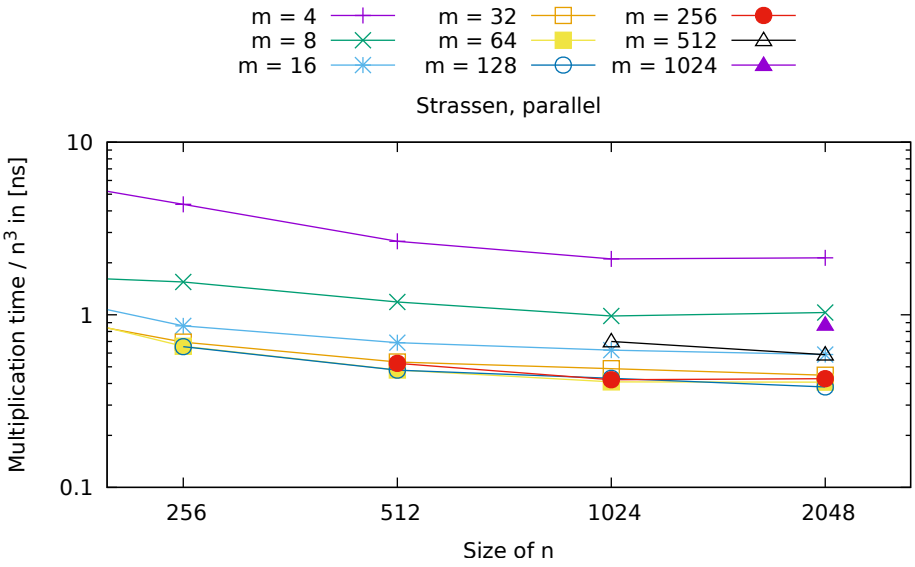

Figure 6: Run times for Strassen's matrix multiplication parallelized, with matrices of side length n and stop condition m.

4 Results of the Horse Race

As seen in Fig. 7, the *Strassen* implementation clearly beats the other implementations. The horse race was run with $s = 4$ for the *Tiled* algorithm, and $m = 64$ for both the *Recursive* algorithms, as well as both *Strassen's*. Surprisingly the *Tiled* implementation is slower than the *Elementary* implementation, however it seems more predictable in its behaviour as can be seen in Fig. 8. Not surprisingly The *Recursive* implementations as well as *Elementary transposed* are slower than the *Elementary* implementation. In Fig. 9 we observe that *Strassen Parallel* has a slow time per operation for low sizes of n, which is probably caused by the expense of initializing all threads in this framework, from start. Overall we believe that the two implementations of *Strassen's* would dominate if the race was to continue.

5 Discussion and Conclusions

As we've observed in Sec. 4 it is only the two implementations of the *Strassen* algorithms that are faster than the elementary implementations. We had expected the *Tiled* implementaion to be faster, and we wonder whether it would have been the case if we had not interchanged the loop order. It seems that the overhead connected with dividing the matrices into submatrices is higher than what is to be gained in regards of cache misses. We are also surprised to find that $s = 4$ should be optimal, since this is well below the boundary of the L1 cache size. On a a matrice with side length $n = 2048$ our horse race showed us that the *Elementary* implementation used around 9.4 seconds while the *Strassen* implementation used around 7.3 seconds, which fits quite well with theory, i.e. $\left(\sqrt[3]{9.4}\right)^{2.7} \approx 7.5$. Even though we did not manage to show any effect of tiling, interchanging the loop order was around 30 times faster when performing elementary matrix multiplication on matrices with side length $n = 4096$, which clearly shows that if we take the hardware architecture into consideration we can gain tremendous performance improvements. Tiling could still help us gain performance if e.g. we parallelized the multiplication of the tiles as seen in the parallel implementaion of *Strassen's*. Further *Strassen's* implementation shows us, that choosing the correct stop condition m, to divide the task into sub tasks of a certain size that fits into a specific cache, improves performance quite noticeably.

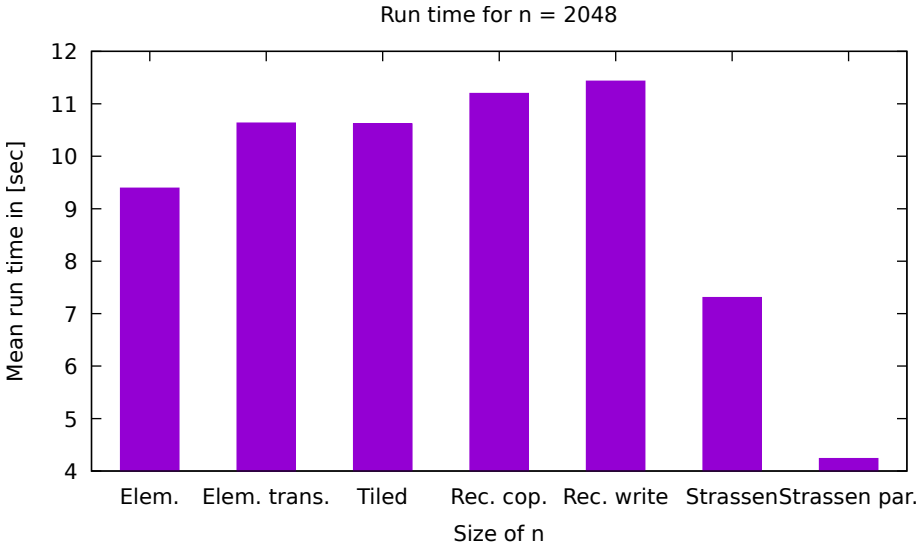

Figure 7: Running times of the horse race

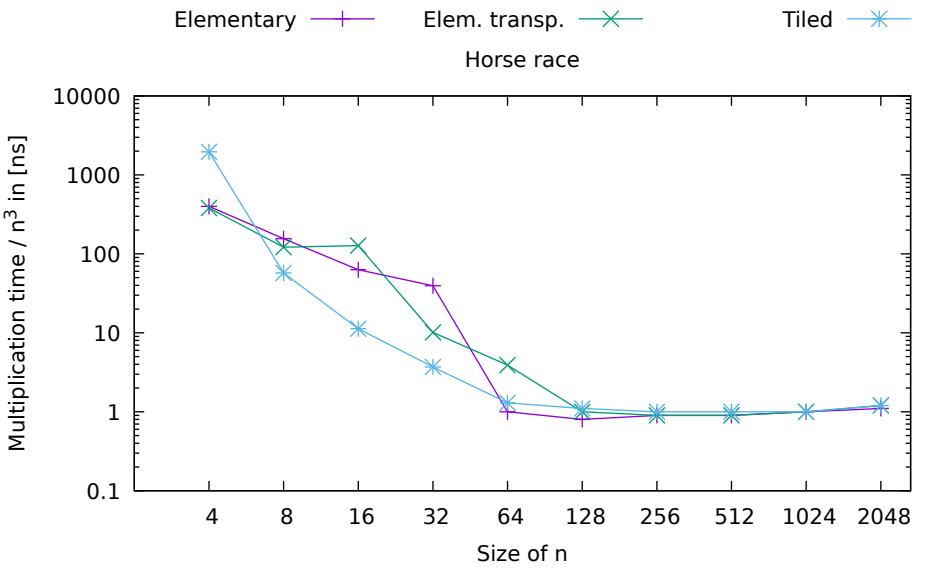

Figure 8: The elementary, transposed elementary and tiled matrix multiplication implementations

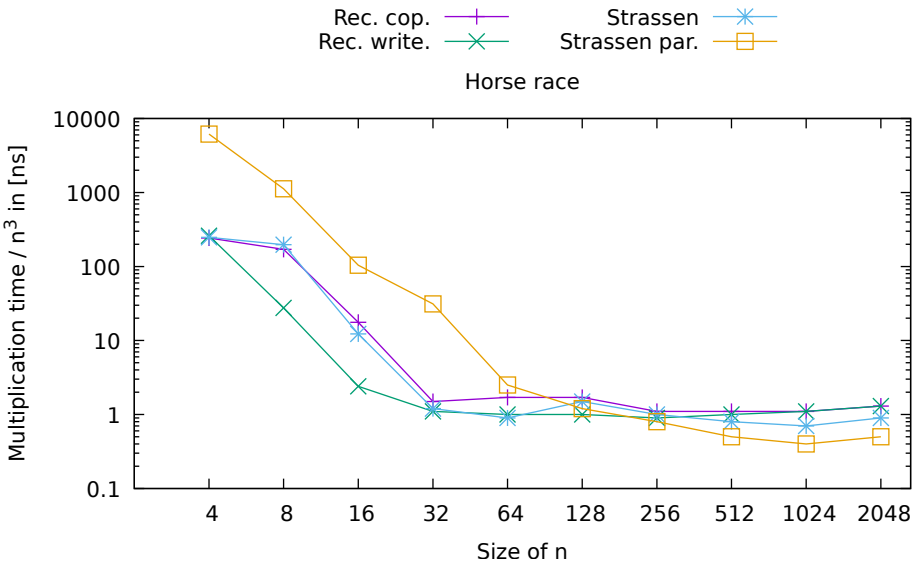

Figure 9: The recursive copying and writethrough implementation along with Strassen's and the latter parallelized.

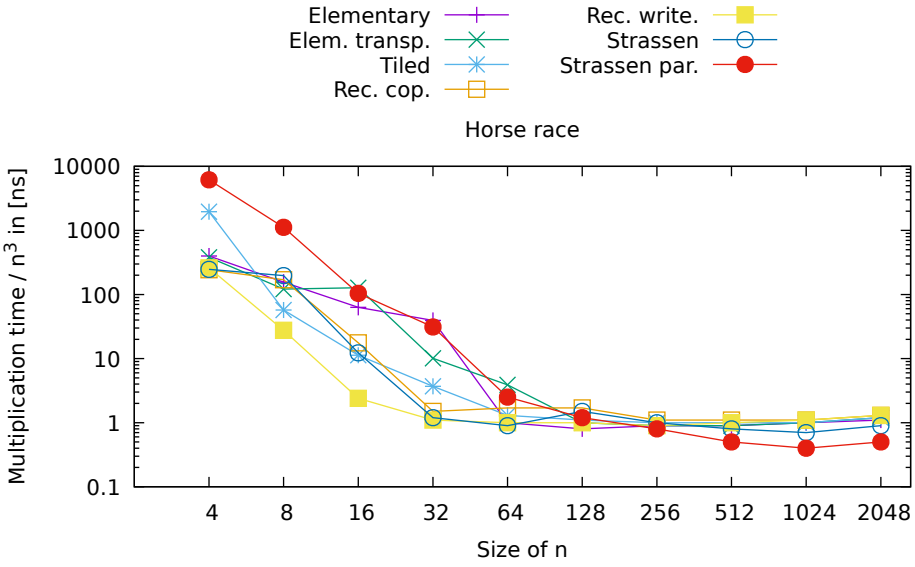

Figure 10: A plot of all the bench marked matrix multiplication implementations

References

[1] Daniel S. Dickstein. In: (2014). DOI: 10.1109/ITI.2008.4588528.

[2] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2012. ISBN: 9780123838728.

[3] Halil Snopce and Lavdrim Elmazi. "The importance of using the linear transformation matrix in determining the number of processing elements in 2-dimensional systolic array for the algorithm of matrix-matrix multiplication". In: (2008), pp. 885–892. DOI: 10.1109/ITI.2008.4588528.