Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm

Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, Wei Zhou

Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm

Brice Boyer* Jean-Guillaume Dumas* Clément Pernet†
Wei Zhou‡

January 27, 2009

Abstract

We propose several new schedules for Strassen-Winograd's matrix multiplication algorithm, they reduce the extra memory allocation requirements by three different means: by introducing a few pre-additions, by overwriting the input matrices, or by using a first recursive level of classical multiplication. In particular, we show two fully in-place schedules: one having the same number of operations, if the input matrices can be overwritten; the other one, slightly increasing the constant of the leading term of the complexity, if the input matrices are read-only. Many of these schedules have been found by an implementation of an exhaustive search algorithm based on a pebble game.

1 Introduction

Strassen's algorithm [16] was the first sub-cubic algorithm for matrix multiplication. Its improvement by Winograd [17] led to a highly practical algorithm. The best asymptotic complexity for this computation has been successively improved since then, down to $\mathcal{O}\left(n^{2.376}\right)$ in [5] (see [3, 4] for a review), but Strassen-Winograd's still remains one of the most practicable. Former studies on how to turn this algorithm into practice can be found in [2, 9, 10, 6] and references therein for numerical computation and in [15, 7] for computations over a finite field.

*Laboratoire J. Kuntzmann, Université de Grenoble. 51, rue des Mathématiques, umr CNRS 5224, bp 53X, F38041 Grenoble, France, \{Brice.Boyer,Jean-Guillaume.Dumas\}@imag.fr

†Laboratoire LIG, Université de Grenoble. umr CNRS, F38330 Montbonnot, France. Clement.Pernet@imag.fr

‡School of Computer Science, University of Waterloo, Waterloo, ON, N2B 3G1, Canada. w2zhou@uwaterloo.ca

In this paper, we propose new schedules of the algorithm, that reduce the extra memory allocation, by three different means: by introducing a few pre-additions, by overwriting the input matrices, or by using a first recursive level of classical multiplication. These schedules can prove useful e.g. for memory efficient computations of the rank, determinant, nullspace basis, system resolution, matrix inversion... Indeed, the matrix multiplication based LQUP factorization of [11] can be computed with no other temporary allocations than the ones involved in its block matrix multiplications [12]. Therefore the improvements on the memory requirements of the matrix multiplication, used together e.g. with cache optimization strategies [1], will directly improve these higher level computations. We only consider here the computational complexity and space complexity, counting the number of the number of arithmetic operations and memory allocations and do not consider stability issues. Further studies have thus to be made in order to use these schedules for numerical computations. They are nonetheless useful for exact computations, for instance on integer/rational or finite field applications [8, 14].

The remaining of this paper is organised as follows: we review Strassen-Winograd's algorithm and existing memory schedules in sections 2 and 3. We then present in section 4 the dynamic program we used to search for schedules. This enables us to give several schedules overwriting their inputs in section 5, and then a new schedule for $C \leftarrow AB + C$ using only two extra temporaries in section 6, all of them preserving the leading term of the arithmetic complexity. Eventually, in section 7, we present a generic way of transforming non in-place matrix multiplication algorithms into in-place ones, with a small constant factor overhead.

2 Strassen-Winograd Algorithm

We first review Strassen-Winograd's algorithm, and setup the notations that will be used throughout the paper.
Let m, n and k be powers of 2. Let A and B be two matrices of dimension $m \times k$ and $k \times n$ and let $C = A \times B$. Consider the natural block decomposition:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where A_{11} and B_{11} respectively have dimensions $m/2 \times k/2$ and $k/2 \times n/2$. Winograd's algorithm computes the $m \times n$ matrix $C = A \times B$ with the following 22 block operations:
• 8 additions:

$$\begin{array}{lll} S_1 \leftarrow A_{21} + A_{22} & S_2 \leftarrow S_1 - A_{11} & S_3 \leftarrow A_{11} - A_{21} \\ T_1 \leftarrow B_{12} - B_{11} & T_2 \leftarrow B_{22} - T_1 & T_3 \leftarrow B_{22} - B_{12} \\ S_4 \leftarrow A_{12} - S_2 & & T_4 \leftarrow T_2 - B_{21} \end{array}$$

- 7 recursive multiplications:

$$\begin{array}{lll} P_1 \leftarrow A_{11} \times B_{11} & P_2 \leftarrow A_{12} \times B_{21} & \\ P_3 \leftarrow S_4 \times B_{22} & P_4 \leftarrow A_{22} \times T_4 & \\ P_5 \leftarrow S_1 \times T_1 & P_6 \leftarrow S_2 \times T_2 & P_7 \leftarrow S_3 \times T_3 \end{array}$$

- 7 final additions:

$$\begin{array}{llll} U_1 \leftarrow P_1 + P_2 & U_2 \leftarrow P_1 + P_6 & & \\ U_3 \leftarrow U_2 + P_7 & U_4 \leftarrow U_2 + P_5 & & \\ U_5 \leftarrow U_4 + P_3 & U_6 \leftarrow U_3 - P_4 & U_7 \leftarrow U_3 + P_5 \end{array}$$

- The result is the matrix: $C = \begin{bmatrix} U_1 & U_5 \\ U_6 & U_7 \end{bmatrix}$.

Figure 1 illustrates the dependencies between these tasks.

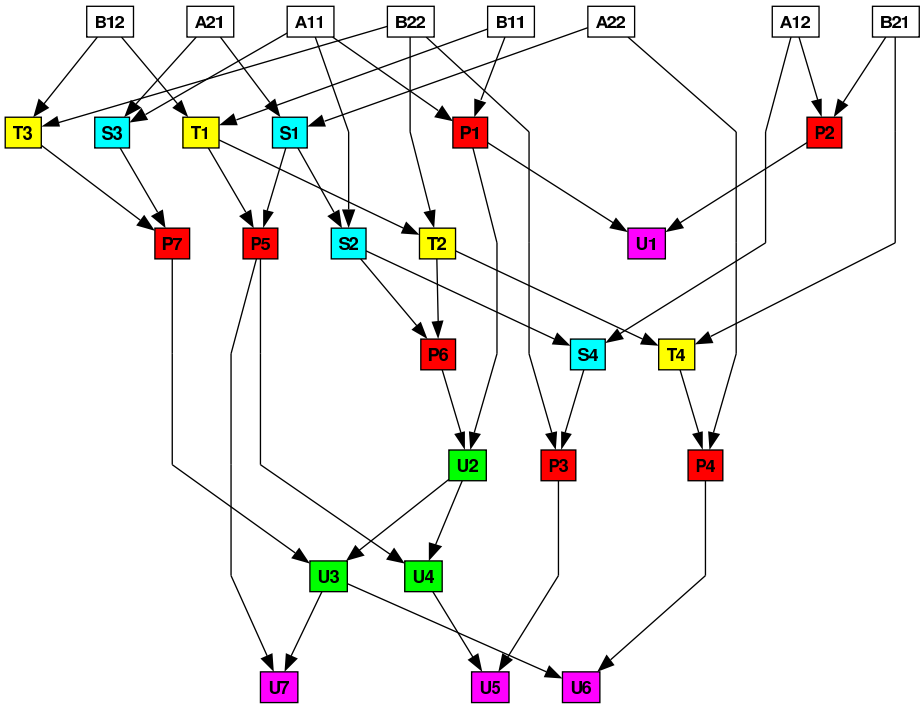

Figure 1: Winograd's task dependency graph

3 Existing memory placements

Unlike the classic multiplication algorithm, Winograd's algorithm requires some extra temporary memory allocations to perform its 22 block operations.

3.1 Standard product

We first consider the basic operation $C \leftarrow A \times B$. The best known schedule for this case was given by [6]. We reproduce a similar schedule in table 1. It

#	operation	loc.	#	operation	loc.
1	$S_3 = A_{11} - A_{21}$	X	12	$P_1 = A_{11}B_{11}$	X
2	$T_3 = B_{22} - B_{12}$	Y	13	$U_2 = P_1 + P_6$	C_{12}
3	$P_7 = S_3T_3$	C_{21}	14	$U_3 = U_2 + P_7$	C_{21}
4	$S_1 = A_{21} + A_{22}$	X	15	$U_4 = U_2 + P_5$	C_{12}
5	$T_1 = B_{12} - B_{11}$	Y	16	$\mathbf{U_7} = U_3 + P_5$	C_{22}
6	$P_5 = S_1T_1$	C_{22}	17	$\mathbf{U_5} = U_4 + P_3$	C_{12}
7	$S_2 = S_1 - A_{11}$	X	18	$T_4 = T_2 - B_{21}$	Y
8	$T_2 = B_{22} - T_1$	Y	19	$P_4 = A_{22}T_4$	C_{11}
9	$P_6 = S_2T_2$	C_{12}	20	$\mathbf{U_6} = U_3 - P_4$	C_{21}
10	$S_4 = A_{12} - S_2$	X	21	$P_2 = A_{12}B_{21}$	C_{11}
11	$P_3 = S_4B_{22}$	C_{11}	22	$\mathbf{U_1} = P_1 + P_2$	C_{11}

Table 1: Winograd's algorithm for operation $C \leftarrow A \times B$, with two temporaries

requires two temporary blocks X and Y whose dimensions are respectively equal to $m/2 \times \max(k/2, n/2)$ and $k/2 \times n/2$. Thus the extra memory used is:

$$E_1(m,k,n) = \frac{m}{2}\max\left(\frac{k}{2},\frac{n}{2}\right) + \frac{k}{2}\frac{n}{2} + E_1\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right).$$

Summing these temporary allocations over every recursive levels leads to a total amount of memory, where for brevity $M = \min\{m,k,n\}$:

$$\begin{aligned} E_1(m,k,n) &= \sum_{i=1}^{\log_2(M)} \frac{1}{4^i}\left(m\max(k,n) + kn\right) \\ &= \frac{1}{3}\left(1 - \frac{1}{M^2}\right)\left(m\max(k,n) + kn\right) \\ &< \frac{1}{3}\left(m\max(k,n) + kn\right). \end{aligned} \tag{1}$$

We can prove in the same manner the following lemma:

Lemma 1. *Let m, k and n be powers of two, $g(x,y,z)$ be homogeneous, $M = \min\{m,k,n\}$ and $f(m,k,n)$ be a function such that*

$$f(m,k,n) = \begin{cases} g\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right) + f\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right) & \text{if } m,n \text{ and } k > 1 \\ 0 & \text{otherwise.} \end{cases}$$

Then $f(m,k,n) = \frac{1}{3}\left(1 - \frac{1}{M^2}\right)g(m,k,n) < \frac{1}{3}g(m,k,n)$.

In the remaining of the paper, we use E_i to denote the amount of extra memory used in table number i. The amount of extra memory we consider is always the sum up to the last recursion level.

Finally, assuming $m = n = k$ gives a total extra memory requirement of $E_1(n, n, n) < 2/3n^2$.

3.2 Product with accumulation

For the more general operation $C \leftarrow \alpha A \times B + \beta C$, a first nave method would compute the product $\alpha A \times B$ using the scheduling of table 1, into a temporary matrix C' and finally compute $C \leftarrow C' + \beta C$. It would require $(1+2/3)n^2$ extra memory allocations in the square case.
Now the schedule of table 2 due to [10, fig. 6] only requires 3 temporary blocks for the same number of operations (7 multiplications and $4+15$ additions). The

#	operation	loc.	#	operation	loc.
1	$S_1 = A_{21} + A_{22}$	X	12	$S_4 = A_{12} - S_2$	X
2	$T_1 = B_{12} - B_{11}$	Y	13	$T_4 = T_2 - B_{21}$	Y
3	$P_5 = \alpha S_1 T_1$	Z	14	$C_{12} = \alpha S_4 B_{22} + C_{12}$	C_{12}
4	$C_{22} = P_5 + \beta C_{22}$	C_{22}	15	$\mathbf{U_5} = U_2 + C_{12}$	C_{12}
5	$C_{12} = P_5 + \beta C_{12}$	C_{12}	16	$P_4 = \alpha A_{22} T_4 - \beta C_{21}$	C_{21}
6	$S_2 = S_1 - A_{11}$	X	17	$S_3 = A_{11} - A_{21}$	X
7	$T_2 = B_{22} - T_1$	Y	18	$T_3 = B_{22} - B_{12}$	Y
8	$P_1 = \alpha A_{11} B_{11}$	Z	19	$U_3 = \alpha S_3 T_3 + U_2$	Z
9	$C_{11} = P_1 + \beta C_{11}$	C_{11}	20	$\mathbf{U_7} = U_3 + C_{22}$	C_{22}
10	$U_2 = \alpha S_2 T_2 + P_1$	Z	21	$\mathbf{U_6} = U_3 - C_{21}$	C_{21}
11	$\mathbf{U_1} = \alpha A_{12} B_{21} + C_{11}$	C_{11}	22		

Table 2: Schedule for operation $C \leftarrow \alpha A \times B + \beta C$ with 3 temporaries

required three temporary blocks X, Y, Z have dimensions $m/2 \times n/2$, $m/2 \times k/2$ and $k/2 \times n/2$. Hence, using lemma 1, we get

$$E_2(m, k, n) = \frac{1}{3}\left(1 - \frac{1}{M^2}\right)(mk + kn + mn). \tag{2}$$

With $m = n = k$, this gives $E_2(n, n, n) < n^2$.
We propose in table 9 a new schedule for the same operation $\alpha A \times B + \beta C$ only requiring two temporary blocks.
Our new schedule is more efficient if some inner calls overwrite their temporary input matrices. We now present some overwriting schedules and the dynamic program we used to find them.

4 Exhaustive search algorithm

We used a brute force search algorithm[1] to get some of the new schedules that will be presented in the following sections. It is very similar to the pebble game of Huss-Lederman et al. [10].

[1] The code is available at `http://ljk.imag.fr/CASYS/LOGICIELS/Galet`.

A sequence of computations is represented as a directed graph, very much like figure 1 is built from Winograd's algorithm.
A node represents a program variable. The nodes can be classified as initials (when they correspond to inputs), temporaries (for intermediate computations) or finals (results or nodes that we want to keep, such as ready-only inputs).
The arcs represent the operations; they point from the operands to the result.
A pebble represents an allocated memory. We can put pebbles on any nodes, move or remove them according to a set of simple rules shown below.
When a pebble arrives on a node, the computation at the associated variable starts, and can be "partially" or "fully" executed. If not specified, it is assumed that the computation is fully executed.
Arcs can be removed, when the corresponding operation has been computed.
These two rules are especially useful for accumulation operations: for example, it is possible to try schedule the multiplication separately from the addition in an otherwise recursive $AB + C$ call; the multiplication-involved arcs would then be removed first and the accumulated part later. It is also useful if we do not want to fix the way some additions are performed: if $U_3 = P_1 + P_6 + P_7$ the associativity allows different ways of computing the sum and we let the program explore these possibilities. At the beginning of the exploration, each initial node has a pebble and we may have a few extra available pebbles. The program then tries to apply the following rules, in order, on each node. The program stops when every final node has a pebble or when there is no more allowed move:

- *Rule* 0. *Computing a result/removing arcs.* If a node has a pebble and parents with pebbles, then the operation can be performed and the corresponding arcs removed. The node is then at least partially computed.
- *Rule* 1. *Freeing some memory/removing a pebble.* If a node is isolated and not final, its pebble is freed. This means that we can reclaim the memory here because this node has been fully computed (no arcs pointing to it) and is no longer in use as an operand (no arcs initiating from it).
- *Rule* 2. *Computing in place/moving a pebble.* If a node P has a full pebble and a single empty child node S and if other parents of S have pebbles on them, then the pebble on P may be transferred to S (corresponding arcs are removed). This means an operation has been made in place in the parent P's pebble.
- *Rule* 3. *Using more memory/adding a pebble.* If parents of an empty node N have pebbles and a free pebble is available, then N can be assigned this pebble and the corresponding arcs are removed. This means that the operation is computed in a new memory location.
- *Rule* 4. *Copying some memory/duplicating a pebble.* A computed node having a pebble can be duplicated. The arcs pointed to or from the original node are then rearranged between them. This means that a temporary result has been copied into some free place to allow more flexibility.

5 Overwriting input matrices

We now relax some constraints on the previous problem: the input matrices A and B can be overwritten, as proposed by [13]. For the sake of simplicity, we first give schedules only working for square matrices (i.e. $m = n = k$ and any memory location is supposed to be able to receive any result of any size). We nevertheless give the memory requirements of each schedule as a function of m; k and n. Therefore it is easier in the last part of this section to adapt the proposed schedules partially for the general case.

5.1 Standard product

We propose in table 3 a new schedule that computes the product $C \leftarrow A \times B$ without any temporary memory allocation. The idea here is to find an ordering where the recursive calls can be made also in place such that the operands of a multiplication are no longer in use after the multiplication because they are overwritten. An exhaustive search showed that no schedule exists overwriting less than four sub-blocks.

#	operation	loc.	#	operation	loc.
1	$S_3 = A_{11} - A_{21}$	C_{11}	12	$S_4 = A_{12} - S_2$	C_{22}
2	$S_1 = A_{21} + A_{22}$	A_{21}	13	$P_6 = S_2T_2$	C_{12}
3	$T_1 = B_{12} - B_{11}$	C_{22}	14	$U_2 = P_1 + P_6$	C_{12}
4	$T_3 = B_{22} - B_{12}$	B_{12}	15	$U_3 = U_2 + P_7$	C_{21}
5	$P_7 = S_3T_3$	C_{21}	16	$P_3 = S_4B_{22}$	B_{11}
6	$S_2 = S_1 - A_{11}$	B_{12}	17	$\mathbf{U_7} = U_3 + P_5$	C_{22}
7	$P_1 = A_{11}B_{11}$	C_{11}	18	$\mathbf{U_6} = U_3 - P_4$	C_{21}
8	$T_2 = B_{22} - T_1$	B_{11}	19	$U_4 = U_2 + P_5$	C_{12}
9	$P_5 = S_1T_1$	A_{11}	20	$\mathbf{U_5} = U_4 + P_3$	C_{12}
10	$T_4 = T_2 - B_{21}$	C_{22}	21	$P_2 = A_{12}B_{21}$	B_{11}
11	$P_4 = A_{22}T_4$	A_{21}	22	$\mathbf{U_1} = P_1 + P_2$	C_{11}

Table 3: Schedule for operation $C \leftarrow A \times B$ in place

Figure 2 shows the affectation of the tasks on each variable, by row.

Note that this schedule uses only two blocks of A or B as extra temporaries (namely A_{11}, A_{21}, B_{11} and B_{12}) but overwrites all of A and B. For instance the recursive computation of P_2 requires overwriting parts of A_{12} and B_{21} too. This schedule can nonetheless overwrite strictly only two blocks of A and two blocks of B. This is achieved by making a backup of the overwritten parts into some available extra memory. The schedule is then modified as follows:

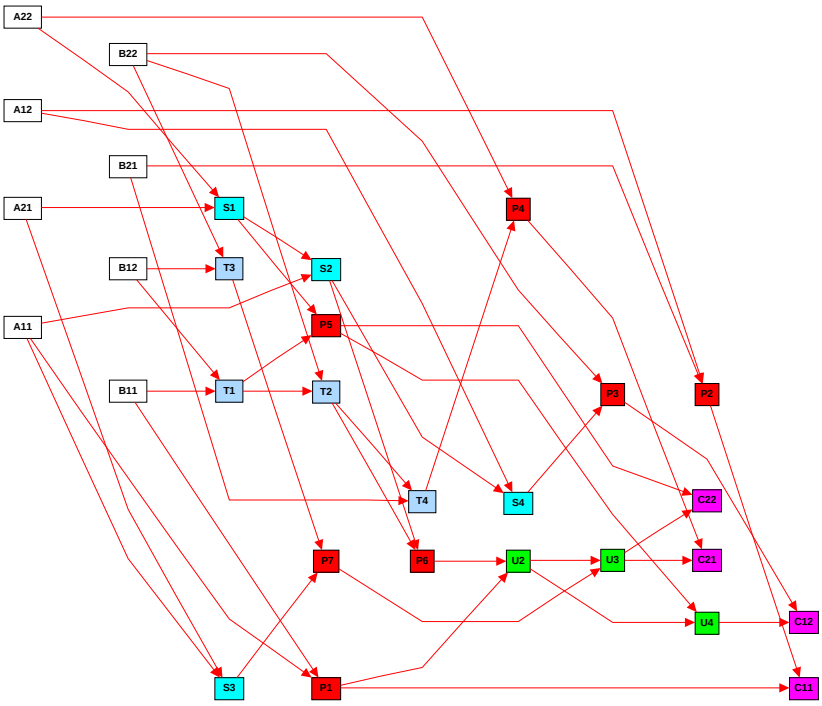

Figure 2: In-place Strassen-Winograd matrix multiplication

#	operation	loc.
10bis	`Copy` A_{22} into C_{12}	C_{12}
11	$P_4 = A_{22}T_4$	A_{21}
11bis	`Restore` A_{22} from C_{12}	A_{22}
15bis	`Copy` B_{22} into B_{12}	B_{12}
16	$P_3 = S_4B_{22}$	B_{11}
16bis	`Restore` B_{22} from B_{12}	B_{22}
20bis	`Copy` A_{12} into A_{21}	A_{21}
20ter	`Copy` B_{21} into B_{12}	B_{12}
21	$P_2 = A_{12}B_{21}$	B_{11}
21bis	`Restore` B_{21} from B_{12}	B_{21}
21ter	`Restore` A_{12} from A_{21}	A_{12}

In the following, we will denote by `IP` for `InPlace`, either one of these two schedules.

We present in tables 4 and 5 two new schedules overwriting only one of the two input matrices, but requiring an extra temporary space. These two schedules are denoted `O1L` and `O1R`. The exhaustive search also showed that no schedule exists overwriting only one of A and B and using no extra temporary. We

#	operation	loc.	#	operation	loc.
1	$S_3 = A_{11} - A_{21}$	C_{22}	13	$T_4 = T_2 - B_{21}$	A_{11}
2	$S_1 = A_{21} + A_{22}$	A_{21}	14	$U_2 = P_1 + P_6$	C_{21}
3	$S_2 = S_1 - A_{11}$	C_{12}	15	$U_4 = U_2 + P_5$	C_{12}
4	$T_1 = B_{12} - B_{11}$	C_{21}	16	$U_3 = U_2 + P_7$	C_{21}
5	$P_1 = \texttt{O1L}(A_{11}B_{11})$	C_{11}	17	$\mathbf{U_7} = U_3 + P_5$	C_{22}
6	$T_3 = B_{22} - B_{12}$	A_{11}	18	$\mathbf{U_5} = U_4 + P_3$	C_{12}
7	$P_7 = \texttt{IP}(S_3T_3)$	X		$A_{21} = \texttt{Copy}(A_{12})$	A_{21}
8	$T_2 = B_{22} - T_1$	A_{11}	19	$P_2 = \texttt{O1L}(A_{12}B_{21})$	X
9	$P_5 = \texttt{IP}(S_1T_1)$	C_{22}		$A_{12} = \texttt{Copy}(A_{21})$	A_{12}
10	$S_4 = A_{12} - S_2$	C_{21}	20	$\mathbf{U_1} = P_1 + P_2$	C_{11}
11	$P_3 = \texttt{O1L}(S_4B_{22})$	A_{21}	21	$P_4 = \texttt{O1R}(A_{22}T_4)$	A_{21}
12	$P_6 = \texttt{O1L}(S_2T_2)$	C_{21}	22	$\mathbf{U_6} = U_3 - P_4$	C_{21}

Table 4: O1L Schedule for operation $C \leftarrow A \times B$ using strictly two blocks of A and one temporary

remark that if it is allowed to use three blocks of A, the two Copy operations of table 4 can be avoided. Note also that if three blocks of A can be overwritten then the last multiplication (P_4) can also be made by a strict recursive call to O1L. Both behaviors can be simultaneous if four blocks of A (the whole matrix) are overwritable. Similarly, for O1R, the copy or the call to O1L for P_3 can also

#	operation	loc.	#	operation	loc.
1	$S_3 = A_{11} - A_{21}$	C_{22}	13	$S_4 = A_{12} - S_2$	B_{11}
2	$S_1 = A_{21} + A_{22}$	C_{21}	14	$U_2 = P_1 + P_6$	C_{21}
3	$T_1 = B_{12} - B_{11}$	C_{12}	15	$U_4 = U_2 + P_5$	C_{12}
4	$P_1 = \texttt{O1R}(A_{11}B_{11})$	C_{11}	16	$U_3 = U_2 + P_7$	C_{21}
5	$S_2 = S_1 - A_{11}$	B_{11}	17	$\mathbf{U_7} = U_3 + P_5$	C_{22}
6	$T_3 = B_{22} - B_{12}$	B_{12}	18	$\mathbf{U_6} = U_3 - P_4$	C_{21}
7	$P_7 = \texttt{IP}(S_3T_3)$	X	19	$P_3 = \texttt{O1L}(S_4B_{22})$	B_{12}
8	$T_2 = B_{22} - T_1$	B_{12}	20	$\mathbf{U_5} = U_4 + P_3$	C_{12}
9	$P_5 = \texttt{IP}(S_1T_1)$	C_{22}		$B_{11} = \texttt{Copy}(B_{21})$	B_{11}
10	$T_4 = T_2 - B_{21}$	C_{12}	21	$P_2 = \texttt{O1R}(A_{12}B_{21})$	B_{12}
11	$P_6 = \texttt{O1R}(S_2T_2)$	C_{21}		$B_{21} = \texttt{Copy}(B_{11})$	B_{21}
12	$P_4 = \texttt{O1R}(A_{22}T_4)$	B_{12}	22	$\mathbf{U_1} = P_1 + P_2$	C_{11}

Table 5: O1R Schedule for operation $C \leftarrow A \times B$ using strictly two blocks of B and one temporary

be avoided if three or four blocks of B are overwritable.
We now compute the extra memory needed for any m, k and n such that table 5 is still valid. We procede in the same way as before and use the above results for the general case (5.3).
The size of the temporary block X is $\frac{m}{2}\frac{n}{2}$, the extra memory required for table 5 hence satisfies: $E_5(m,k,n) < \frac{1}{3}mn$.

5.2 Product with accumulation

We now consider the operation $C \leftarrow \alpha A \times B + \beta C$, where the input matrices A and B can be overwritten. We propose in table 6 a schedule that only requires 2 temporary block matrices, instead of the 3 in table 2. This is achieved by overwriting the inputs and by using two additional pre-additions on the matrix C. We also propose in table 6 a similar schedule overwriting only e.g. the right

#	operation	loc.	#	operation	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	13	$P_4 = \alpha A_{22} T_4 - \beta Z_2$	C_{21}
2	$S_1 = A_{21} + A_{22}$	X	14	$S_4 = A_{12} - S_2$	A_{21}
3	$T_1 = B_{12} - B_{11}$	Y	15	$P_6 = \alpha \mathtt{IP}(S_2 T_2)$	X
4	$Z_2 = C_{21} - Z_1$	C_{21}	16	$P_2 = \alpha A_{12} B_{21} + \beta C_{11}$	C_{11}
5	$T_3 = B_{22} - B_{12}$	B_{12}	17	$\mathbf{U_1} = P_1 + P_2$	C_{11}
6	$S_3 = A_{11} - A_{21}$	A_{21}	18	$U_2 = P_1 + P_6$	X
5	$P_7 = \alpha S_3 T_3 + \beta Z_1$	C_{22}	17	$U_3 = U_2 + P_7$	C_{22}
8	$S_2 = S_1 - A_{11}$	A_{21}	20	$U_4 = U_2 + P_5$	X
9	$T_2 = B_{22} - T_1$	B_{12}	21	$\mathbf{U_6} = U_3 - P_4$	C_{21}
10	$P_5 = \alpha S_1 T_1 + \beta C_{12}$	C_{12}	22	$\mathbf{U_7} = U_3 + P_5$	C_{22}
11	$P_1 = \alpha \mathtt{IP}(A_{11} B_{11})$	Y	23	$P_3 = \alpha \mathtt{IP}(S_4 B_{22})$	C_{12}
12	$T_4 = T_2 - B_{21}$	X	24	$\mathbf{U_5} = U_4 + P_3$	C_{12}

Table 6: `AccLR` Schedule for $C \leftarrow \alpha A \times B + \beta C$ overwriting A and B with 2 temporaries, 4 recursive calls

input matrix. It also uses only two temporaries, but has to call the `O1R` schedule. The extra memory required by X and Y in table 6 is:

$$\max\left(\frac{m}{2}, \frac{k}{2}\right)\frac{n}{2} + \max\left\{\frac{m}{2}\frac{k}{2}, \frac{k}{2}\frac{n}{2}, \frac{m}{2}\frac{n}{2}\right\}.$$

Hence, using lemma 1:

$$E_6(m,k,n) < \frac{1}{3}\left(\max(m,k)n + \max\{mk, kn, mn\}\right). \tag{3}$$

The extra memory $E_7(m,k,n)$ required for table 7 in the top level of recursion is:

$$\frac{m}{2}\frac{k}{2} + \max\left(\frac{k}{2}, \frac{m}{2}\right)\frac{n}{2} + \max\left(E_7, E_5\right)\left(\frac{m}{2}, \frac{k}{2}, \frac{n}{2}\right).$$

Since the second term in the sum is greater than $\frac{m}{2}\frac{n}{2}$, we have $E_7 > E_5$ and:

$$E_7(m,k,n) < \frac{1}{3}\left(\frac{m}{2}\frac{k}{2} + \max\left(\frac{k}{2}, \frac{m}{2}\right)\frac{n}{2}\right).$$

Compared with the schedule of table 2, the possibility to overwrite the input matrices makes it possible to have further in place calls and replace recursive calls (with accumulation) by calls without accumulation. We show in theorem 3 that this enables us to almost compensate for the additional additions performed.

#	operation	loc.	#	operation	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	13	$P_2 = \alpha A_{12}B_{21} + \beta C_{11}$	C_{11}
2	$T_1 = B_{12} - B_{11}$	X	14	$S_2 = S_1 - A_{11}$	Y
3	$Z_2 = C_{21} - Z_1$	C_{21}	15	$P_6 = \alpha\texttt{O1R}(S_2T_2)$	B_{21}
4	$T_3 = B_{22} - B_{12}$	B_{12}	16	$S_4 = A_{12} - S_2$	Y
5	$S_3 = A_{11} - A_{21}$	Y	17	$U_2 = P_1 + P_6$	B_{21}
6	$P_7 = \alpha S_3T_3 + \beta Z_1$	C_{22}	18	$U_3 = U_2 + P_7$	C_{22}
7	$S_1 = A_{21} + A_{22}$	Y	19	$U_4 = U_2 + P_5$	B_{21}
8	$T_2 = B_{22} - T_1$	B_{12}	20	$\mathbf{U_6} = U_3 - P_4$	C_{21}
9	$P_5 = \alpha S_1T_1 + \beta C_{12}$	C_{12}	21	$\mathbf{U_1} = P_1 + P_2$	C_{11}
10	$T_4 = T_2 - B_{21}$	X	22	$\mathbf{U_7} = U_3 + P_5$	C_{22}
11	$P_4 = \alpha A_{22}T_4 - \beta Z_2$	C_{21}	23	$P_3 = \alpha\texttt{IP}(S_4B_{22})$	C_{12}
12	$P_1 = \alpha\texttt{O1R}(A_{11}B_{11})$	X	24	$\mathbf{U_5} = U_4 + P_3$	C_{12}

Table 7: `AccR` Schedule for $C \leftarrow \alpha A \times B + \beta C$ overwriting B with 2 temporaries, 4 recursive calls

5.3 The general case

We now examine the sizes of the temporary locations used, when the involved matrices are not of identical sizes. We want to adapt table 3 to make it work for general matrices.

First we remark that there is no way to compute a general matrix multiplication in place using Strassen-Winograd algorithm. Indeed, the only available memory is the one in C. Consider the following pebble game. No available pebble (i.e. some C_{ij}) can be put on any S_i or T_i since the size of C_{ij} may be too small for the purpose. No initial pebble can be moved since each node has at least two empty child nodes at the beginning. So no solution exists for this game.

However, table 3 is still valid for rectangular matrices as soon as both A and B are smaller in size than C, that is to say $k \max(m, n) \leqslant mn$ which can be rewritten as:

$$k \leqslant \min(m, n).$$

Table 6 is also valid with this assumption (since it calls `IP` internally). The problem remaining is when $k > \min(m, n)$. We thus propose the following algorithm 1:

Proposition 1. *Algorithm 1 computes the product $C = AB$ in place, overwriting A and B.*

Proof. It suffices to show that there is enough extra temporary space in $A_{(1)}$ and $B^{(1)}$, during the computation $C \leftarrow A_{(i)}B^{(i)} + C$ in the `for` loop.

On the one hand, the size of the extra memory used for the recursive calls in table 6 is smaller than the quantity from eq. (3). Since $k_0 \leqslant m, n$, that expression simplifies into:

$$E_6(m, k_0, n) < \frac{1}{3}(mn + mn) = \frac{2}{3}mn. \tag{4}$$

Algorithm 1 `IP0MM`: In-Place Overwrite Matrix Multiply

Input: A and B of resp. sizes $m \times k$ and $k \times n$.
Output: $C = A \times B$

1: Let $k_0 = min(m, n)$ and $K = \lceil k/k_0 \rceil$.
2: Split $A = \left[\begin{array}{c|c|c} A_{(1)} & \ldots & A_{(K)} \end{array}\right]$, $B = \left[\begin{array}{c} B^{(1)} \\ \hline \vdots \\ \hline B^{(K)} \end{array}\right]$ ▷ where $A_{(i)}$ (resp. $B^{(j)}$) has dimension $m \times k_0$ (resp. $k_0 \times n$), expect perhaps for the smaller K-th submatrices.
3: $C \leftarrow A_{(1)}B^{(1)}$ ▷ with alg. of table 3.
4: **for** $i = 2 \ldots K$ **do**
5: $\quad C \leftarrow A_{(i)}B^{(i)} + C$ ▷ with alg. of table 6 using $A_{(1)}$ and $B^{(1)}$ as temporary space.
6: **end for**

On the other hand, the available extra memory in the `for` loop is the sum of the sizes of $A_{(1)}$ and $B^{(1)}$, that is:

$$(m + n) \min(m, n). \tag{5}$$

Since the quantity (5) contains at least one mn term, it is larger than (4). Therefore, there is enough free space to compute the products with accumulations. □

Finally, table 8 shows how to perform an accumulation in the general case, overwriting only one of the inputs, with only two temporaries.

#	operation	loc.	#	operation	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	13	$P_2 = \alpha A_{12}B_{21} + \beta C_{11}$	C_{11}
2	$T_1 = B_{12} - B_{11}$	X	14	$\mathbf{U_1} = P_1 + P_2$	C_{11}
3	$Z_2 = C_{21} - Z_1$	C_{21}	15	$S_2 = S_1 - A_{11}$	Y
4	$T_3 = B_{22} - B_{12}$	B_{12}	16	$U_2 = \alpha S_2 T_2 + P_1$	X
5	$S_3 = A_{11} - A_{21}$	Y	17	$U_3 = U_2 + P_7$	C_{22}
6	$P_7 = \alpha S_3 T_3 + \beta Z_1$	C_{22}	18	$\mathbf{U_6} = U_3 - P_4$	C_{21}
7	$S_1 = A_{21} + A_{22}$	Y	19	$\mathbf{U_7} = U_3 + P_5$	C_{22}
8	$T_2 = B_{22} - T_1$	B_{12}	20	$U_4 = U_2 + P_5$	X
9	$P_5 = \alpha S_1 T_1 + \beta C_{12}$	C_{12}	21	$S_4 = A_{12} - S_2$	Y
10	$T_4 = T_2 - B_{21}$	X	22	$P_3 = \alpha S_4 B_{22}$	C_{12}
11	$P_4 = \alpha A_{22} T_4 - \beta Z_2$	C_{21}	23	$\mathbf{U_5} = U_4 + P_3$	C_{12}
12	$P_1 = \alpha A_{11} B_{11}$	X	24		

Table 8: `AccR` Schedule for $C \leftarrow \alpha A \times B + \beta C$ with 5 recursive calls, 2 temporaries and overwriting B

Here, the size of the extra temporaries is $\max\left(\frac{m}{2}, \frac{k}{2}\right)\frac{n}{2} + \frac{m}{2}\frac{k}{2}$ and $E_8(m, k, n)$

is equal to:

$$\max\left(\frac{m}{2},\frac{k}{2}\right)\frac{n}{2}+\frac{m}{2}\frac{k}{2}+\max\left(E_8,E_1\right)\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right).$$

If $m<k<n$ or $k<m<n$, then $E_8(m,k,n)<E_1(m,k,n)$:

$$\begin{aligned}E_8(m,k,n)&=\max\left(\frac{m}{2},\frac{k}{2}\right)\frac{n}{2}+\frac{m}{2}\frac{k}{2}+E_1\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right)\\&<\max\left(\frac{m}{2},\frac{k}{2}\right)\frac{n}{2}+\frac{m}{2}\frac{k}{2}+\frac{1}{3}\left(\frac{m}{2}\frac{n}{2}+\frac{k}{2}\frac{n}{2}\right).\end{aligned}$$

Otherwise $E_8(m,k,n)\geqslant E_1(m,k,n)$ and:

$$E_8(m,k,n)<\frac{1}{3}\left(\max\left(m,k\right)n+mk\right).$$

In the square case, this simplifies into $E_8(n,n,n)\leqslant\frac{2}{3}n^2$.

In addition, if the size of B is bigger than that of A, then one can store S_2 in e.g. B_{12} and separate the recursive call 16 into a multiplication and an addition, which reduces the arithmetic complexity. Otherwise, a scheduling with only 4 recursive calls exists too, but we need for instance to recompute S_4 at step 21.

6 Hybrid scheduling

By combining techniques from sections 3 and 5, we now propose in table 9 a hybrid algorithm that performs the computation $C\leftarrow\alpha A\times B+\beta C$ with constant input matrices A and B, with a lower extra memory requirement than the scheduling of [10] (table 2). We have to pay a price of order $n^2\log(n)$ additional operations, as we need to compute the temporary variable T_2 twice.

Again, the two temporary blocks X and Y respectively have dimensions $X_s=m/2\times\max\left(k/2,n/2\right)$ and $Y_s=k/2\times n/2$ so that:

$$E_9=Y_s+\max\left\{X_s+E_9,X_s+E_6,E_8\right\}\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right).$$

In all cases, $E_6+X_s\geqslant E_8$. But X_s+Y_s is not as large as the size of the two temporaries in table 6. We therefore get:

$$\begin{aligned}E_9(m,k,n)&=Y_s+X_s+E_6\left(\frac{m}{2},\frac{k}{2},\frac{n}{2}\right)\\&<\frac{m}{2}\max\left(\frac{k}{2},\frac{n}{2}\right)+\frac{k}{2}\frac{n}{2}+\frac{1}{3}\left(\max\left(\frac{m}{2},\frac{k}{2}\right)\frac{n}{2}\right.\\&\qquad\left.+\max\left\{\frac{m}{2}\frac{k}{2},\frac{k}{2}\frac{n}{2},\frac{m}{2}\frac{n}{2}\right\}\right).\end{aligned}$$

Assuming $m=n=k$, one gets $E_9(n,n,n)<\frac{2}{3}n^2$, which is smaller than the extra memory requirement of table 2.

#	operation	loc.	#	operation	loc.
1	$Z_1 = C_{22} - C_{12}$	C_{22}	14	$P_2 = \alpha A_{12}B_{21} + \beta C_{11}$	C_{11}
2	$Z_3 = C_{12} - C_{21}$	C_{12}	15	$\mathbf{U_1} = P_1 + P_2$	C_{11}
3	$S_1 = A_{21} + A_{22}$	X	16	$\mathbf{U_5} = U_2 + P_3$	C_{12}
4	$T_1 = B_{12} - B_{11}$	Y	17	$S_3 = A_{11} - A_{21}$	X
5	$P_5 = \alpha S_1 T_1 + \beta Z_3$	C_{12}	18	$T_3 = B_{22} - B_{12}$	Y
6	$S_2 = S_1 - A_{11}$	X	19	$U_3 = P_7 + U_2$	C_{21}
7	$T_2 = B_{22} - T_1$	Y		$= \alpha\texttt{AccLR}(S_3T_3 + U_2)$	
8	$P_6 = \alpha S_2 T_2 + \beta C_{21}$	C_{21}	20	$\mathbf{U_7} = U_3 + W_1$	C_{22}
9	$S_4 = A_{12} - S_2$	X	21	$T'_1 = B_{12} - B_{11}$	Y
10	$W_1 = P_5 + \beta Z_1$	C_{22}	22	$T'_2 = B_{22} - T'_1$	Y
11	$P_3 = \alpha S_4 B_{22} + P_5$	C_{12}	23	$T'_4 = T'_2 - B_{21}$	Y
12	$P_1 = \alpha A_{11}B_{11}$	X	24	$\mathbf{U_6} = U_3 - P_4$	C_{21}
13	$U_2 = P_6 + P_1$	C_{21}		$= -\alpha\texttt{AccR}(A_{22}T'_4 + U_3)$	

Table 9: Schedule for operation $C \leftarrow \alpha A \times B + \beta C$ with 2 temporaries

7 A sub-cubic in-place algorithm

Following the improvements of the previous section, the question was raised whether extra memory allocation was intrinsic to sub-cubic matrix multiplication algorithms. More precisely, is there a matrix multiplication algorithm computing $C \leftarrow A \times B$ in $\mathcal{O}\left(n^{\log_2 7}\right)$ arithmetic operations without extra memory allocation and without overwriting its input arguments? We show in this section that a combination of Winograd's algorithm and a classic block algorithm provides a positive answer. Furthermore this algorithm also improves the extra memory requirement for the product with accumulation $C \leftarrow \alpha A \times B + \beta C$.

7.1 The algorithm

The key idea is to split the result matrix C into four quadrants of dimension $n/2 \times n/2$. The first three quadrants C_{11}, C_{12} and C_{21} are computed using fast rectangular matrix multiplication, i.e. $2k/n$ standard Winograd multiplications on blocks of dimension $n/2 \times n/2$. The temporary memory for these computations is stored in C_{22}. Lastly, the block C_{22} is computed recursively up to a base case, as shown on algorithm 2. This base case, when the matrix is too small to benefit from the fast routine, is then computed with the classical matrix multiplication.

Theorem 1. *The complexity of algorithm 2 is:*

$$G(n,n) = 7.2n^{\log_2(7)} - 13n^2 + 6.8n$$

when $k = n$ *and* $FastThreshold = 1$.

Proof. Recall that the cost of Winograd's algorithm for square matrices is $W(n) = 6n^{\log_2 7} - 5n^2$ for the operation $C \leftarrow A \times B$ and $W_{\text{acc}}(n) = 6n^{\log_2 7} - 4n^2$

Algorithm 2 `IPMM`: In-Place Matrix Multiply

Input: A and B, of dimensions resp. $n \times k$ and $k \times n$ with k, n powers of 2 and $k \geqslant n$.
Input: An integer $FastThreshold \geqslant 1$.
Output: $C = A \times B$

1: **if** $k \leqslant FastThreshold$ **then**
2: $C = A \times B$ ▷ with the classical algorithm.
3: **else**
4: Split $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$, $A = \left[\begin{array}{c|c|c} A_{1,1} & \dots & A_{1,2k/n} \\ \hline A_{2,1} & \dots & A_{2,2k/n} \end{array}\right]$ and $B = \left[\begin{array}{c|c} B_{1,1} & B_{1,2} \\ \hline \vdots & \vdots \\ \hline B_{2k/n,1} & B_{2k/n,2} \end{array}\right]$ ▷ where each $A_{i,j}, B_{i,j}$ and $C_{i,j}$ have dimension $n/2 \times n/2$.
5: **do** ▷ with alg. of table 1 using C_{22} as temp. space
6: $C_{11} = A_{1,1}B_{1,1}$
7: $C_{12} = A_{1,1}B_{1,2}$
8: $C_{21} = A_{2,1}B_{1,1}$
9: **end do**
10: **for** $i = 2 \dots \frac{2k}{n}$ **do** ▷ with alg. of table 2 using C_{22} as temporary space:
11: $C_{11} = A_{1,i}B_{i,1} + C_{11}$
12: $C_{12} = A_{1,i}B_{i,2} + C_{12}$
13: $C_{21} = A_{2,i}B_{i,1} + C_{21}$
14: **end for**
15: $C_{22} = A_{2,*} \times B_{*,2}$ ▷ recursively using `IPMM`.
16: **end if**

for the operation $C \leftarrow A \times B + C$. The cost $G(n,k)$ of algorithm 2 is given by the relation

$$G(n,k) = 3W(n/2) + 3(2k/n - 1)W_{\text{acc}}(n/2) + G(n/2,k),$$

the base case being a classical dot product: $G(1,k) = 2k - 1$. Thus, $G(n,k) = 7.2kn^{\log_2(7)-1} - 12kn - n^2 + 34k/5$. □

Theorem 2. *For any m, n and k, algorithm 2 is in place.*

Proof. W.l.o.g, we assume that $m \geqslant n > 1$ (otherwise we could use the transpose). The exact amount of extra memory from algorithms in table 1 and 2 is respectively given by eq. (1) and (2).
If we cut B into p_i stripes at recursion level i, then the sizes for the involved submatrices of A (resp. B) are $m/2^i \times k/p_i$ (reps. $k/p_i \times n/2^i$). The lower right corner submatrix of C that we would like to use as temporary space has a size $m/2^i \times n/2^i$. Thus we need to ensure that the following inequality holds:

$$\max(E_1, E_2)\left(\frac{m}{2^i}, \frac{k}{p_i}, \frac{n}{2^i}\right) \leqslant \frac{m}{2^i}\frac{n}{2^i}. \tag{6}$$

It is clear that $E_1 < E_2$, which simplifies the previous inequality. Let us now write $K = k/p_i$, $M = m/2^i$ and $N = n/2^i$. We need to find, for every i an integer $p_i > 1$ so that eq. (6) holds. In other words, let us show that there exists some $K < k$ such that, for any (M, N), the inequality $E_2(M, K, N) \leqslant MN$ holds. Then the fact that $E(M, 2, N) < \frac{1}{3}(2M + 2N + MN) \leqslant \frac{1}{3}(4M + MN) \leqslant MN$ provides at least one such K.
As the requirements in algorithm 2 ensure that $k > N$ and $M = N$, there just remains to prove that $E(M, N, N) \leqslant MN$. Since $E(M, N, N) < \frac{1}{3}(2MN + N^2)$ and again $M \geqslant N$, algorithm 2 is indeed in place. □

Hence a fully in-place $\mathcal{O}\left(n^{\log_2 7}\right)$ algorithm is obtained for matrix multiplication. The overhead of this approach appears in the multiplicative constant of the leading term of the complexity, growing from 6 to 7.2.
This approach extends to the case of matrices with general dimensions, using e.g. peeling or padding techniques.
It is also useful if any sub-cubic algorithm is used instead of Winograd's. For instance, in the square case, one can use the product with accumulation in table 9 instead of table 2.

7.2 Reduced memory usage for the product with accumulation

In the case of computing the product with accumulation, the matrix C can no longer be used as temporary storage, and extra memory allocation cannot be avoided. Again we can use the idea of the classical block matrix multiplication at the higher level and call Winograd algorithm for the block multiplications. As in the previous subsection, C can be divided into four blocks and then the product can be made with 8 calls to Winograd algorithm for the smaller blocks, with only one extra temporary block of dimension $n/2 \times n/2$.
More generally, for square $n \times n$ matrices, C can be divided in t^2 blocks of dimension $\frac{n}{t} \times \frac{n}{t}$. Then one can compute each block with Winograd algorithm using only one extra memory chunk of size $(n/t)^2$. The complexity is changed to $R_t(n) = t^2 t W_{\text{acc}}(n/t)$, which is $R_t(n) = 6t^{3-\log_2(7)} n^{\log_2(7)} - 4tn^2$ for an accumulation product with Winograd's algorithm. Using the parameter t, one can then balance the memory usage and the extra arithmetic operations. For example, with $t = 2$,

$$R_2 = 6.857 n^{\log_2 7} - 8n^2 \quad \text{and} \quad \text{ExtraMem} = \frac{n^2}{4}$$

and with $t = 3$,

$$R_3 = 7.414 n^{\log_2 7} - 12n^2 \quad \text{and} \quad \text{ExtraMem} = \frac{n^2}{9}.$$

Note that one can use the algorithm of table 9 instead of the classical Winograd accumulation as the base case algorithm. Then the extra memory requirements drop to $\frac{2n^2}{3t^2}$ and the arithmetic complexity increases to $R_t(n) + t^{2-\log_2(3)} n^{\log_2(6)} - tn^2$.

8 Conclusion

With constant input matrices, we reduced the number of extra memory allocations for the operation $C \leftarrow \alpha A \times B + \beta C$ from n^2 to $\frac{2}{3}n^2$, by introducing two extra pre-additions. As shown below, the overhead induced by these supplementary additions is amortized by the gains in number of memory allocations.

If the input matrices can be overwritten, we proposed a fully *in-place* schedule for the operation $C \leftarrow A \times B$ without any extra operation. We also reduced the extra memory allocations for the operation $C \leftarrow \alpha A \times B + \beta C$ from n^2 to $\frac{2}{3}n^2$, by introducing only two extra pre-additions. We also proposed variants for the operation $C \leftarrow A \times B$, where only one of the input matrices is being overwritten and one temporary is required.

Some algorithms with an even more reduced memory usage, but with some increase in arithmetic complexity, are also shown. Table 10 gives a summary of the features of each schedule that has been presented. The complexities are given only for $m = k = n$ being a power of 2.

Theorem 3. *The arithmetic and memory complexities given in table 10 are correct.*

Proof. For the operation $A \times B$, the arithmetic complexity of the schedule of table 1 classically satisfies

$$\begin{cases} W_1(n) &= 7W_1(\frac{n}{2}) + 15\left(\frac{n}{2}\right)^2 \\ W_1(1) &= 1 \end{cases},$$

so that $W_1(n) = 6n^{\log_2(7)} - 5n^2$.

The schedule of table 1 requires

$$\begin{cases} M_1(n) &= 2\left(\frac{n}{2}\right)^2 + M_1\left(\frac{n}{2}\right) \\ M_1(1) &= 0 \end{cases}$$

extra memory space, which is $M_1(n) = \frac{2}{3}n^2$. Its total number of allocations satisfies $A_1(n) = 2\left(\frac{n}{2}\right)^2 + 7A_1\left(\frac{n}{2}\right)$ which is $A_1(n) = \frac{2}{3}(n^{\log_2(7)} - n^2)$.

The schedule of table 4 requires $M_4(n) = \left(\frac{n}{2}\right)^2 + M_4\left(\frac{n}{2}\right)$ extra memory space, which is $M_4(n) = \frac{1}{3}n^2$. Its total number of allocations satisfies $A_4(n) = \left(\frac{n}{2}\right)^2 + 5A_4\left(\frac{n}{2}\right)$ which is $A_4(n) = n^{\log_2(5)} - n^2$.

The schedule of table 5 requires the same amount of arithmetic operations or memory.

For $A \times B + \beta C$, the arithmetic complexity of [10] satisfies

$$W_2(n) = 5W_2\left(\frac{n}{2}\right) + 2W_1\left(\frac{n}{2}\right) + 14\left(\frac{n}{2}\right)^2,$$

so that $W_2(n) = 6n^{\log_2(7)} - 4n^2$; its number of extra memory satisfies $M_2(n) = 3\left(\frac{n}{2}\right)^2 + M_2\left(\frac{n}{2}\right)$, which is $M_2(n) = n^2$; its total number of allocations satisfies

	Algorithm	Input matrices	# of extra temporaries	total extra memory	total # of extra allocations	arithmetic complexity
$A \times B$	Table 2 [6]	Constant	2	$\frac{2}{3}n^2$	$\frac{2}{3}(n^{2.807} - n^2)$	$6n^{2.807} - 5n^2$
	Table 3	Both Overwritten	0	0	0	$6n^{2.807} - 5n^2$
	Table 4 or 5	A or B Overwritten	1	$\frac{1}{3}n^2$	$n^{2.322} - n^2$	$6n^{2.807} - 5n^2$
	7.1	Constant	0	0	0	$7.2n^{2.807} - 13n^2$
$\alpha A \times B + \beta C$	Table 2 [10]	Constant	3	n^2	$\frac{14}{3}n^{2.807} - 7n^{2.322} + \frac{7}{3}n^2$	$6n^{2.807} - 4n^2$
	Table 6	Both Overwritten	2	$\frac{2}{3}n^2$	$\frac{1}{2}n^2 \log_2(n)$	$6n^{2.807} - 4n^2 + \frac{1}{2}n^2 \log_2(n)$
	Table 7	B Overwritten	2	$\frac{2}{3}n^2$	$2n^{2.322} - 2n^2$	$6n^{2.807} - 4n^2 + \frac{1}{2}n^2 \log_2(n)$
	Table 9	Constant	2	$\frac{2}{3}n^2$	$\frac{2}{9}n^{2.807} + 2n^{2.322} - \frac{22}{9}n^2$	$6n^{2.807} - 4n^2 + \frac{4}{3}n^2 \log_2(n)$
	7.2	Constant	N/A	$\frac{1}{4}n^2$	$\frac{1}{4}n^2$	$6.857n^{2.807} - 8n^2$
	7.2	Constant	N/A	$\frac{1}{9}n^2$	$\frac{1}{9}n^2$	$7.414n^{2.807} - 12n^2$

Table 10: Complexities of the presented matrix multiplication schedules

$A_2(n) = 3\left(\frac{n}{2}\right)^2 + 5A_2\left(\frac{n}{2}\right) + 2A_1(n)$, which is

$$A_2(n) = \frac{1}{3}\left(14n^{\log_2(7)} + 7n^2 - 21n^{\log_2(5)}\right).$$

The arithmetic complexity of the schedule of table 6 satisfies

$$W_6(n) = 4W_6\left(\frac{n}{2}\right) + 3W_1\left(\frac{n}{2}\right) + 17\left(\frac{n}{2}\right)^2,$$

so that $W_6(n) = 6n^{\log_2(7)} - 4n^2 + \frac{1}{2}n^2\log_2(n)$; its number of extra memory satisfies $M_6(n) = 2\left(\frac{n}{2}\right)^2 + M_6\left(\frac{n}{2}\right)$, which is $M_6(n) = \frac{2}{3}n^2$; its total number of allocations satisfies $A_6(n) = 2\left(\frac{n}{2}\right)^2 + 4A_6\left(\frac{n}{2}\right)$, which is $A_6(n) = n^2 + \frac{1}{2}n^2\log_2(n)$.

The arithmetic complexity of the schedule of table 7 satisfies

$$W_7(n) = 4W_7\left(\frac{n}{2}\right) + W_1\left(\frac{n}{2}\right) + 2W_5\left(\frac{n}{2}\right) + 16\left(\frac{n}{2}\right)^2,$$

so that $W_7(n) = 6n^{\log_2(7)} - 4n^2 + \frac{1}{2}n^2\log_2(n)$; its number of extra memory satisfies $M_7(n) = 2\left(\frac{n}{2}\right)^2 + M_7\left(\frac{n}{2}\right)$, which is $M_7(n) = \frac{2}{3}n^2$; its total number of allocations satisfies $A_7(n) = 2\left(\frac{n}{2}\right)^2 + 4A_7\left(\frac{n}{2}\right) + 2A_5\left(\frac{n}{2}\right)$, which is $A_7(n) = 2n^{\log_2(5)} - 2n^2$.

The arithmetic complexity of the schedule of table 9 satisfies

$$W_9(n) = 4W_9\left(\frac{n}{2}\right) + W_1\left(\frac{n}{2}\right) + 2W_6\left(\frac{n}{2}\right) + 17\left(\frac{n}{2}\right)^2,$$

so that $W_9(n) = 6n^{\log_2(7)} - 4n^2 + \frac{4}{3}n^2\left(\log_2(n) - \frac{10}{3}\right) + \frac{4}{9}$; its number of extra memory satisfies $M_9(n) = 2\left(\frac{n}{2}\right)^2 + M_9\left(\frac{n}{2}\right)$, which is $M_9(n) = \frac{2}{3}n^2$; its total number of allocations satisfies $A_9(n) = 2\left(\frac{n}{2}\right)^2 + 4A_9\left(\frac{n}{2}\right) + A_1\left(\frac{n}{2}\right) + 2A_6\left(\frac{n}{2}\right)$, which is $A_9(n) = \frac{2}{9}n^{\log_2(7)} + 2n^{\log_2(5)} - \frac{22}{9}n^2 + \frac{2}{9}$. □

For instance, by adding up allocations and arithmetic operations in table 10, one sees that the overhead in arithmetic operations of the schedule of table 9 is somehow amortized by the decrease of memory allocations. Thus it makes it competitive with the algorithm of [10] as soon as $n \geqslant 10$.

Also, problems with dimensions that are not powers of two can be handled by combining the cuttings of algorithms 1 and 2 with peeling or padding techniques. Moreover, some cut-off can be set in order to stop the recursion and switch to the classical algorithm. The use of these cut-offs will in general decrease both the extra memory requirements and the arithmetic complexity overhead.

References

[1] Michael Bader and Christoph Zenger. Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra and its Applications*, 417(2–3):301–313, September 2006.

[2] David H. Bailey. Extra high speed matrix multiplication on the cray-2. *SIAM Journal on Scientific and Statistical Computing*, 9(3):603–607, 1988.

[3] Dario Bini and Victor Pan. *Polynomial and Matrix Computations, Volume 1: Fundamental Algorithms.* Birkhauser, Boston, 1994.

[4] Michael Clausen, Peter Bürgisser, and Mohammad A. Shokrollahi. *Algebraic Complexity Theory.* Springer, 1997.

[5] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[6] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics*, 110:1–10, 1994.

[7] Jean-Guillaume Dumas, Thierry Gautier, and Clément Pernet. Finite field linear algebra subroutines. In Teo Mora, editor, *ISSAC'2002*, pages 63–74. ACM Press, New York, July 2002.

[8] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. FFPACK: Finite field linear algebra package. In James Gutierrez, editor, *ISSAC'2004*, pages 119–126. ACM Press, New York, July 2004.

[9] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*. ACM Press and IEEE Computer Society Press, 1996. `www.supercomp.org/sc96/proceedings/SC96PROC/JACOBSON/`.

[10] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Strassen's algorithm for matrix multiplication : Modeling analysis, and implementation. Technical report, Center for Computing Sciences, November 1996. CCS-TR-96-17.

[11] Oscar H. Ibarra, Shlomo Moran, and Roger Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3(1):45–56, March 1982.

[12] Claude-Pierre Jeannerod, Clément Pernet, and Arne Storjohann. Rank sensitive, in-place fast computation of the echelon form. Technical report, 2007.

[13] Antoni Kreczmar. On memory requirements of Strassen's algorithms. In A. Mazurkiewicz, editor, *Proceedings of the 5th Symposium on Mathematical Foundations of Computer Science*, volume 45 of *LNCS*, pages 404–407, Gdańsk, Poland, September 1976. Springer.

[14] Julian Laderman, Victor Pan, and Xuan-He Sha. On practical algorithms for accelerated matrix multiplication. *Linear Algebra and its Applications*, 162–164:557–588, 1992.

[15] Clément Pernet. Implementation of Winograd's fast matrix multiplication over finite fields using ATLAS level 3 BLAS. Technical report, Laboratoire Informatique et Distribution, July 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.

[16] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[17] Shmuel Winograd. On multiplication of 2 X 2 matrices. *Linear Algebra and Application*, 4:381–388, 1971.