Matrix Multiplication: Homework Solutions

Emanuele Ballarin

July 14, 2020

Introductory text:
Clone the Strassen's project template from:

`https://github.com/albertocasagrande/AD_strassen_template`

and solve the following exercises.

Exercise 1

Text:
Generalize the implementation to deal with non-square matrices.

Solution:
The generalization of *Volker Strassen's Algorithm for optimized square matrix multiplication* followed two main directions: the generalization to square matrices with odd dimensions, and the subsequent generalization to arbitrary rectangular matrices.

To begin with, it may be useful to recall that the original formulation of Strassen's Algorithm *(Strassen, 1969)* deals with the multiplication of $m \times m$ square matrices via subsequent recursive partitioning of each matrix into 4 equal blocks, which are then multiplied, added and/or subtracted in order to re-obtain the result looked-for, by relying on the *Matrix Block Multiplication theorem* and some algebraic manipulations.

As a consequence, such algorithm can be recurred till the base-case of scalar-scalar multiplication only if m is a power of 2. Even more generally, one single recursive iteration can be executed only if m is even.

However, since the overhead associated to matrix manipulations can easily overweight the lower asymptotic complexity of Strassen's approach for small m, recursion may be early-stopped falling back to conventional *element-by-element* naive matrix multiplication below a fixed threshold for m.
What is needed, then, is to ensure that (still square) matrices have even dimensions as long as the recursive partitioning has to be applied.

To overcome this requirement, *dynamic padding to even* has been employed – i.e. adding a row/column of zeros to odd dimensions just when partitioning should have been applied, exploiting the fact that such additional manipulation will sort no effect on the multiplication. A same-shaped and same-positioned row/column of zeros should be removed, too, from the resulting matrix in order to preserve dimensionality, after the multiplication has been executed.

To extend such approach to non-square matrices, the principle of *minimal padding* has been followed. Instead of padding to the smallest square matrix embedding the rectangular matrices whose multiplication is seeked, the *sliding-square matrix multiplication* approach has been applied.
First, the largest square matrix whose side is an integer divisor of original matrices dimensions is determined. Then, the initial product is decomposed in many square-matrix products of such aforementioned dimension. The result is then obtained by summation and/or juxtaposition of the resulting blocks.

Since this approach turned out to be heavily affected by the overhead associated to looping over the different square blocks, a further approach has been pursued – though not originally thought as part of this exam-submission.

By following the then-seminal idea of Turnbull *(Turnbull et al., 1996)*, an implementation based on *direct partitioning with even dynamic peeling* has been produced, with some success.
In this case, recursive partitioning is applied directly to rectangular matrices – as long as they have even dimensions, following the same scheme as in square-Strassen. In case some dimensions are not even, the largest even-sized matrix is considered and, afterwards, the missing contribution is added through a *correction* consisting in, at most, 8 among scalar-scalar, vector-scalar, vector-vector or vector-submatrix products, all performed via naive element-wise multiplication.

Such procedure does not require any padding, is lazily-applied, offers tighter time-complexity and memory bounds compared to padding-based implementations, and conditional branching is maintained at the bare minimum. This theoretical advantage is also confirmed by benchmarks.

Exercise 2

Text:
Improve the implementation of the Strassen's algorithm by reducing the memory allocations and test the effects on the execution time.

Solution:
In order to fulfill the request, only 2 `S` and 1 `P` matrices have been allocated, by re-using previously-allocated matrices and transferring via summation intermediate results to the final resulting matrix as soon as possible.

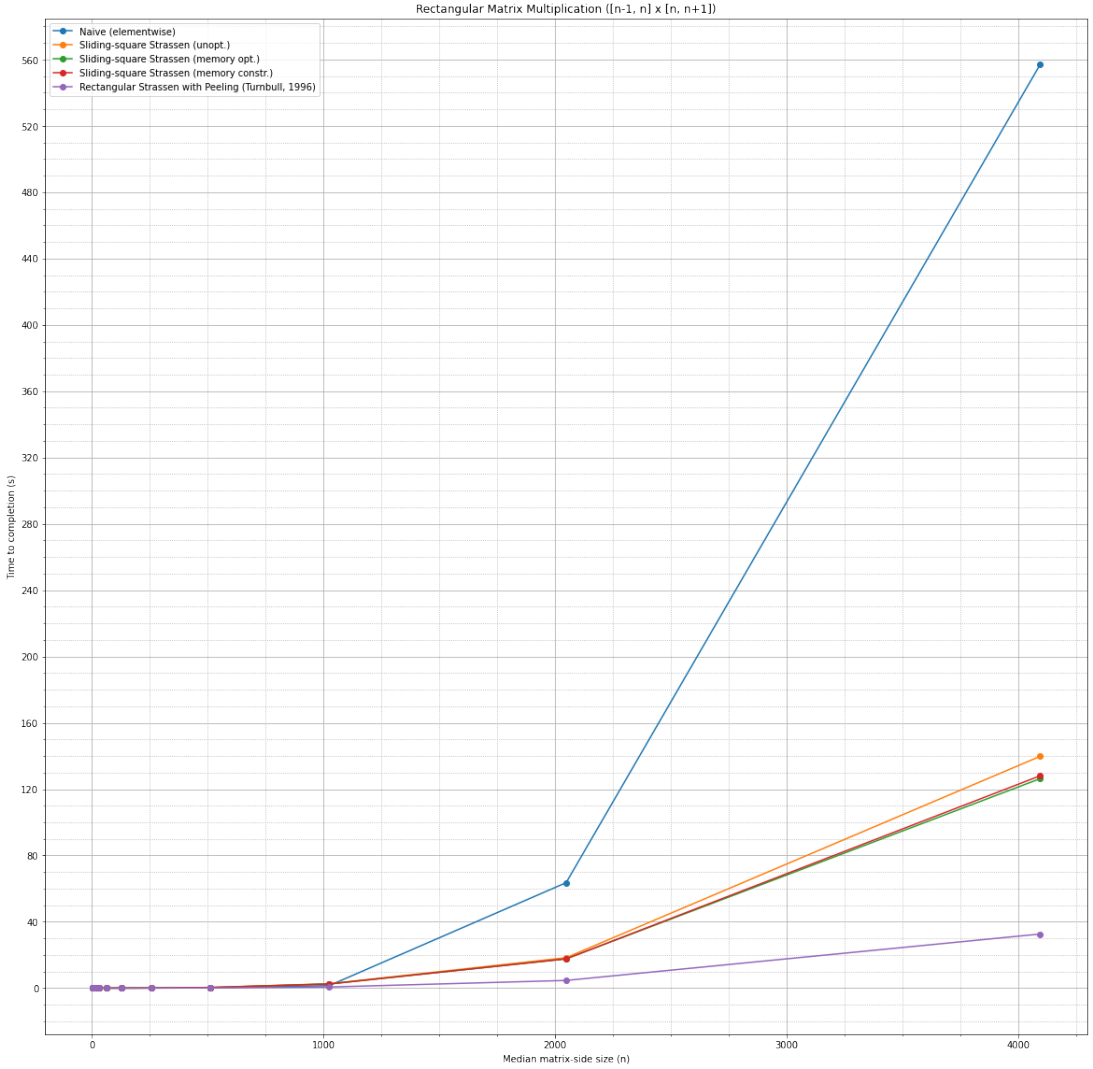
Allocation of a smaller number of matrices would have been impossible without a major rework of the algorithm (and of debatable performance effectiveness nonetheless).

Such approach – though increasing the number of total sums and differences w.r.t. another (and tested) possibility – i.e. allocating as many matrices as required to just perform summations or differences once per block of the resulting matrix – still offers, according to benchmarks, the overall smallest (though not by a large margin) execution time across different sizes.
This can be explained by noting that over-writing an already-allocated matrix requires (much) less time than also having to allocate space on the system heap.

Benchmark results

As it is possible to see from the *benchmark* results reported on next page, any Strassen-like implementation turns out to be asymptotically more efficient than naive matrix multiplication. However – bearing in mind that for a mean matrix-side size ≤ 64 recursion is stopped and naive matrix multiplication is employed – it is worth noting that:

- Below a median matrix-side size of 1000, the *sliding-square* implementations of Strassen's Algorithm show (though not by a large margin) even worse performance compared to naive multiplication;
- Above a median matrix-side size of 1000, the Strassen-based version begins to show evident (and growing) performance benefits;
- Among the *sliding-square* versions, the *memory-otimized* one performs slightly better than the *memory-constrained* one for bigger matrices, which in turn is appreciably faster than the *unoptimized* one. Such effect seems to grow with the size of the matrices.
- The *peeling-based* implementation is the only one to consistently behave faster than (or equal to) the naive one across the entire set of sizes. Indeed, its performance gains are very significant even compared to the other Strassen-like implementations.


Rectangular Matrix Multiplication ([n-1, n] x [n, n+1])
Naive (elementwise)
Sliding-square Strassen (unopt.)
Sliding-square Strassen (memory opt.)
Sliding-square Strassen (memory constr.)
Rectangular Strassen with Peeling (Turnbull, 1996)
Time to completion (s)
Median matrix-side size (n)
0
40
80
120
160
200
240
280
320
360
400
440
480
520
560
0
1000
2000
3000
4000