The Number of the Beast: Reducing Additions in Fast Matrix Multiplication Algorithms for Dimensions up to 666

Erik Mårtensson[1,2] and Paul Stankovski Wagner[1]

[1]Dept. of Electrical and Information Technology, Lund University, Lund, Sweden {erik.martensson,paul.stankovski_wagner}@eit.lth.se
[2]Advenica AB, Malmö, Sweden

Abstract

While a naive algorithm for multiplying two 2×2 matrices requires eight multiplications and four additions, Strassen showed how to compute the same matrix product using seven multiplications and 18 additions. Winograd reduced the number of additions to 15, which was assumed to be optimal. However, by introducing a change of basis, Karstadt and Schwartz showed how to lower the number of additions to 12, which they further showed to be optimal within this generalized Karstadt-Schwartz (KS) framework. Since the cost of the change of basis is smaller than one addition (per each recursive level), it is disregarded in this cost metric.

In this work we present improved methods for classical optimization of the number of additions in Strassen-type matrix multiplication schemes, but for larger matrix sizes, and without including any change of basis.

We find specific examples where our methods beat the best instances found within the KS framework, the most impressive one being Laderman's algorithm for multiplying 3×3 matrices, which we reduce from the naive 98 additions to 62, compared to 74 in the KS framework. We indicate that our approach performs better compared to previous work within the KS framework, as the matrix dimensions increase.

We additionally apply our algorithms to another reference set of algorithms due to Moosbauer and Kauers for which we do not have results in the KS framework as a comparison. For parameters (n, m, k), when multiplying an $(n \times m)$ matrix with an $(m \times k)$ matrix, the schoolbook algorithm uses $nk(m-1)$ additions. Using the reference set of algorithms we find that our algorithm leads to an optimized number of additions of roughly $cnk(m-1)$, where c is a small constant which is independent of the dimensions.

We further provide concrete and practical implementations of our methods that are very efficient for dimensions including (and surpassing) the 666 limit, i.e. $(n, m, k) = (6, 6, 6)$, in our reference set of fast multiplication algorithms.

1 Introduction

Multiplying two $n \times n$ matrices directly from the definition requires $\mathcal{O}(n^{\log_2(8)}) = \mathcal{O}(n^3)$ operations. By finding a way of multiplying two 2×2 matrices using only seven instead of eight scalar multiplications, in 1969 Strassen surprisingly showed that it is possible to perform matrix multiplication faster than by directly using the definition, presenting an algorithm requiring only $\mathcal{O}(n^{\log_2(7)})$ operations [1]. Seven multiplications were shown to be optimal in [2].

One branch of research has aimed to improve the asymptotic performance of Strassen's algorithm, gradually leading to asymptotically faster algorithms, culminating in the current record of $\mathcal{O}(n^{2.371552})$ [3]. Although theoretically interesting, with the exception of Pan's algorithm [4], these algorithms are not fast in practical implementation.

Another branch of research aims at improving upon Strassen's algorithm, while still finding practically fast algorithms. This is essentially done by searching for ways to multiply matrices larger than 2×2, using as few scalar multiplications as possible. Recent improvements include [5, 6, 7].

When employing Strassen-type algorithms in practice, only a few recursive levels are typically utilized, and naive multiplication is performed below some (size) threshold [8]. The larger the dimensions of the Strassen-type algorithm, the fewer recursion levels are used. Thus, while Strassen-type algorithms are asymptotically superior, they tend to be faster only by a fairly small constant factor in practice, compared to their naive counterpart algorithm.

1.1 Minimizing the Number of Additions

Strassen's algorithm originally required 18 additions [1]. Winograd improved this number to 15 [9][1], which due to [9, 10] was thought to be optimal. In terms of the complexity of running Strassen's algorithm, this improvement reduces the constant before the dominant term from seven to six.

Surprisingly, in [11] it was shown that by performing a change of basis – a method not considered in the optimality results of [9, 10] – the number of additions can be reduced to 12. This number was shown to be optimal within this updated framework in [11]. This, in turn, reduces the constant before the dominant term from six to five.

In [12] Schwartz and Vaknin compared efficient implementations of naive matrix multiplication, Winograd's form for matrix multiplication and a method using a change of basis to optimize the number of additions. Roughly speaking, the improvement in runtime of going from the naive algorithm to Winograd's form, is similar to the improvement of going from Winograd's form to a method using a change of basis. Although the latter does not improve the asymptotic behavior, it shows that minimizing the number of additions matters in practical implementation. Also interesting to note in [12] is that their Strassen-type algorithm beats the naive one already at the small dimension $n = 96$.

[1]Probert introduces the scheme in [9], but acknowledges Winograd as the inventor of it.

For larger matrices, optimizing the number of additions becomes more complicated, but also more important, as the number of recursive calls that can be made before resorting to the naive algorithm is lower. While many papers have been written on minimizing the number of multiplications, the study of the number of additions for fast matrix multiplication algorithms is relatively sparse. The approach of [11] was applied to a set of larger matrices in [13], which thus constitutes our main point of comparison in this paper.

In this work we study the type of classical optimization of additions that lead to 15 additions in [9], but for larger matrix sizes. We do better, worse, or identical to [13], depending on the dimensions and the precise scheme. Most interestingly and surprisingly, we reduce the number of additions for Laderman's algorithm [14], from 74 in [13] down to 62, which we show to be optimal in our framework. We indicate that, in general, compared to [13], we do better the larger the matrix dimensions are.

Beyond the algorithms covered by [13], we also apply our algorithms to the recent set of algorithms found in [5, 6]. We give an approximate linear model for the optimized number of additions needed, as a function of the matrix dimensions.

2 Method

2.1 Background

Given matrices $\mathbf{A} \in R^{n\times m}$, $\mathbf{B} \in R^{m\times k}$ and $\mathbf{C} = \mathbf{AB} \in R^{n\times k}$, for some ring R. An r-rank algorithm for computing $\mathbf{C}$ performs the following computations.

$$M_1 = (\alpha_{11}^{(1)}A_{11} + \alpha_{12}^{(1)}A_{12} + \cdots)(\beta_{11}^{(1)}B_{11} + \beta_{12}^{(1)}B_{12} + \cdots), \tag{1}$$

$$\vdots \tag{2}$$

$$M_r = (\alpha_{11}^{(r)}A_{11} + \alpha_{12}^{(r)}A_{12} + \cdots)(\beta_{11}^{(r)}B_{11} + \beta_{12}^{(r)}B_{12} + \cdots), \tag{3}$$

$$C_{11} = \gamma_{11}^{(1)}M_1 + \gamma_{11}^{(2)}M_2 + \cdots + \gamma_{11}^{(r)}M_r, \tag{4}$$

$$\vdots \tag{5}$$

$$C_{nk} = \gamma_{nk}^{(1)}M_1 + \gamma_{nk}^{(2)}M_2 + \cdots + \gamma_{nk}^{(r)}M_r, \tag{6}$$

where $\alpha_{ij}^{(l)}, \beta_{ij}^{(l)}, \gamma_{ij}^{(l)} \in K$. While some works have studied fractional coefficients or larger integer coefficients, most works let $K = \{-1, 0, 1\}$. Recent works [7, 5, 6] show improvements for some dimensions when $R = \mathbb{Z}_2$, which corresponds to setting $K = \{0, 1\}$. This setting has applications in algebraic and algorithmic cryptanalysis [15, 16]. Given the dimensions (n, m, k), most previous works have focused on minimizing the value r. This work starts with a scheme specified as in (1)-(6), minimized with respect to the number r, and tries to compute the entire matrix $\mathbf{C}$, using a minimal number of additions.

It is common to describe a matrix multiplication algorithm in tensor form, as in, for example [7, 5, 6]. For the context of our work however, the form of (1)-(6)

is more convenient to work with. Translation between the two representations is straightforward.

2.2 Warm-up Example

Consider the Winograd form of [9]. Given $\mathbf{A}$ and $\mathbf{B}$, we compute

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \mathbf{C} = \mathbf{AB} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}. \tag{7}$$

We can find $\mathbf{C}$ using 7 multiplications and 24 additions by computing

$$\begin{aligned}
M_1 &= A_{11}B_{11}, \\
M_2 &= A_{12}B_{21}, \\
M_3 &= (A_{21} - A_{11})(B_{12} - B_{22}), \\
M_4 &= (A_{21} + A_{22})(B_{12} - B_{11}), \\
M_5 &= (A_{21} + A_{22} - A_{11})(B_{11} + B_{22} - B_{12}), \\
M_6 &= (A_{11} + A_{12} - A_{21} - A_{22})B_{22}, \\
M_7 &= A_{22}(B_{12} + B_{21} - B_{11} - B_{22}),
\end{aligned}$$

and

$$\begin{aligned}
C_{11} &= M_1 + M_2, \\
C_{12} &= M_1 \qquad\qquad + M_4 + M_5 + M_6, \\
C_{21} &= M_1 \quad + M_3 \quad + M_5 \quad + M_7, \\
C_{22} &= M_1 \quad + M_3 + M_4 + M_5.
\end{aligned}$$

However, some additions are re-used. By creating the temporary variables

$$\begin{aligned}
M_8 &= M_1 + M_5, \\
M_9 &= M_8 + M_3,
\end{aligned}$$

we can now compute

$$\begin{aligned}
C_{11} &= M_1 + M_2, \\
C_{12} &= \qquad M_4 \quad + M_6 \quad + M_8, \\
C_{21} &= \qquad\qquad + M_7 \quad + M_9, \\
C_{22} &= \qquad M_4 \qquad\qquad + M_9,
\end{aligned}$$

saving 3 additions. Using the same techniques we can save 3 additions when creating all the "A"-factors and 3 additions when computing all the "B"-factors, in the M_i terms above. This leads to a total of 15 additions, which is optimal within this framework [9, 10].

However, by performing a change of basis [11] manages to need only 12 additions, which they in turn show to be optimal in their extended framework. The authors also study the performance of this approach on larger matrices, where they do not make similar optimality claims.

2.3 The General Problem

In this work we study the performance of classical optimization methods like in Section 2.2, but on matrices larger than the 2×2 ones, to see how they scale with the problem size compared to [13]. All three problems above can be stated as instances of the following general problem. Given

$$s_1 = g_{11}a_1 + \cdots g_{1q}a_q, \tag{8}$$

$$\vdots \tag{9}$$

$$s_p = g_{p1}a_1 + \cdots g_{pq}a_q, \tag{10}$$

where $g_{ij} \in \{-1, 0, 1\}$. Compute $s_1, \ldots, s_p$ using a minimum number of additions and subtractions. We can obviously represent the problem on matrix form as

$$G = \begin{bmatrix} g_{11} & \cdots & g_{1q} \\ \vdots & \ddots & \vdots \\ g_{p1} & \cdots & g_{pq} \end{bmatrix}. \tag{11}$$

Now, the optimization process from Section 2.2 generally corresponds to repeatedly choosing two columns in (11) and forming a new column by either adding or subtracting values from one column to the other.

2.3.1 Adding Two Columns

Consider column i and column j. By convention we always let $i < j$. Let us denote the action of adding columns i and j by $(i, j, +1)$. For each κ such that $g_{\kappa i} = g_{\kappa j} \neq 0$ we say that we have a matching pattern between column i and column j on row κ. Now consider the total number of such matching patterns. Let us assume that the number is at least two. By replacing all instances of $a_i + a_j$ by $a_{q+1} = a_i + a_j$ we have now saved additions compared to naively performing the addition once for each row. The number of additions we save is one less than the number of matching patterns. We refer to a substitution that saves at least one addition as a *saving action*.

In terms of our matrix representation in (11), performing the action of adding columns i and j corresponds to setting $g_{\kappa i} := 0$ and $g_{\kappa j} := 0$, for all matching rows κ. Then we create a new column, which has the value 1 for each row κ where $g_{\kappa i} = g_{\kappa j} = 1$, and value -1 for each row κ where $g_{\kappa i} = g_{\kappa j} = -1$, and 0 for the remaining rows.

2.3.2 Subtracting Two Columns

Now, let us consider subtracting columns i and j. By convention we always let $i < j$. Let us denote the action of subtracting columns i and j by $(i, j, -1)$. For each κ such that $g_{\kappa i} = -g_{\kappa j} \neq 0$ we say that we have a matching pattern between column i and column j on row κ. Just like for addition we say that the number of such patterns is equal to the number of matching patterns for the action of subtracting i and j. By replacing all instances of $a_i - a_j$ by $a_{q+1} = a_i - a_j$ we have now saved additions compared to naively performing the subtractions once for each row. The number of additions we save is one less than the number of matching patterns. Also for subtraction, we refer to a substitution that saves at least one addition as a *saving action*.

In terms of our matrix representation in (11), performing the action of subtracting columns i and j corresponds to setting $g_{\kappa i} := 0$ and $g_{\kappa j} := 0$, for all matching rows. Then we create a new column, which has the value 1 for each row κ where $g_{\kappa i} = 1$ and $g_{\kappa j} = -1$, and value -1 for each row κ where $g_{\kappa i} = -1$ and $g_{\kappa j} = +1$, and 0 in the remaining positions.

2.4 On the Potential of a Matrix

A key concept in our approach to solving the problem of minimizing the number of additions is the potential of a matrix. To introduce it, we first need a couple of definitions.

Definition 1 (Number of Matching Patterns)**.** *Let G be a matrix, and let (i, j, σ) with $i < j$ be an action. The number of matching patterns in G with respect to the action (i, j, σ) is then denoted*

$$M\big(G, (i, j, \sigma)\big).$$

Performing the action (i, j, σ) saves $M\big(G, (i, j, \sigma)\big) - 1$ additions.

Definition 2 (Matrix Resulting from Saving Action)**.** *Given a matrix G and a saving action (i, j, σ). We use $f\big(G, (i, j, \sigma)\big)$ to denote the matrix G' resulting from performing the action (i, j, σ) on the matrix G.*

Definition 3 (Set of Saving Actions)**.** *Let G be a matrix, and let (i, j, σ) with $i < j$ be an action. The set of all saving actions on G is then denoted*

$$S(G) := \Big\{ (i, j, \sigma) \,\big|\, M\big(G, (i, j, \sigma)\big) \geq 2 \Big\}.$$

Given such a set, we now use $P(G)$ to denote the potential of the matrix, which we define as follows.

Definition 4 (Potential)**.** *Let G be a matrix. The potential of G, denoted $P(G)$, is*

$$\sum_{(i,j,\sigma) \in S(G)} \Big(M\big(G, (i, j, \sigma)\big) - 1 \Big) = \sum_{(i,j,\sigma) \in S(G)} M\big(G, (i, j, \sigma)\big) - |S(G)|. \quad (12)$$

The potential is the sum of the number of saved additions for all saving actions. This is an upper limit of how many additions we can save, compared to performing all additions naively. This number is typically an overly optimistic limit, as performing an action tends to significantly reduce the number of matching patterns for other actions. It is nonetheless a useful metric to estimate how many more additions can be saved for a given matrix G. We are now ready to introduce our main theorem of this work.

Theorem 1. *Given a saving action (i, j, σ) applied to a matrix G. The resulting matrix $G' = f(G, (i, j, \sigma))$ has potential reduced by at least the number of additions saved by performing the action (i, j, σ).*

Proof of Theorem 1. Consider the action (i, j, σ) applied to G and the set of saving actions $S(G) = \{(k, l, \sigma')|M(G, (k, l, \sigma')) \geq 2\}$[2]. Let us denote the index of the new column by t. Let $m = M(G, (i, j, \sigma))$ denote the number of matching patterns for the action (i, j, σ).

Performing the action (i, j, σ) means that for all the rows in i and j with a matching pattern, the corresponding positions in G' are set to 0. For the non-zero entries in the new column, either the corresponding positions in G' are equal to the corresponding values in columns i and j of G (in case $\sigma = +1$) or the one in column i is equal and one in column j is the opposite sign (in case $\sigma = -1$).

The difference between G and G' means that the only possible saving actions for G' are the saving actions for G, plus actions of the form (l, t, σ'), where (k, l, σ') is a saving action of G, $k \in \{i, j\}$ and $l \notin \{i, j\}$[3].

To show that the potential of G' is lower than the potential of G by at least $m - 1$ additions, let us divide the set $S(G)$ into three categories, depending on if they have zero, one or two overlaps in indices with the action (i, j, σ).

- For the action (i, j, σ) itself in G', this action obviously has zero matching patterns and the contribution to the potential with respect to this action is thus $m - 1$ additions less for G' than for G. The action $(i, j, -\sigma)$ can potentially have fewer matching patterns in G' than in G and never more.

- For an action (k, l, σ'), where $k \notin \{i, j\}$ and $l \notin \{i, j\}$, the number of matching patterns is unchanged. Thus, this action contributes to the potential in the same way in G' as in G.

- Finally, let us cover the case of one of two indices matching. Let us w.l.o.g. assume that $k = i$ and $l > j$. All other cases where either k or l matches with either i or j - but the other one does not - can be dealt with analogously. First of all, $M(G', (k, l, \sigma')) \leq M(G, (k, l, \sigma'))$, as we have set some positions of column k in G' to zero. Consider the non-zero entries in column t. We have two cases.

[2]Notice that for readability, within this proof we let k denote a column of the matrices G and G', instead of a dimension of a matrix.

[3]or actions of the form (k, t, σ'), where (k, l, σ') is a saving action of G, $k \notin \{i, j\}$ and $l \in \{i, j\}$.

- If $M(G', (l, t, \sigma')) \leq 1$, then we have not created a new saving action involving column t. The number of saved additions of performing (k, l, σ') on G' is then upper limited by performing the same action on G.
- So let us assume that $M(G', (l, t, \sigma')) \geq 2$. For everyone of these matching patterns, there are corresponding positions in columns i and j of G' that are set to zero. No matter the sign σ of the current action, for each matching pattern on a row κ, the action copies the value of $g_{\kappa i}$ in G to $g_{\kappa t}$ in G'. Thus, for every match between columns l and t, the same value as in column t of G' appears in column i of G. Therefore, every matching pattern of (l, t, σ') in G' had a corresponding matching pattern in (i, l, σ'). It follows that $M((G, (i, l, \sigma'))) \geq M(G', (i, l, \sigma')) + M(G', (l, t, \sigma'))$. Thus, the contribution of (i, l, σ') plus the new action (l, t, σ') to the potential of G'; is smaller than or equal to the original contribution of (i, l, t) to G.

As the contribution of the action (i, j, σ) itself is reduced by $m - 1$ and we have a non-increased contribution corresponding to every other saving action, $P(G) \geq P(G') + m - 1$, just as the theorem stated. □

An immediate implication of Theorem 1 is the confirmation that the potential of a matrix is indeed an upper limit of how many additions can be saved on that matrix.

2.4.1 A Note on the Remaining Number of Saving Actions

Typically, performing a saving action also lowers the number of remaining saving actions, but this is not always the case. The following small example shows that the number of saving actions can even increase after performing a saving action. Consider performing the action $(0, 1, +1)$ to G below, forming G'.

$$G = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & -1 \\ 0 & 1 & -1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & -1 & 1 \\ -1 & -1 & 1 \end{pmatrix} \to G' = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & -1 \end{pmatrix}.$$

Before the action we had $S(G) = \{(0, 1, +1), (0, 2, +1), (0, 2, -1), (1, 2, +1), (1, 2, -1)\}$. After the action we have $S(G') = \{(0, 2, +1), (0, 2, -1), (1, 2, +1),$

$(1, 2, -1), (2, 3, +1), (2, 3, -1)\}$. The total number of saving actions has increased by one. Notice however that the potential of G' is 6, while the potential is 15 for G. Thus, while the number of saving actions is higher, the potential of the new matrix is lower by at least 3 (the number of additions that the action $(0, 1, +1)$ saved), like Theorem 1 says.

2.5 Formulation as Longest Path in a Directed Tree

The problem of finding an algorithm that saves a maximum number of additions can be represented as finding the longest path in a directed tree.

Each node represents as a matrix. The root of the tree is the initial matrix. For each node there is an edge for each saving action. Each edge leads to a node representing the matrix resulting from performing that action. The weight of the edge is equal to the number of additions the action saves. A leaf corresponds to a matrix that has no saving actions.

As the potential of the resulting matrix gets lower after each saving action – according to Theorem 1 – we are guaranteed reach a leaf after a finite number of steps[4]; leading to a finite number of saved additions.

While finding the longest path problem for directed trees is generally an easy problem, it becomes tricky in our setting due to the enormous size of the tree.

2.6 Algorithms

We have developed two types of algorithms for solving our longest path problem. One algorithm that finds an optimal solution, but is too slow as the dimension grows. One that finds a good approximate solution much more quickly.

2.6.1 Exhaustive Search

The most obvious approach to solving the problem is to simply search the whole tree exhaustively for the best solution. The complexity of this approach is proportional to the number of nodes and the number of edges of the tree respectively.

While exhaustive search has the advantage of finding an optimal solution - within our framework - it has the disadvantage of being too slow already at fairly moderate problem sizes. However, it is useful to benchmark the two greedy approaches in Section 2.6.2 on small dimensions.

We speed up this process slightly by first finding an approximate solution using the greedy algorithms covered in Section 2.6.2. Throughout the search we discard a branch as soon as we can determine that it is unable to beat the performance of the greedy approach. This can be done since the potential of a node is an upper limit of the number of additions that can be saved starting from that node, according to Theorem 1.

A potential further improvement of brute-force search is to take advantage of the fact that in some cases we have multiple equivalent matrices, which we

[4]Since by the time the potential is at zero, we cannot perform any more saving actions.

in the current description calculate independently. Assume for example that we perform the action (i_1, j_1, σ_1) followed by the action (i_2, j_2, σ_2), where i_1 and j_1 are both different from both i_2 and j_2. If we would perform the two actions in the opposite order, then we would end up with two matrices that are identical, except for the order of the two new columns. We only need to apply exhaustive search on one of these matrices. We leave figuring out the details of this improvement for further study.

2.6.2 Greedy Algorithms - Vanilla and Potential

A straightforward approach for solving the problem is a greedy one where we in each step choose the action that saves the most number of additions. In case of a tie, we simply choose whichever such action we found first. This approach we refer to as Greedy Vanilla.

An improvement over Greedy Vanilla is to take into consideration both the number of additions an action saves and the potential of the matrix that remains after performing the action. In case of a tie with respect to this metric, we simply choose the first best action we found. More formally, let $G' = f\big(g, (i, j, \sigma)\big)$. We now score the action (i, j, σ) as

$$M\big(G, (i, j, \sigma)\big) - 1 + \alpha P(G'). \tag{13}$$

Here α is a parameter which we can vary. The larger value of α, the more the algorithm prioritizes the potential of the remaining matrix over the number of saved additions by the current action. We refer to this greedy approach as Greedy Potential.

While the idea is very simple, it performs surprisingly well. For all the matrix multiplication schemes where we are able to perform exhaustive search, Greedy Potential matches the exhaustive search. See Section 3 for more details.

3 Results

We detail how our algorithm compares to that of [13] for a range of algorithms in Section 3.1. In Section 3.2 we cover how our algorithms perform on a set of schemes with larger dimensions, with the schemes taken from [5, 6]. We briefly mention the speed of - and the potential improvements of the speed of - the implementation of our algorithm for optimizing the number of additions in Section 3.3. Finally, we illustrate the performance of the resulting optimized matrix multiplication algorithms in Sections 3.4 and 3.5

3.1 Comparison Against [13]

The number of additions needed for our approaches compared to [13] is covered in Table 1 below, for matrix multiplication schemes where $(n, m, k) = (3, 3, 3)$. All schemes use 23 scalar multiplications. We distinguish the different Grey schemes by the number of non-zero elements in the scheme specifications. The naive number of additions corresponds to the number of additions needed if performing no optimization of the number of additions. Notice that both Laderman and Smirnov in their respective works point out that it is possible to reduce the number of additions. The method of [13] requires a change of basis. While not affecting the leading coefficient, the cost of it is not 0. Thus, we mark our result as better also for Smirnov's algorithm. For the Grey-221 scheme we were unable to run the exhaustive search due to the size of the search tree.

A couple of interesting observations can be made. Our method performs particularly well for Laderman's algorithm. Our algorithm's performance compared to [13] depends quite a lot on the precise scheme. Notice that for both methods, the number of additions varies a lot with the scheme, even though all the schemes perform the same number of scalar multiplications. Also note that the naive number of additions does not predict the optimized number of additions very well, for either of the two approaches.

In [17] the authors show how to drastically reduce the leading coefficients of matrix multiplication algorithms in some cases. Their method corresponds to the number of additions of a naive matrix multiplication algorithm. However, as pointed out in [13], the method introduces a large number of lower-order terms, meaning that its practical performance is unclear. Another method for reducing the coefficient for the dominant complexity term is shown in [18], but also in this case the practical runtime seems to not be faster due to larger memory footprint and communication costs.

		without change of basis			with
	Naïve additions	Greedy Vanilla	Greedy Potential	Exhaustive	[13]
Laderman [14]	98	70	**62**	**62**	74
Smirnov [19]	84	**68**	**68**	**68**	68
Grey-221 [20]	166	85	82	-	73
Grey-152 [20]	97	64	63	63	61
Grey-144 [20]	89	68	68	68	65
Grey-143 [20]	88	66	**64**	**64**	65
Grey-142 [20]	87	**65**	**65**	**65**	66

Table 1: Comparison of the number of additions for 3×3 matrix multiplication algorithms. Recall that [13] assumes the KS framework while we do not. Results where we match the results of [13] are marked in bold, as our approach does not have the cost of changing basis.

We also have a look at other schemes where we have reference results from [13] in Table 2. For each scheme the first three digits refer to the matrix dimensions (n, m, k) and the last refer to the number of scalar multiplications. For all but the last two algorithms we were able to run the brute-force algorithm to verify that our methods are indeed optimal, within our framework. Notice the clear trend that our algorithm tends to do better compared to [13], the larger the dimension is.

Notice that for both Table 1 and 2 for all cases where we were able to perform an exhaustive search, we were able to confirm that Greedy Potential is optimal within our framework.

		without change of basis			with
	Naïve additions	Greedy Vanilla	Greedy Potential	Exhaustive	[13]
Strassen-222-7 [1]	18	18	18	18	12
Winograd-222-7 [9]	24	15	15	15	12
Grey-322-11 [20]	22	21	21	21	18
Grey-422 [20]	48	37	37	37	28
Grey-323-15 [20]	64	44	44	44	39
Grey-522-18 [20]	53	40	40	40	32
Grey-423-20 [20]	92	59	58	58	-
Grey-234-20 [20]	96	63	62	62	58
Grey-433-29 [20]	164	**99**	**98**	-	102
Grey-343-29 [20]	167	**103**	**101**	-	109

Table 2: Comparison of the number of additions. Recall that [13] assumes the KS framework while we do not.

3.2 Some Results for Larger Dimensions

For larger dimensions we fail to find algorithms to compare with against the approach of [13]. Here we compile two tables of how our methods perform on two sets of algorithms from [5, 6]. While we do not have reference results to compare against, it is interesting to see how our method scales with the dimension. The naive number of additions correspond to performing the algorithms of [5, 6] without performing any optimization on the number of additions. Notice of course that neither of these works attempted to minimize the number of additions.

Below in Table 3 algorithms performed over $\mathbb{Z}$ are covered, while Table 4 covers algorithms performed over $\mathbb{Z}_2$. Notice for both tables that even with specific values for $(\alpha_A, \alpha_B, \alpha_C)$, the results might vary slightly, depending on which order the algorithm processes the saving actions[5].

Kauers and Moosbauer searched for matrix multiplication algorithms with coefficients in $\mathbb{Z}_2$, resulting in the algorithms labeled as Moosbauer-mod2 in Table 4 below. Using these solutions they attempted transform them into more general algorithms over $\mathbb{Z}$, resulting in the algorithms labeled Moosbauer in Table 3. Most of these algorithms feature coefficients outside of $\{-1, 0, 1\}$[6].

[5]In case of a tie by the metric in (13), the algorithm chooses the first such saving action it finds.

[6]While these coefficients do not change the asymptotics, it is a bit misleading to just assume that multiplying submatrices with constant factors is an operation without any cost. While multiplying with a factor like 2 can be done faster than adding two submatrices, the

To handle coefficients outside of $\{-1, 0, 1\}$ we have to slightly generalize our saving actions. When adding columns i and j we now have a matching pattern on row κ if $g_{\kappa i} = g_{\kappa j} \neq 0$. When subtracting columns i and j we now have a matching pattern on row κ if $g_{\kappa i} = -g_{\kappa j} \neq 0$. For both operations on rows κ with matching patterns we store the value $g_{\kappa i}$ in the new column. For the other positions in the new column we store the value 0.

The two algorithms in Table 4 labeled as Adaptive refer to the two algorithms over $\mathbb{Z}_2$ introduced in [6]. In Table 4 the Moosbauer-mod0 rows use algorithms from Table 3 that are initially defined over a larger ring, but then converts them to binary.

In both tables we managed to confirm using exhaustive search that the solutions that we found using the greedy algorithm were optimal (within our framework) for Moosbauer's algorithms 223-11, 224-14 and 233-15.

3.3 On the Runtime of the Implementation

Running a single-threaded, non-optimized Python implementation with parameters according to Table 4 for the $(n, m, k) = (6, 6, 6)$ takes takes around 1,000 seconds on a basic laptop[7]. The runtime is significantly lower for the smaller problem sizes. Running our corresponding single-threaded, non-optimized C implementation takes about 42 seconds[8]. A parallelized and optimized implementation would likely be orders of magnitude faster.

3.4 Illustrating the Algorithm Performance

In Figures 1 we plot the performance of the algorithms from Table 3 and Table 4. Each data point corresponds to one algorithm from [5, 6]. For an nmk algorithm the x value is equal to $nk(m-1)$, which corresponds to the number of additions the naive algorithm uses. The y value corresponds to the number of additions needed for the naive implementations of an algorithm from [5, 6] or the corresponding algorithm where the number of additions has been minimized using Greedy Potential, respectively.

Figure 1a covers algorithms from Table 3 and Figure 1b covers algorithms from Table 4.

We notice that in both cases the there is roughly a linear relation between the number of additions for the naive algorithm and the corresponding number of additions for the fast algorithm, both before and after optimizing the number of additions. The deviations from the linear model are stronger in the algorithms over $\mathbb{Z}$ than $\mathbb{Z}_2$ and stronger before optimizing the number of additions than after doing so.

cost is still proportional to the size of the submatrix, affecting the coefficient of the dominant complexity term like an addition.

[7]The implementation can be found at `https://github.com/ErikMaartensson/fmm_add_reduction`.

[8]The implementation can be found at `https://github.com/werekorren/fmm_add_reduction`.

For Figure 1a the slopes of the two lines are roughly 15.5 and 4.77 before and after optimization respectively. For Figure 1b the respective slopes are 18.2 and 6.61. Just like the recent trend of finding more efficient algorithms in terms of scalar multiplications in $\mathbb{Z}_2$, it seems like those algorithms also lead to a lower number of additions. Notice that these two improvements over algorithms over $\mathbb{Z}$ are synergetic.

	Naïve	Greedy		Improvement		
	additions	Vanilla	Potential	ops	%	$\alpha_A, \alpha_B, \alpha_C$
Moosbauer						
223-11	42	31	31	11	26.2	0, 0, 0
224-14	99	66	65	34	34.3	0, 0, 0.2
225-18	130	79	77	53	40.7	0, 0.1, 0.1
233-15	76	48	48	28	36.8	0, 0, 0
234-20	161	100	99	62	38.5	0.4, 0, 0
235-25	199	106	103	96	48.2	0.3, 0.3, 0
244-26	319	175	173	146	45.8	0, 0.21, 0
245-33	359	177	172	187	52.1	0.2, 0, 0.1
255-40	682	323	313	369	54.1	0.1, 0.12, 0.06
266-56	1,133	442	419	714	63.0	0.2, 0.1, 0.08
334-29	202	107	105	97	48.0	0.1, 0.5, 0
335-36	370	181	176	194	52.4	0.1, 0.3, 0.1
336-42	728	274	264	464	63.7	0, 0.136, 0.065
344-38	401	205	198	203	50.6	0.2, 0.15, 0.2
345-47	714	284	276	438	61.3	0, 0.11, 0.08
346-56	795	325	319	476	59.9	0.25, 0, 0.09
355-58	689	331	326	363	52.7	0, 0.174, 0.07
356-71	950	399	386	564	59.4	0.255, 0.11, 0.14
444-49	629	335	325	304	48.3	0.22, 0.24, 0.12
445-62	1,175	501	476	699	59.5	0.18, 0.133, 0.08
446-74	1,187	488	467	720	60.7	0, 0.11, 0.06
455-76	1,107	469	451	656	59.2	0.17, 0.1, 0.104
456-93	1,522	567	542	980	64.4	0.21, 0.132, 0.053
555-97	1,710	729	701	1,009	59.0	0.1, 0.161, 0.061
555-97a	2,208	886	852	1,356	61.4	0.1, 0.087, 0.056
556-116	1,990	747	712	1,278	64.2	0.01, 0.119, 0.07

Table 3: Algorithms over $\mathbb{Z}$. Comparison of how many additions schemes from [5, 6] need naïvely vs. after our optimizations.

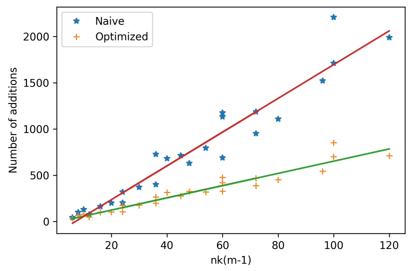
3.5 Examples Illustrating Performance Differences

For a concrete example in which Greedy Potential performs better than Greedy Vanilla, see Tables 5, 6 and 7 below.

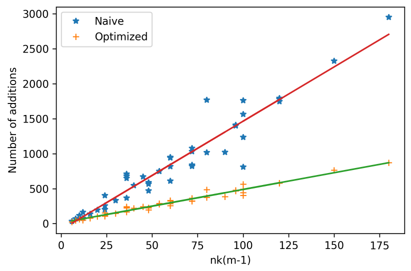
Table 5 has a complete specification of how Laderman's algorithm performs a multiplication of two 3×3 matrices using 23 multiplications and 98 naive

	Naïve	Greedy		Improvement		
	additions	Vanilla	Potential	ops	%	$\alpha_A, \alpha_B, \alpha_C$
Moosbauer-mod0						
223-11	35	23	23	12	34.3	0, 0, 0
224-14	69	46	43	26	37.7	0, 0.335, 0.005
225-18	116	63	62	54	46.6	0, 0.255, 0
233-15	76	48	48	28	36.8	0, 0, 0
234-20	139	79	76	63	45.3	0, 0.145, 0.005
235-25	196	101	100	96	49.0	0.25, 0, 0
244-26	255	115	113	142	55.7	0, 0, 0.005
255-40	546	226	218	328	60.1	0.135, 0.005, 0.055
266-56	942	338	330	612	65.0	0, 0.07, 0.05
334-29	201	107	104	97	48.3	0.01, 0.5, 0.01
335-36	332	152	144	188	56.6	0.005, 0.005, 0.09
336-42	681	244	234	447	65.6	0.2375, 0.105, 0.0575
344-38	365	179	166	199	54.5	0.18, 0.185, 0.2
345-47	668	254	242	426	63.8	0, 0.005, 0.07
346-56	749	299	286	463	61.8	0.225, 0.005, 0.07
355-58	608	266	255	353	58.1	0.005, 0.145, 0.075
356-71	839	338	324	515	61.3	0.12, 0.005, 0.105
444-49	470	200	196	274	58.3	0.0005, 0.084, 0.0005
445-62	940	349	323	617	65.6	0.1715, 0.116, 0.055
446-74	1,031	380	363	668	64.8	0.005, 0.12 ,0.05
455-76	1,015	388	372	643	63.3	0.145, 0.11, 0.005
456-93	1,398	500	478	920	65.8	0.1825, 0.1275, 0.032
555-97	1,233	462	440	793	64.3	0.14, 0.115, 0.07
555-97a	1,760	592	564	1,196	68.0	0.065, 0.085 , 0.025
556-116	1,791	604	577	1,214	67.8	0.1376, 0.1085, 0.0436
Moosbauer-mod2						
226-21	162	74	71	91	56.2	0, 0, 0.105
236-30	401	159	149	252	62.8	0.37, 0.125, 0.08
246-39	709	253	242	467	65.9	0, 0.01, 0.07
256-48	588	241	227	361	61.3	0.21, 0.105, 0.08
266-56	948	332	327	621	65.5	0, 0, 0.05
336-42	650	231	222	428	65.8	0.26, 0.09, 0.06
346-56	747	293	278	469	62.8	0.18, 0.105, 0.06
356-71	819	335	321	498	60.8	0.24, 0.105, 0.055
366-85	1,020	415	385	635	62.3	0.22, 0.06, 0.03
444-47	567	234	223	344	60.7	0.28, 0.13, 0.01
445-60	817	293	281	536	65.6	0.167, 0.132, 0.046
446-74	1,077	377	360	717	66.6	0.08, 0.065, 0.055
455-76	1,015	388	372	643	63.3	0.15, 0.11, 0.01
456-93	1,409	506	465	944	67.0	0.23, 0.123, 0.042
466-116	1,745	621	581	1,164	66.7	0.0002, 0.094, 0.0312
555-95	808	418	402	406	50.2	0.17, 0.21, 0.005
556-116	1,777	601	580	1,197	67.4	0.13, 0.005, 0.06
566-144	2,325	815	764	1,561	76.1	0.1, 0.095, 0.0334
666-164	2,948	925	871	2,077	70.5	0.1296, 0.103, 1/30
Adaptive						
455-73	1,766	516	480	1,286	72.8	0.01, 0.072, 0.036
555-94	1,563	509	492	1,071	68.5	0.11, 0.132 , 0.0255

Table 4: Algorithms over $\mathbb{Z}_2$. Comparison of how many additions are needed naively vs. after our optimizations, with scheme taken from [5, 6].



(a) Arithmetic performed over $\mathbb{Z}$.



(b) Arithmetic performed over $\mathbb{Z}_2$.

Figure 1: The number of additions for Strassen-type algorithms, as a function of the number of additions for a naive algorithm with the same dimensions.

additions. Table 6 shows an algorithm needing only 70 additions, resulting from optimizing the number of additions using Greedy Vanilla. Finally, Table 7 shows an algorithm needing only 62 additions, resulting from optimizing the number of additions using Greedy Potential.

4 Conclusions and Future Work

We have shown that the classical method for optimizing the number of additions in Strassen-type algorithms can outperform the best results found (so far) within the KS framework, with a seeming general trend that the classical method performs better when the dimensions of the matrices are higher.

An interesting idea to study is if a hybrid between classical optimization and the method based on basis change can outperform both approaches.

For the case of coefficients outside of $\{-1, 0, 1\}$ our current approach only considers substitutions of the forms $a_i + a_j$ and $a_i - a_j$. We currently do not consider substitutions of the form $d_1 a_i + d_2 a_j$, where at least one of the coefficients d_1, d_2 are outside of $\{-1, 0, 1\}$. Including such substitutions may potentially lead to improved performance, at least for some schemes.

Except for the smallest schemes, for most cases, it may be possible to improve upon our results even within our framework of not using a change of basis. More advanced heuristics may be specifically tailored to improve performance. We leave this for future work.

Even for Strassen's algorithm, the number of optimized additions depends on which of the equivalent representations is chosen. The work of [5] showed that there is a systematic way of finding an almost unlimited number of non-equivalent algorithms with the same number of multiplications. None of the schemes in this work are designed with a minimal (reduced) number of additions in mind. Thus, by searching both throughout the flip graph and then among the equivalent representations within a chosen node of the flip graph, it should be possible to find algorithms where the reduced number of additions is significantly

$$\begin{array}{rcl}
M_0 &=& (A_0 + A_1 + A_2 - A_3 - A_4 - A_7 - A_8)B_4 \\
M_1 &=& (A_0 - A_3)(-B_1 + B_4) \\
M_2 &=& A_4(-B_0 + B_1 + B_3 - B_4 - B_5 - B_6 + B_8) \\
M_3 &=& (-A_0 + A_3 + A_4)(B_0 - B_1 + B_4) \\
M_4 &=& (A_3 + A_4)(-B_0 + B_1) \\
M_5 &=& A_0 B_0 \\
M_6 &=& (-A_0 + A_6 + A_7)(B_0 - B_2 + B_5) \\
M_7 &=& (-A_0 + A_6)(B_2 - B_5) \\
M_8 &=& (A_6 + A_7)(-B_0 + B_2) \\
M_9 &=& (A_0 + A_1 + A_2 - A_4 - A_5 - A_6 - A_7)B_5 \\
M_{10} &=& A_7(-B_0 + B_2 + B_3 - B_4 - B_5 - B_6 + B_7) \\
M_{11} &=& (-A_2 + A_7 + A_8)(B_4 + B_6 - B_7) \\
M_{12} &=& (A_2 - A_8)(B_4 - B_7) \\
M_{13} &=& A_2 B_6 \\
M_{14} &=& (A_7 + A_8)(-B_6 + B_7) \\
M_{15} &=& (-A_2 + A_4 + A_5)(B_5 + B_6 - B_8) \\
M_{16} &=& (A_2 - A_5)(B_5 - B_8) \\
M_{17} &=& (A_4 + A_5)(-B_6 + B_8) \\
M_{18} &=& A_1 B_3 \\
M_{19} &=& A_5 B_7 \\
M_{20} &=& A_3 B_2 \\
M_{21} &=& A_6 B_1 \\
M_{22} &=& A_8 B_8
\end{array}$$

$$\begin{array}{rcl}
C_0 &=& M_5 + M_{13} + M_{18} \\
C_1 &=& M_0 + M_3 + M_4 + M_5 + M_{11} + M_{13} + M_{14} \\
C_2 &=& M_5 + M_6 + M_8 + M_9 + M_{13} + M_{15} + M_{17} \\
C_3 &=& M_1 + M_2 + M_3 + M_5 + M_{13} + M_{15} + M_{16} \\
C_4 &=& M_1 + M_3 + M_4 + M_5 + M_{19} \\
C_5 &=& M_{13} + M_{15} + M_{16} + M_{17} + M_{20} \\
C_6 &=& M_5 + M_6 + M_7 + M_{10} + M_{11} + M_{12} + M_{13} \\
C_7 &=& M_{11} + M_{12} + M_{13} + M_{14} + M_{21} \\
C_8 &=& M_5 + M_6 + M_7 + M_8 + M_{22}
\end{array}$$

Table 5: The original Laderman multiplication scheme with 98 additions. Here we use the compact indexing notation

$$\begin{pmatrix} C_0 & C_1 & C_2 \\ C_3 & C_4 & C_5 \\ C_6 & C_7 & C_8 \end{pmatrix} = \mathbf{C} = \mathbf{AB} = \begin{pmatrix} A_0 & A_1 & A_2 \\ A_3 & A_4 & A_5 \\ A_6 & A_7 & A_8 \end{pmatrix} \begin{pmatrix} B_0 & B_1 & B_2 \\ B_3 & B_4 & B_5 \\ B_6 & B_7 & B_8 \end{pmatrix}.$$

$$
\begin{aligned}
t_0 &= A_0 - A_3\\
t_1 &= A_0 - A_6\\
t_2 &= A_2 - A_4\\
t_3 &= A_7 + A_8\\
t_4 &= A_1 + t_2\\
t_5 &= A_7 - t_1\\[1em]
u_0 &= B_0 - B_1\\
u_1 &= B_0 - B_2\\
u_2 &= B_4 + B_6\\
u_3 &= B_5 - B_8\\
u_4 &= B_3 - u_2\\
u_5 &= B_5 + u_1\\[1em]
M_0 &= (t_0 - t_3 + t_4)B_4\\
M_1 &= t_0(-B_1 + B_4)\\
M_2 &= A_4(-u_0 - u_3 + u_4)\\
M_3 &= (A_4 - t_0)(B_4 + u_0)\\
M_4 &= (A_3 + A_4)(-u_0)\\
M_5 &= A_0B_0\\
M_6 &= t_5u_5\\
M_7 &= -t_1(B_2 - B_5)\\
M_8 &= (A_6 + A_7)(-u_1)\\
M_9 &= (-A_5 + t_4 - t_5)B_5\\
M_{10} &= A_7(B_7 + u_4 - u_5)\\
M_{11} &= (-A_2 + t_3)(-B_7 + u_2)\\
M_{12} &= (A_2 - A_8)(B_4 - B_7)
\end{aligned}
$$

$$
\begin{aligned}
M_{13} &= A_2B_6\\
M_{14} &= t_3(-B_6 + B_7)\\
M_{15} &= (A_5 - t_2)(B_6 + u_3)\\
M_{16} &= (A_2 - A_5)u_3\\
M_{17} &= (A_4 + A_5)(-B_6 + B_8)\\
M_{18} &= A_1B_3\\
M_{19} &= A_5B_7\\
M_{20} &= A_3B_2\\
M_{21} &= A_6B_1\\
M_{22} &= A_8B_8\\[1em]
v_0 &= M_5 + M_{13}\\
v_1 &= M_1 + M_3\\
v_2 &= M_6 + M_7\\
v_3 &= M_{11} + M_{12}\\
v_4 &= M_{15} + M_{16}\\[1em]
C_0 &= M_{18} + v_0\\
C_1 &= M_0 + M_3 + M_4 + M_{11} + M_{14} + v_0\\
C_2 &= M_6 + M_8 + M_9 + M_{15} + M_{17} + v_0\\
C_3 &= M_2 + v_0 + v_1 + v_4\\
C_4 &= M_4 + M_5 + M_{19} + v_1\\
C_5 &= M_{13} + M_{17} + M_{20} + v_4\\
C_6 &= M_{10} + v_0 + v_2 + v_3\\
C_7 &= M_{13} + M_{14} + M_{21} + v_3\\
C_8 &= M_5 + M_8 + M_{22} + v_2
\end{aligned}
$$

Table 6: Laderman multiplication scheme reduced from 98 to 70 additions with Greedy Vanilla. Here we use the compact indexing notation

$$\begin{pmatrix} C_0 & C_1 & C_2\\ C_3 & C_4 & C_5\\ C_6 & C_7 & C_8 \end{pmatrix} = \mathbf{C} = \mathbf{AB} = \begin{pmatrix} A_0 & A_1 & A_2\\ A_3 & A_4 & A_5\\ A_6 & A_7 & A_8 \end{pmatrix} \begin{pmatrix} B_0 & B_1 & B_2\\ B_3 & B_4 & B_5\\ B_6 & B_7 & B_8 \end{pmatrix}.$$

$$\begin{aligned}
t_0 &= A_0 - A_3\\
t_1 &= A_0 - A_6\\
t_2 &= A_2 - A_5\\
t_3 &= A_2 - A_8\\
t_4 &= A_4 - t_0\\
t_5 &= A_4 - t_2\\
t_6 &= A_7 - t_1\\
t_7 &= A_7 - t_3
\end{aligned}$$

$$\begin{aligned}
u_0 &= B_0 - B_1\\
u_1 &= B_0 - B_2\\
u_2 &= B_4 - B_7\\
u_3 &= B_5 - B_8\\
u_4 &= B_4 + u_0\\
u_5 &= B_5 + u_1\\
u_6 &= B_6 + u_2\\
u_7 &= B_6 + u_3
\end{aligned}$$

$$\begin{aligned}
M_0 &= (A_1 - t_4 - t_7)B_4\\
M_1 &= t_0(-B_1 + B_4)\\
M_2 &= A_4(B_3 - u_4 - u_7)\\
M_3 &= t_4 u_4\\
M_4 &= (A_3 + A_4)(-u_0)\\
M_5 &= A_0 B_0\\
M_6 &= t_6 u_5\\
M_7 &= -t_1(B_2 - B_5)\\
M_8 &= (A_6 + A_7)(-u_1)\\
M_9 &= (A_1 - t_5 - t_6)B_5\\
M_{10} &= A_7(B_3 - u_5 - u_6)\\
M_{11} &= t_7 u_6\\
M_{12} &= t_3 u_2\\
M_{13} &= A_2 B_6\\
M_{14} &= (A_7 + A_8)(-B_6 + B_7)\\
M_{15} &= t_5 u_7\\
M_{16} &= t_2 u_3\\
M_{17} &= (A_4 + A_5)(-B_6 + B_8)\\
M_{18} &= A_1 B_3\\
M_{19} &= A_5 B_7\\
M_{20} &= A_3 B_2\\
M_{21} &= A_6 B_1\\
M_{22} &= A_8 B_8
\end{aligned}$$

$$\begin{aligned}
v_0 &= M_3 + M_5\\
v_1 &= M_5 + M_6\\
v_2 &= M_{11} + M_{13}\\
v_3 &= M_{13} + M_{15}\\
v_4 &= M_1 + v_0\\
v_5 &= M_7 + v_1\\
v_6 &= M_{12} + v_2\\
v_7 &= M_{16} + v_3
\end{aligned}$$

$$\begin{aligned}
C_0 &= M_5 + M_{13} + M_{18}\\
C_1 &= M_0 + M_4 + M_{14} + v_0 + v_2\\
C_2 &= M_8 + M_9 + M_{17} + v_1 + v_3\\
C_3 &= M_2 + v_4 + v_7\\
C_4 &= M_4 + M_{19} + v_4\\
C_5 &= M_{17} + M_{20} + v_7\\
C_6 &= M_{10} + v_5 + v_6\\
C_7 &= M_{14} + M_{21} + v_6\\
C_8 &= M_8 + M_{22} + v_5
\end{aligned}$$

Table 7: Laderman multiplication scheme reduced from 98 to 62 additions with Greedy Potential using parameters $(\alpha_A, \alpha_B, \alpha_C) = (0.1, 0.1, 0.2)$. Here we use the compact indexing notation

$$\begin{pmatrix} C_0 & C_1 & C_2\\ C_3 & C_4 & C_5\\ C_6 & C_7 & C_8 \end{pmatrix} = \mathbf{C} = \mathbf{AB} = \begin{pmatrix} A_0 & A_1 & A_2\\ A_3 & A_4 & A_5\\ A_6 & A_7 & A_8 \end{pmatrix} \begin{pmatrix} B_0 & B_1 & B_2\\ B_3 & B_4 & B_5\\ B_6 & B_7 & B_8 \end{pmatrix}.$$

lower than those reported in this work.

The problem of minimizing the number of additions in the systems we consider is both quite general and simple. Although it has not been systematically studied in the context of matrix multiplication, it might have been studied elsewhere previously.

References

[1] Volker Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13(4):354–356, aug 1969.

[2] S. Winograd. On multiplication of 2 × 2 matrices. *Linear Algebra and its Applications*, 4(4):381–388, 1971.

[3] Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. New Bounds for Matrix Multiplication: from Alpha to Omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835.

[4] V. Ya. Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *19th Annual Symposium on Foundations of Computer Science (FOCS 1978)*, pages 166–176, 1978.

[5] Manuel Kauers and Jakob Moosbauer. Flip graphs for matrix multiplication. In *Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation (ISSAC '23)*, page 381–388, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Yamato Arai, Yuma Ichikawa, and Koji Hukushima. Adaptive flip graph algorithm for matrix multiplication. In *Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation*, ISSAC '24, page 292–298, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610:47–53, 2022.

[8] Jianyu Huang, Tyler Smith, Greg Henry, and Robert van de Geijn. Strassen's algorithm reloaded. pages 690–701, 11 2016.

[9] Robert L. Probert. On the additive complexity of matrix multiplication. *SIAM Journal on Computing*, 5(2):187–203, 1976.

[10] Nader H. Bshouty. On the additive complexity of 2 × 2 matrix multiplication. *Information Processing Letters*, 56(6):329–335, 1995.

[11] Elaye Karstadt and Oded Schwartz. Matrix multiplication, a little faster. *J. ACM*, 67(1), jan 2020.

[12] Oded Schwartz and Noa Vaknin. Pebbling game and alternative basis for high performance matrix multiplication. *SIAM Journal on Scientific Computing*, 45(6):C277–C303, 2023.

[13] Gal Beniamini, Nathan Cheng, Olga Holtz, Elaye Karstadt, and Oded Schwartz. Sparsifying the operators of fast matrix multiplication algorithms, 2020.

[14] Julian D. Laderman. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. *Bulletin of the American Mathematical Society*, 82(1):126 – 128, 1976.

[15] Gregory V. Bard. *Algebraic Cryptanalysis.* Springer Publishing Company, Incorporated, 1st edition, 2009.

[16] Antoine Joux. *Algorithmic Cryptanalysis.* Chapman & Hall/CRC, 1st edition, 2009.

[17] Gal Beniamini and Oded Schwartz. Faster matrix multiplication via sparse decomposition. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19')*, page 11–22, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Murat Cenk and M. Anwar Hasan. On the arithmetic complexity of strassen-like matrix multiplications. *Journal of Symbolic Computation*, 80:484–501, 2017.

[19] A.V Smirnov. The bilinear complexity and practical algorithms for matrix multiplication. *Comput. Math. and Math. Phys.*, 53:1781–1795, 2013.

[20] Austin R. Benson and Grey Ballard. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices*, 50(8):42–53, jan 2015.