On Why and How to Minimize the Arithmetic Complexity of Fast Matrix Multiplication Algorithms

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

[1]Lund University, Lund, Sweden {erik.martensson,paul.stankovski_wagner}@eit.lth.se
[2]Advenica AB, Malmö, Sweden

Abstract. Naively multiplying two 2×2 matrices requires eight multiplications and four additions. Strassen showed how to perform the same computation using seven multiplications and 18 additions. By changing basis, Karstadt and Schwartz lowered the number of additions to 12, which they showed to be optimal within this generalized Karstadt-Schwartz (KS) framework.

We present improved methods for optimizing the number of additions in Strassen-type matrix multiplication schemes for larger matrix sizes, and without any change of basis. Considering fast matrix generation process holistically as consisting of scheme generation and addition reduction, we discuss how to optimize both parts of this pipeline. We indicate that minimizing additions during the generation process is advantageous.

We implement of our methods and use them to optimize the number of additions for schemes with dimensions up to $(n, m, k) = (5, 7, 10)$. Our methods can handle larger dimensions than (what has been published within) the KS framework.

We compare our results against solutions within the KS framework on several large sets of schemes. We show that our method performs better relative to the KS framework, the larger the matrix dimensions are. We also apply our algorithms to a large number of schemes where we do not have apples-to-apples results in the KS framework as a comparison.

We optimize the arithmetic complexity for two sets of thousands of schemes with the same rank. The number of additions needed after optimization roughly follows a normal distribution. Thus, we need to generate many solutions to minimize arithmetic complexity.

Finally, our results on a large set of schemes and our extensive list of future research directions make for a valuable benchmark and facilitate future study of the arithmetic complexity of fast matrix multiplication.

* This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Parts of this work were presented at the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA25) [28].

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 [46]. Seven multiplications were shown to be optimal in [48].

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 $\mathcal{O}(n^{2.371339})$ [1]. Although theoretically interesting, with the exception of Pan's algorithm [31], 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 slightly larger than 2×2, using as few scalar multiplications as possible. Recent improvements include methods based on flip graph search [2, 20, 22, 26, 33, 35] and machine learning methods [12, 30]. Sedoglavic showed in [40] how to combine schemes into efficient schemes of larger dimension. He keeps a table of the schemes with the lowest ranks for dimensions from $(2, 2, 2)$ to $(32, 32, 32)$ at [41].

When employing Strassen-type algorithms in practice, only a few recursive levels are typically utilized, and naive multiplication is performed below some (size) threshold [16]. 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.

Copyright © 2026
Copyright for this paper is retained by authors

1.1 Why Minimize the Number of Additions?

Strassen's algorithm originally required 18 additions [46]. Winograd improved this number to 15 [36][1], which due to [8, 36] 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 [19] it was shown that by performing a change of basis – a method not considered in the optimality results of [8, 36] – the number of additions can be reduced to 12. This number was shown to be optimal within this updated framework in [19]. This, in turn, reduces the constant before the dominant term from six to five.

In [39] 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 [39] is that their change-of-basis algorithm beats the naive one already at the small dimension $n = 96$.

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. Although 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 [19] was applied to a set of larger matrices in [4]. Recently, the approach was extended to a larger set of schemes in [14]. These two works constitute our main point of comparison in this paper.

1.2 How to Minimize Additions

In this work we study the type of classical optimization[2] of additions that lead to 15 additions in [36], but for larger matrix sizes. We do better, worse, or identical to [4, 14], depending on the dimensions and the precise scheme. Surprisingly, we reduce the number of additions for Laderman's algorithm [25], from 74 in [4] down to 62, which we show to be optimal in our framework. In general, compared to [4, 14], we do better the larger the dimensions of the matrix.

Beyond the algorithms covered by [4, 14], we also apply our algorithms to two sets of algorithms found in [2, 20, 26, 21] and [12, 30], respectively. We show that the optimized number of additions approximately grows linearly with the product of the matrix dimensions.

Recently, in [34], some extensions to the method presented in [28] were made, showing that further improvements are still possible and strengthening our case that reduction without change of basis becomes more competitive against methods with change of basis, as the matrix dimensions increase. In terms of reducing the number of additions for a specific scheme, our main focus in this work is to make a comprehensive and practical comparison of addition reduction with and without a change of basis.

1.2.1 Searching Among a Large Set of Schemes

We find two sets of thousands of matrix multiplication schemes with the same rank [12, 13]. For both sets we optimize the number of additions for every scheme and study the distribution of the reduced number of additions. We find that the optimized number of additions roughly follows a normal distribution. Thus, in order to find a scheme with a truly low optimized number of additions, we need a large number of schemes.

The 3×3 Setting. In terms of arithmetic complexity, we manage to find three schemes for multiplying 3×3 matrices using 61 additions, all matching the best result found within the KS framework [4]. We can find two more (sets of) works that optimize the number of additions by searching a large set of schemes in the 3×3 setting [29, 32, 45][3]. The works [29, 45] both generate a large set of schemes for multiplying 3×3 matrices by modeling a scheme as a neural network, finding a scheme with only 60 and only 59 additions, respectively. Perminov on the other hand starts from a 3×3 scheme and then searches its vast flip graph until he finds a scheme with a minimal arithmetic complexity, resulting in a scheme with only 58 additions.

Notice that the method of Perminov is different from what we envision here, starting from a single scheme and searching a neighborhood, instead of generating schemes independently. It would, of course, be interesting to study the merging of Perminov's local flip graph search with some methods of generating many solutions, like in [29, 45].

Both of these methods result in fewer additions than any of the schemes we study in this work, highlighting the importance of how schemes are generated and possibly also post-processed.

[1] Probert introduces the scheme in [36], but acknowledges Winograd as the inventor of it.

[2] The type of optimization used here is sometimes referred to as Common Subexpression Elimination (CSE).

[3] These works only provide a single scheme each, though.

Copyright © 2026
Copyright for this paper is retained by authors

The 4×4 Setting. In the 4×4 setting, we find a scheme using 162/161 additions in the standard/binary metric, which is better than the 165 additions achieved by performing two layers of Strassen's algorithm on Winograd's form, which was the previous state-of-the-art for general 4×4 schemes of rank 49.

There are also general schemes of rank 48 introduced using complex coefficients in [18, 30] and reduced to rational coefficients in [10, 11], but the corresponding arithmetic complexities of 341 and 355 in [10] and [11] respectively, likely make these schemes slow in practical implementations.

In the $\mathbb{Z}_2$ setting there are schemes of rank 47 [12, 20]. For comparison, we achieve 199 additions for the rank-47 scheme found by AlphaTensor [12], see Table 3.6.

1.3 Holistic FMM Pipeline Remarks Finding a practically efficient fast matrix multiplication (FMM) scheme with both low rank and low arithmetic complexity boils down to a two-step pipeline, as illustrated in Figure 1.1. We need a procedure to generate a large number of schemes of low rank. Next, we apply reduction of additions to the entire set of generated schemes. The scheme with the lowest number of additions is now a good candidate for an efficient matrix multiplication algorithm.

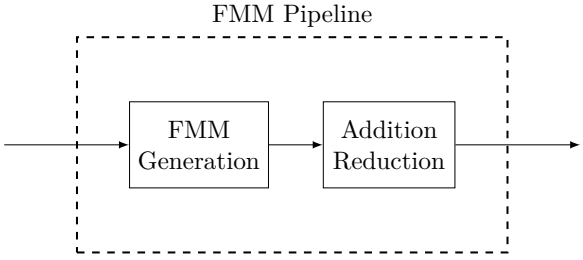


Figure 1.1: Pipeline for generation of implementation efficient fast matrix multiplication algorithms.

The sets we study in this work were generated using SAT solving and reinforcement learning, respectively. We leave details of how to construct a general approach for FMM generation for future research. We have this as one of many future research directions in our extensive list in Section 4.

2 Method We introduce our optimization method, starting with some necessary background information.

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

$$\begin{aligned}
M_1 &= (\alpha_{11}^{(1)}A_{11} + \alpha_{12}^{(1)}A_{12} + \cdots)(\beta_{11}^{(1)}B_{11} + \beta_{12}^{(1)}B_{12} + \cdots),\\
&\vdots\\
M_r &= (\alpha_{11}^{(r)}A_{11} + \alpha_{12}^{(r)}A_{12} + \cdots)(\beta_{11}^{(r)}B_{11} + \beta_{12}^{(r)}B_{12} + \cdots),\\
C_{11} &= \gamma_{11}^{(1)}M_1 + \gamma_{11}^{(2)}M_2 + \cdots + \gamma_{11}^{(r)}M_r,\\
&\vdots\\
C_{nk} &= \gamma_{nk}^{(1)}M_1 + \gamma_{nk}^{(2)}M_2 + \cdots + \gamma_{nk}^{(r)}M_r,
\end{aligned} \tag{2.1}$$

where $\alpha_{ij}^{(l)}, \beta_{ij}^{(l)}, \gamma_{ij}^{(l)} \in K$. In this work we mainly consider letting $K = \{-1, 0, 1\}$, which we call ternary and denote by $\mathbb{T}$. Multiple works [2, 12, 20, 26] show improvements for some dimensions when $R = \mathbb{Z}_2$, corresponding to setting $K = \{0, 1\}$, which we also consider. This setting has applications in algebraic and algorithmic cryptanalysis [3, 17]. For many dimensions improved solutions were found by considering larger sets of K, such as arbitrary integers [20, 21], rational numbers [22, 42, 43] or even complex numbers [18, 30].

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 (2.1), minimized with respect to r, and computes $\mathbf{C}$ using as few additions as possible.

It is common to describe a fast matrix multiplication algorithm in tensor form, as in, for example [2, 12, 20, 26]. However, for the context of our work, the form of (2.1) is more convenient to work with. Translation between the two representations is straightforward.

2.2 Warm-up Example Consider the Winograd form of [36]. Given $\mathbf{A}$ and $\mathbf{B}$, we compute

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

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

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

Copyright © 2026
Copyright for this paper is retained by authors

and

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

However, some additions are re-used. By introducing and applying the substitution variables

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

we can now compute

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

saving 3 additions. While these savings are useful for the system of "C"-expressions, note that the same techniques are applicable for the systems of "A"- and "B"-expressions as well, which allows us to divide the overall minimization problem into three independent (atomic) minimization problems.

Using the same technique, we can save 3 additions when computing all the A-expressions, and another 3 when computing the B-expressions, totaling 15 additions, which is optimal within this search space [8, 36].[4]

However, by performing a change of basis [19] manages to use 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 Atomic Minimization Problem In this work we study the performance of classical optimization methods like in Section 2.2, but for matrices larger than the 2×2 ones, to see how they scale with the problem size compared to [4]. Each of the three atomic problems above can be stated as instances of the following general minimization problem. Given

$$\begin{aligned}
s_1 &= g_{11}a_1 + \cdots g_{1q}a_q, \\
&\vdots \\
s_p &= g_{p1}a_1 + \cdots g_{pq}a_q,
\end{aligned}$$

where $g_{ij} \in \{-1, 0, 1\}$, compute $s_1, \ldots, s_p$ using as few additions and subtractions as possible. We can conveniently represent the problem on matrix form as

$$G = \begin{bmatrix} g_{11} & \cdots & g_{1q} \\ \vdots & \ddots & \vdots \\ g_{p1} & \cdots & g_{pq} \end{bmatrix}. \tag{2.2}$$

Now, the optimization process in Section 2.2 corresponds to iteratively transforming G by combining two columns in G to form a new additional column, directly corresponding to applying a variable substitution.

2.4 Our Framework To reduce the search space, we make some limitations on the types of variable substitutions we consider.

Specifically, our search space is spanned by repeated applications of so-called *saving substitutions*, which we define in Section 2.3. These substitutions only consider adding terms that are explicitly included in at least two of the sums in our minimization problem.

For a concrete example where it is possible to improve upon our work, see Appendix B. However, extending the search space to find such patterns makes the optimization problem considerably more involved. We leave performing such extended search on the actual matrix multiplication schemes studied in this paper for future work.

2.4.1 Combining Two Columns We first consider addition of two columns i and j. The corresponding case for subtraction is analogous. By convention we always let $i < j$. Let us denote the substitution 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 all such matching patterns, and let us assume that there are at least two. Introduce and apply the variable substitution $a_{q+1} := a_i + a_j$, replacing all instances of $a_i + a_j$ with a_{q+1}. We then save additions compared to naively performing the addition once for each such 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 operation as a *saving substitution*.

In terms of our matrix representation in (2.2), 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.

The case for subtraction is analogous, denoting the substitution of subtracting columns i and j by $(i, j, -1)$, and replacing all instances of $a_i - a_j$ by $a_{q+1} = a_i - a_j$. In our matrix representation in (2.2), in the new column we put value 1 for each row κ where $g_{\kappa i} = 1$ and $g_{\kappa j} = -1$, and value -1 for each row κ where $g_{\kappa i} = -1$

[4] See further details of this example in Appendix A.

Copyright © 2026
Copyright for this paper is retained by authors

and $g_{\kappa j} = 1$, and 0 for the remaining rows. Also for subtraction, we refer to a substitution that saves at least one operation as a *saving substitution*.

2.5 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. We first need a couple of definitions.

DEFINITION 2.1 (Number of Matching Patterns). *The number of matching patterns in matrix G with respect to the substitution (i, j, σ) is denoted*

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

If $M\big(G, (i, j, \sigma)\big) \geq 2$, then (i, j, σ) is a *saving substitution*, saving $M\big(G, (i, j, \sigma)\big) - 1$ operations.

DEFINITION 2.2 (Resulting Matrix). *Given a matrix G and a saving substitution (i, j, σ), let $f\big(G, (i, j, \sigma)\big)$ denote the matrix G' resulting from performing the substitution (i, j, σ) on the matrix G.*

DEFINITION 2.3 (Set of Saving Substitutions). *The set of all saving substitutions on matrix G is denoted*

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

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

DEFINITION 2.4 (Potential). *Let G be a matrix. The potential of G, denoted $P(G)$, is*

$$\sum_{(i,j,\sigma) \in S(G)} \Big(M\big(G, (i, j, \sigma)\big) - 1 \Big) \tag{2.3}$$
$$= \sum_{(i,j,\sigma) \in S(G)} M\big(G, (i, j, \sigma)\big) - |S(G)|.$$

The potential is the sum of the number of saved additions for all saving substitutions. As shown in Theorem 2.5 below, given our framework of saving substitutions – 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 a substitution tends to significantly reduce the number of matching patterns for other substitutions. 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 2.5. *By applying a saving substitution (i, j, σ) 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 substitution (i, j, σ).*

Proof. Consider the substitution (i, j, σ) applied to G and the set of saving substitutions $S(G) = \{(k, l, \sigma') | M(G, (k, l, \sigma')) \geq 2\}$[5]. 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 substitution (i, j, σ).

Performing the substitution (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 substitutions for G' are the saving substitutions for G, plus substitutions of the form (l, t, σ'), where (k, l, σ') is a saving substitution of G, $k \in \{i, j\}$ and $l \notin \{i, j\}$[6].

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 substitution (i, j, σ).

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

- For a substitution (k, l, σ'), where $k \notin \{i, j\}$ and $l \notin \{i, j\}$, the number of matching patterns is unchanged. Thus, this substitution 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.

  - If $M(G', (l, t, \sigma')) \leq 1$, then we have not created a new saving substitution involving

[5]For readability, within this proof we let k denote a column of the matrices G and G', instead of a dimension of a matrix.

[6]Or, analogously, substitutions of the form (k, t, σ'), where (k, l, σ') is a saving substitution of G, $k \notin \{i, j\}$ and $l \in \{i, j\}$.

Copyright © 2026
Copyright for this paper is retained by authors

column t. The number of additions saved by performing (k, l, σ') on G' is then upper limited by the number of additions saved by performing the same substitution 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 substitution, for each matching pattern on a row κ, the substitution 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 of (i, l, σ') in G. 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 substitution (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 substitution (i, j, σ) itself is reduced by $m - 1$ and we have a non-increased contribution corresponding to every other saving substitution, $P(G) \geq P(G') + m - 1$, as the theorem states. □

An immediate implication of Theorem 2.5 is that - limiting the search space to *saving substitutions* - the potential of a matrix is indeed an upper limit of how many additions can be saved on that matrix[7].

2.5.1 On the Remaining Saving Substitutions Typically, performing a saving substitution also lowers the number of remaining saving substitutions, but this is not always the case. The following example shows that the number of saving substitutions can even increase after performing a saving substitution. Consider performing the substitution $(0, 1, +1)$ to G below, forming G'.

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

In this case we get

$$\begin{aligned} S(G) = \{&(0, 1, +1), (0, 2, +1), (0, 2, -1), \\ &(1, 2, +1), (1, 2, -1)\}, \\ S(G') = \{&(0, 2, +1), (0, 2, -1), (1, 2, +1), \\ &(1, 2, -1), (2, 3, +1), (2, 3, -1)\}, \end{aligned}$$

increasing the number of saving substitutions by one. However, notice that the potential of G' is 6, while the potential is 15 for G. Thus, while the number of saving substitutions is higher, the potential of the new matrix is lower by at least 3 (the number of additions that the substitution $(0, 1, +1)$ saved), in accordance with Theorem 2.5.

2.6 Formulation as Longest Path The problem of finding an algorithm that saves a maximum number of additions can be represented as finding the longest path in a directed, weighted tree.

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

As the potential of the resulting matrix decreases with each saving substitution – according to Theorem 2.5 – we are guaranteed to reach a leaf after a finite number of steps[8].

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

2.7 Algorithms We have developed two types of algorithms for solving our longest path problem. One

[7]Even allowing for arbitrary substitutions, the potential is likely impossible to beat for most matrix schemes.

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

Copyright © 2026
Copyright for this paper is retained by authors

type that finds an optimal solution, but becomes too slow as the dimension grows. One that finds a good approximate solution much more quickly.

2.7.1 Exhaustive Search The most obvious approach to solving the problem is to simply search the whole tree exhaustively for the best solution. 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 for benchmarking the two greedy approaches in Section 2.7.2 on small dimensions.

We speed up this process slightly with pruning. We first find an approximate solution using the faster greedy algorithms covered in Section 2.7.2. 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 2.5.

2.7.2 Greedy Vanilla and Potential A straightforward approach for solving the problem is a greedy one where we in each step choose the substitution that saves the most number of additions. In case of a tie, we simply choose whichever such substitution 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 a substitution saves and the potential of the matrix that remains after performing the substitution. In case of a tie with respect to this metric, we simply choose the first best substitution we found. More formally, letting $G' = f\big(G, (i,j,\sigma)\big)$, we score the substitution (i,j,σ) as

$$M(G, \big(i,j,\sigma)\big) - 1 + \alpha P(G'). \tag{2.4}$$

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 substitution. 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 In Section 3.1 we apply our reduction algorithms to a large range of schemes, with and without comparisons against previous works.

- We detail how our search algorithms compare to those of [4, 14] for a range of algorithms in Section 3.1.1.
- In Section 3.1.2 we cover how our algorithms perform on two larger sets of schemes, with the schemes taken from [2, 20, 21, 26] and [12, 30] respectively. Our focus is on the performance of our algorithms on the minimization problems introduced in Section 2.3. Here we do not have reference results to compare with.
- In Section 3.1.3 we show how we can improve the arithmetic complexity of a recently introduced scheme for fast, structured matrix multiplication [37].

In Section 3.2 we study the performance of our algorithm on two large data sets of schemes in the 3×3 and the 4×4 settings, respectively.

In Section 3.3 we study the correlation between the arithmetic complexity of FMM schemes before and after reducing additions. We comment on how this can be used to optimize the generation of FMM schemes.

We briefly mention the speed of – and the potential improvements of the speed of – the implementation of our algorithms for optimizing the number of additions in Section 3.4. Finally, we illustrate the performance of the resulting optimized matrix multiplication algorithms in Sections 3.5 and 3.6.

3.1 FMM Addition Reduction Here we apply our reduction algorithms to a wide range of FMM schemes.

3.1.1 Comparison Against Previous Work The number of additions needed for our approaches compared to [4] is covered in Table 3.1 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 nonzero 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 [4] 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 [4] depends quite a lot on the precise scheme. Notice

Copyright © 2026
Copyright for this paper is retained by authors

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 there is no clear relation between the naive and the optimized number of additions. This relation is analyzed in more detail in Sections 3.2.1 and 3.3, where we study thousands of 3×3 schemes.

In [5] 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 [4], 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 [9], but also in this case the practical runtime seems to not be faster due to larger memory footprint and communication costs.

We note that Schwartz et al. claim improvements over [4] in a recent patent application [38], the practical implications of which are currently unclear.

We also have a look at other schemes where we have reference results from [4] in Table 3.2. 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 some of the schemes, running the brute-force method was not feasible. Notice the clear trend that our algorithm tends to do better compared to [4], the larger the dimension is.

In our comparion, we include all schemes from Table 2 of [4], except the three with fractional coefficients. Notice that [14] does not improve over [4] for the schemes covered in Table 3.2, hence no reference to results from that work.

Notice that for both Table 3.1 and Table 3.2, 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.

A final comparison between our work and previous works is in Table 3.3. Here we only used Greedy Potential. For each scheme we report one (possible) triple of α values leading to our optimized number of additions. The table includes most of the AlphaTensor schemes [12] from Table A.1 of [14]. We removed the schemes with non-ternary coefficients. We have also added two Deepmind schemes with no reference results at the end of the table. Recently Google Deepmind's coding agent AlphaEvolve found a couple of new schemes with record-low ranks [30]. At the end of the table we have gathered all ternary schemes from AlphaEvolve, below the final horizontal line. We confirmed for the schemes 222-7, 223-11, 224-14, 225-18, 226-21 and 233-15 that the exhaustive algorithm matches the results of Greedy Potential. Notice that also in this setting, for all comparable schemes, our methods do better compared to methods within the KS framework, the larger the matrix dimensions are.

3.1.2 Some Results for Larger Dimensions

For schemes in this subsection we fail to find comparisons against the approach of [4, 14]. Here we compile two tables of how our methods perform on two sets of algorithms from [2, 20, 26] and one table of how our methods perform over $\mathbb{Z}_2$ for the Deepmind schemes [12, 30]. 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 [2, 20, 26] and [12, 30] respectively, without optimizing the number of additions. Notice of course that none of these works attempted to minimize the number of additions.

Flip Graph Schemes. Below in Table 3.4 algorithms performed over $\mathbb{Z}$ are covered, while Table 3.5 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 substitutions[9].

Kauers and Moosbauer searched for matrix multiplication algorithms with coefficients in $\mathbb{Z}_2$, resulting in the algorithms labeled as Moosbauer $\mathbb{Z}_2$ in Table 3.5. Using these solutions they attempted to transform them into more general algorithms over $\mathbb{Z}$, resulting in the algorithms labeled Moosbauer $\mathbb{Z}$ in Table 3.4. Most of these algorithms feature non-ternary coefficients. For such schemes, in this work we only attempt to minimize the number of additions and not the number of scalar multiplications by non-ternary coefficients. We leave optimization of the combined number of additions plus scalar multiplications for future work.

To handle non-ternary coefficients, we have to slightly generalize our saving substitutions. 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 3.5 labeled as Adaptive refer to the two algorithms over $\mathbb{Z}_2$ introduced in [2]. The two algorithms labeled as Moosbauer-Poole refer to the two algorithms over $\mathbb{Z}_2$ introduced in [26]. The first block of Table 3.5 covers algorithms from Table 3.4

[9] In case of a tie by the metric in (2.4), the algorithm chooses the first such saving substitution it finds.

Copyright © 2026
Copyright for this paper is retained by authors

	Naive additions	without change of basis: Greedy Vanilla	Greedy Potential	Exhaustive	with [4]
Laderman [25]	98	70	**62**	**62**	74
Smirnov [42]	84	**68**	**68**	**68**	68
Grey-221 [6]	166	85	82	-	73
Grey-152 [6]	97	64	63	63	61
Grey-144 [6]	89	68	68	68	65
Grey-143 [6]	88	66	**64**	**64**	65
Grey-142 [6]	87	**65**	**65**	**65**	66

Table 3.1: Comparison of the number of additions for 3×3 matrix multiplication algorithms. Recall that [4] assumes the KS framework while we do not. Results where we match the results of [4] are marked in bold, as our approach does not have the cost of changing basis.

	Naive additions	without change of basis: Greedy Vanilla	Greedy Potential	Exhaustive	with [4]
Strassen-222-7 [46]	18	18	18	18	12
Winograd-222-7 [36]	24	15	15	15	12
Grey-322-11 [6]	22	21	21	21	18
Madan-232-11 [6]	22	22	22	22	18
Grey-422-14 [6]	48	37	37	37	28
HopcroftKerr-323-15 [15]	55	44	43	43	39
Grey-323-15 [6]	64	44	44	44	39
Grey-522-18 [6]	53	40	40	40	32
Grey-423-20 [42]	78	56	54	54	51
Grey-423-20 [6]	82	60	59	59	51
Grey-423-20 [6]	86	**52**	**52**	**52**	54
Grey-423-20 [6]	92	59	**58**	**58**	-
Grey-423-20 [6]	104	60	59	-	56
Grey-234-20 [6]	96	63	62	62	58
Grey-433-29 [6]	164	**99**	**98**	-	102
Grey-343-29 [6]	167	**103**	**101**	-	105
Smirnov-343-29 [43]	137	**95**	**95**	-	109
Smirnov-353-36 [43]	199	**132**	**129**	-	139

Table 3.2: Comparison of the number of additions. Recall that [4] assumes the KS framework while we do not.

that are initially defined over Z, but then reduced to $\mathbb{Z}_2$.

The Moosbauer-Poole algorithms in Table 3.4 refer to the two algorithms over $\mathbb{Z}$ introduced in [26]. Notice that for these two algorithms, when reducing the coefficients to $\mathbb{Z}_2$, we get exactly the two corresponding Moosbauer-Poole algorithms in Table 3.5.

The Moosbauer-Poole-Kemper and Kauers-Wood-Kemper algorithms in Table 3.4 refer to Kemper's reduction [23] of the Moosbauer-Poole [26] schemes and the Kauers-Wood [21] schemes respectively, to ternary coefficients.

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.

Very recently, Kauers and Wood presented a longer version [22] of their preliminary findings from [21]. There they introduced over 1,000 schemes of improved rank, spread across 29 different combinations of (n, m, k) values. Unfortunately, the majority of their schemes involve fractional coefficients. We leave studying these schemes for future work.

Also very recently, Perminov introduced a large set of ternary schemes, some with record-breaking ranks in [33, 35]. We also leave studying these schemes for future work.

Copyright © 2026
Copyright for this paper is retained by authors

	Naive additions	without change of basis: Greedy Vanilla	without change of basis: Greedy Potential	without change of basis: Greedy $\alpha_A, \alpha_B, \alpha_C$	with [14]
222-7	22	16	16	0, 0, 0	12
223-11	25	25	24	0, 0, 0	21
224-14	48	36	36	0, 0, 0	31
225-18	65	43	42	0, 0, 0	40
226-21	54	49	45	0, 0, 0	42
233-15	58	46	46	0, 0, 0	42
234-20	88	62	**60**	0, 0.01, 0	60
235-25	113	74	**73**	0.01, 0, 0	75
244-26	122	**94**	**94**	0, 0, 0	96
255-40	283	156	**154**	0.29, 0, 0	156
355-58	369	238	**222**	0.26, 0.27, 0.08	300
444-49	468	180	**171**	0.01, 0.01, 0.074	222
455-76	549	322	**310**	0.26, 0.01, 0.11	437
4510-152	1568	632	**587**	0.18, 0.01, 0.09	-
557-134	918	554	**529**	0.09, 0.26, 0.17	-
247-45	308	181	177	0, 0.26, 0	-
248-51	354	197	190	0.38, 0.01, 0.15	-
256-47	332	189	184	0.43, 0.01, 0	-
333-23	95	65	64	0.01, 0, 0	-
348-74	461	272	267	0, 0.29, 0	-
445-61	455	248	238	0, 0.01, 0.12	-
447-85	631	326	320	0.25, 0.25, 0	-
555-93	846	401	387	0.01, 0.2, 0.08	-
456-90*	771	384	366	0.2, 0.19, 0.01	-

Table 3.3: Optimized number of additions in schemes found by AlphaTensor [12] above the horizontal line and schemes found by AlphaEvolve [30] below the horizontal line. All schemes have ternary coefficients. Recall that [14] assumes the KS framework while we do not. The scheme marked with a '*' is ternary due to a modification found by Kemper [23]. The notation $4510-152$ should be read as a rank-152 scheme for dimensions $(n, m, k) = (4, 5, 10)$.

Deepmind Schemes Over $\mathbb{Z}_2$. We study the arithmetic complexity of the schemes presented by Deepmind in [12, 30], in the $\mathbb{Z}_2$ setting and present the results in Table 3.6.

The top third of the table contains schemes from the AlphaTensor paper by Deepmind [12], defined over $\mathbb{Z}_2$. The results for the AlphaTensor schemes, originally with coefficients outside $\mathbb{Z}_2$, but then reduced modulo 2 by us is found in the middle part of the table. Finally, the bottom part of the table contains algorithms from AlphaEvolve, with coefficients reduced modulo 2. For two of the schemes, our algorithm performed slightly worse here than it did for ternary coefficients, see Table 3.3. Over $\mathbb{Z}_2$, the Greedy Potential heuristic simply fails to find the substitutions that it does over ternary coefficients. However, we can of course simply copy the solution we found over ternary coefficients, replace all minus signs by plus signs and match the result over $\mathbb{Z}_2$.

3.1.3 On Structured Matrix Multiplication

In [37], a new scheme for efficiently computing XX^T is presented for $X \in \mathbb{R}^{4\times 4}$, using a rank of 26. Notice that while XX^T is a symmetric matrix, when applying the algorithm on a large matrix X, all submatrix multiplications are non-symmetric. Thus, the algorithm cannot be applied recursively.

The authors also optimize the arithmetic complexity, reducing it from the naive 139 additions to 100. Using our methods, we achieved 96 additions. By reducing the scheme to $\mathbb{Z}_2$, we managed to further reduce this to 93 additions.

We note the very recent work on fast, structured matrix multiplication from [24]. We leave studying the arithmetic complexity of this work for the future.

3.2 Studying Two Large Sets of Schemes

In this part we study two large sets of schemes, one set for multiplying 3×3 matrices in Section 3.2.1 and one set for multiplying 4×4 matrices in Section 3.2.2.

Copyright © 2026
Copyright for this paper is retained by authors

	Naive	Greedy		Improvement		
	additions	Vanilla	Potential	ops	%	$\alpha_A, \alpha_B, \alpha_C$
Moosbauer $\mathbb{Z}$						
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
Moosbauer-Poole $\mathbb{Z}$						
555-93	1,031	450	427	604	58.6	0.15, 0.15, 0.01
666-153	2,261	739	712	1,549	68.5	0.1407, 0.0962, 0.0625
Moosbauer-Poole-Kemper $\mathbb{T}$						
555-93	1,013	442	425	588	58.0	0.18, 0.14, 0.08
666-153	2,232	704	681	1551	69.5	0.1, 0.11, 0.066
Kauers-Wood-Kemper $\mathbb{T}$						
456-90	1,082	418	404	678	62.7	0.14 , 0.02 , 0.08
466-106	1,292	478	464	828	64.1	0.16 , 0.08 , 0.02
556-110	1,325	511	492	833	62.9	0.18 , 0.12 , 0.08
557-127	1,518	577	566	952	62.7	0.0 , 0.1 , 0.08
566-130	1,666	625	603	1063	63.8	0.1 , 0.12 , 0.06
567-150	2,132	720	685	1447	67.8	0.14, 0.14, 0.04

Table 3.4: Algorithms over $\mathbb{Z}$. Comparison of how many additions schemes from [2, 20, 21, 26] need naively vs. after our optimizations. The two last blocks of schemes have ternary coefficients due to a reduction by Kemper [23].

3.2.1 Studying a Large Set of 3×3 Schemes In [13], Heule et al. present a set of more than 17,000 schemes for multiplying 3×3 matrices using 23 scalar multiplications. All schemes have ternary coefficients. We apply the Greedy Potential algorithm to all of these schemes, both in the regular metric and over $\mathbb{Z}_2$. From this set we found three schemes using only 61 additions, improving our previous result from [28].

Recently, Stapleton applied deep learning-based methods to discover a large set of 3×3 schemes with

Copyright © 2026
Copyright for this paper is retained by authors

	Naive additions	Greedy		Improvement		$\alpha_A, \alpha_B, \alpha_C$
		Vanilla	Potential	ops	%	
Moosbauer $\mathbb{Z} \xrightarrow{Red.} \mathbb{Z}_2$						
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
245-33	289	135	132	157	54.3	0.26, 0.02 , 0
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 $\mathbb{Z}_2$						
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
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
Moosbauer-Poole $\mathbb{Z}_2$						
555-93	1,013	438	417	596	58.5	0.16, 0.16, 0.02
666-153	2,232	712	679	1,553	69.6	0.09, 0.09, 0.062
Adaptive $\mathbb{Z}_2$						
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
Kauers-Wood $\mathbb{Z} \xrightarrow{Red.} \mathbb{Z}_2$						
456-90	1,082	421	408	674	62.3	0.14, 0.08, 0.02
456-90	1,049	406	398	651	62.1	0, 0.02, 0.04
457-104	1,340	477	465	875	65.3	0.1, 0.06, 0.02
466-106	1,292	474	455	837	64.8	0.16, 0.12, 0.06
467-123	1,754	595	575	1179	67.2	0.0, 0.1, 0.02
477-144	1,951	714	648	1303	66.8	0.1, 0.12, 0.02
477-144	2,106	683	680	1426	67.7	0.02, 0.1, 0.02
556-110	1,333	511	495	838	62.9	0.1, 0.12, 0.08
556-110	1,325	511	496	829	62.6	0.16, 0.16, 0.08
557-127	1,518	584	560	958	63.1	0, 0.18, 0.06
557-127	1,554	584	568	986	63.4	0.02, 0.08, 0.06
557-127	1,590	570	563	1027	64.6	0.02, 0.16, 0.06
557-127	1,488	585	553	935	62.8	0.14, 0.14, 0.08
557-127	1,600	562	553	1047	65.4	0.02, 0.18, 0.02
557-127	1,589	571	559	1030	64.8	0, 0.16, 0.06
557-128	1,539	583	566	973	63.2	0.14, 0.14, 0.06
566-130	1,666	616	591	1075	64.5	0.14, 0.14, 0.06
567-150	2,100	706	684	1416	67.4	0.02, 0.14, 0.04
567-150	2,132	721	688	1444	67.7	0.14, 0.14, 0.04
577-176	2,663	855	817	1846	69.3	0.06, 0.06, 0.04
577-176	2,677	863	826	1851	69.1	0.12, 0.1, 0.02
667-183	2,807	876	845	1962	69.9	0.11, 0.11, 0.05
667-183	2,848	888	850	1998	70.2	0.11, 0.07, 0.03

Table 3.5: Algorithms over $\mathbb{Z}_2$. Comparison of how many additions are needed naively vs. after our optimizations, with scheme taken from [2, 20, 21, 26].

Copyright © 2026
Copyright for this paper is retained by authors

	Naive	Greedy		Improvement		
	additions	Vanilla	Potential	ops	%	$\alpha_A, \alpha_B, \alpha_C$
AlphaTensor $\mathbb{Z}_2$						
222-7	24	15	15	9	37.5	0, 0, 0
223-11	31	22	22	9	29.0	0, 0, 0
224-14	69	36	36	33	47.8	0, 0, 0
225-18	71	37	37	34	47.9	0, 0, 0
233-15	71	45	44	27	38.0	0.01, 0, 0
234-20	90	57	57	33	36.7	0, 0, 0
235-25	118	73	72	46	39.0	0.01, 0, 0
244-26	149	89	87	62	41.6	0.28, 0, 0
245-33	190	109	107	83	43.7	0, 0.22, 0.13
255-40	256	148	144	112	43.8	0.28, 0, 0.01
333-23	112	68	67	45	40.2	0, 0.01, 0
334-29	141	98	96	45	31.9	0.01, 0, 0
335-36	230	127	122	108	47.0	0.2, 0, 0
344-38	216	130	124	92	42.6	0.25, 0, 0.08
345-47	287	171	161	126	43.9	0.23, 0.01, 0.01
355-58	413	232	220	193	46.7	0.21, 0.23, 0.06
444-47	340	210	199	141	41.5	0.01, 0.01, 0.15
445-63	425	249	238	187	44.0	0.01, 0.18, 0.15
455-76	549	325	310	239	43.5	0.19, 0.1, 0.1
555-96	745	404	310	435	58.4	0.19, 0.1, 0.1
AlphaTensor $\mathbb{Z} \xrightarrow{Red.} \mathbb{Z}_2$						
222-7	22	16	16	6	27.3	0, 0, 0
223-11	25	24	24	1	4.0	0, 0, 0
224-14	48	36	36	12	25.0	0, 0, 0
225-18	65	42	42	23	35.4	0, 0, 0
226-21	54	45	45	9	16.7	0, 0, 0
227-25	65	51	51	14	21.5	0, 0, 0
228-28	98	61	61	37	37.8	0, 0, 0
233-15	58	46	46	12	20.7	0, 0, 0
234-20	88	61	60	28	31.8	0, 0.01, 0
235-25	113	74	73	40	35.4	0.01, 0, 0
244-26	122	94	94	28	23.0	0, 0, 0
245-33	200	114	113	87	43.5	0.3, 0, 0
255-40	283	155	155	128	45.2	0, 0, 0
333-23	102	67	67	35	34.3	0, 0, 0
334-29	145	96	96	49	33.8	0, 0, 0
335-36	178	113	111	67	37.6	0, 0.01, 0
344-38	201	143	135	66	32.8	0.01, 0.34, 0.09
345-47	288	176	165	123	42.7	0.21, 0.01, 0.08
355-58	369	233	222	147	39.8	0.26, 0.27, 0.08
359-105	660	384	370	290	43.9	0.09, 0.27, 0.08
444-49	468	181	170	298	63.7	0.17, 0.17, 0.1
445-63	472	249	238	234	49.6	0.16, 0.14, 0.12
455-76	549	325	310	239	43.5	0.19, 0.1, 0.1
459-139	1025	560	544	481	46.9	0.18, 0.14, 0.1
4510-152	1568	633	587	981	62.6	0.18, 0.07, 0.07
4511-169	1221	629	620	601	49.2	0.21, 0.19, 0.08
555-98	638	367	354	284	44.5	0.34, 0.23, 0.08
557-134	918	539	518	400	43.6	0.17, 0.25, 0.09
579-234	2586	994	969	1617	62.5	0.15, 0.14, 0.01
5710-257	3013	1089	1064	1949	64.7	0.14, 0.01, 0.04
AlphaEvolve $\xrightarrow{Red.} \mathbb{Z}_2$						
247-45	308	180	176	132	42.9	0, 0.21, 0
248-51	354	190	183	171	48.3	0.34, 0.1, 0.15
256-47	332	188	184	148	44.6	0.31, 0.01, 0
333-23	95	65	64	31	32.6	0.01, 0, 0
348-74	461	272	*268	193	41.9	0, 0.17, 0
356-68	480	269	261	219	45.6	0.01, 0.01, 0.09
357-80	650	354	347	303	46.6	0, 0.14, 0.1
445-61	455	248	238	217	47.7	0, 0.01, 0.1
447-85	631	325	*322	309	49.0	0.21, 0.01, 0
456-90	771	375	358	413	53.6	0.19, 0.17, 0.01
555-93	846	399	384	462	54.6	0.13, 0.18, 0.11

Table 3.6: Results from applying Greedy Potential to a set of algorithms found by Deepmind [12, 30], all over $\mathbb{Z}_2$. The top part consists of algorithm presented over $\mathbb{Z}_2$ in [12]. The middle part consists of schemes presented over various subsets of $\mathbb{Z}$ in [12], but reduced to $\mathbb{Z}_2$ by us. The bottom part consists of schemes presented over various subsets of $\mathbb{Z}$ in [30], but reduced to $\mathbb{Z}_2$ by us. The asterisks annotating two schemes mark that our Greedy algorithm performed worse than it did for the same scheme over ternary coefficients, see Table 3.3. Here, 4510, 4511, 5710 are to be read as $(n, m, k) = (4, 5, 10)$, $(n, m, k) = (4, 5, 11)$ and $(n, m, k) = (5, 7, 10)$, respectively.

Copyright © 2026
Copyright for this paper is retained by authors

23 scalar multiplications [45]. Using the Greedy Potential algorithm he found a couple of schemes with an arithmetic complexity of 60. By further applying this methodology, Mårtensson, Stankovski and Stapleton reported a scheme with 59 additions in [29]. Finally, Perminov found a rank-23 scheme with only 58 additions by searching starting from a scheme with rank 23 and then searching its neighborhood within the flip graph [32].

These improved results compared to what we found by studying the schemes found be Heule et al. highlight that to find a solution with low arithmetic complexity, it is not enough to just generate many schemes. The procedure with which the schemes are generated also matters. We leave the details of how to achieve this more generally for future work. Note that the work by Perminov starts from a single scheme and searches a local neighborhood, while the others methods we have looked at generate schemes independently.

In Figure 3.2 we show the distribution of the optimized number of additions, for the schemes from [13]. We also fit a normal distribution to better understand the behavior. We use a logarithmic scale on the y axis to better see the tail ends of the distribution. We note that the normal distribution is a good approximation, but that it overestimates probabilities for the tail ends of the distribution. Importantly for us, this means that (if we could keep sampling schemes from the same distribution) sampling a scheme with 60 additions is less likely than the model suggests. We do not attempt to correct for this to estimate the probability of sampling such a scheme.

Figure 3.2 is in the regular metric. In the binary metric, the mean number of additions is slightly lower. However, we still do not find any scheme with 60 additions among the Heule et al. schemes in the binary metric either.

3.2.2 Studying a Large Set of 4×4 Schemes

Deepmind provided more than 14,000 schemes of rank 49 for multiplying 4×4 matrices [12]. We apply the Greedy Potential algorithm to all of these schemes. We plot a frequency distribution for the optimized number of additions in Figure 3.1. We do not attempt to make a statistical model for the distributions. Both distributions are similar to a normal distributions. However, for both settings, the distributions look roughly like a normal distribution in logarithmic scale, and thus have much steeper peaks than a normal distribution. For both settings, the bulk of the distribution is on the left side, but both have a lot of outliers requiring many more additions. Trying to model the behavior in more detail is futile, as we do not have access to the details of how Deepmind generated and selected these schemes.

The best scheme requires 162 additions in the standard metric. In the binary metric, the best scheme requires 161 additions. Notice the difference here, unlike in the 3×3 case of Section Section 3.2.1. This is in line with our observations in Sections 3.1.1 and 3.1.2. For both the flip graph schemes and the Deepmind schemes, the difference in performance between the $\mathbb{Z}$ setting and the $\mathbb{Z}_2$ increases with the dimension.

The arithmetic complexity can naturally be compared with that of performing two levels of Strassen's algorithm, using Winograd's form. At the second recursive level, $7 \cdot 15 = 105$ additions are then performed. At the first recursive level, 15 additions are performed, all of which involve four times more elements than the additions at the second level. Thus, the total arithmetic complexity is then $105 + 4 \cdot 15 = 165$.

To conclude the 3×3 and 4×4 settings, in order to achieve a scheme with a very low arithmetic complexity, a lot of schemes need to be generated. The method with which the schemes are generated also matters, of course.

3.3 How to Optimize FMM Generation

Recall the FMM generation pipeline in Figure 1.1. It is not immediately evident if *minimization* or *maximization* of additions in FMM generation performs better in terms of *global* minimization of additions over the entire FMM pipeline. Fewer additions give a lower addition count at the start of the reduction process, while a higher addition count provides much more alternatives in which the reduction process can find reduction benefits.

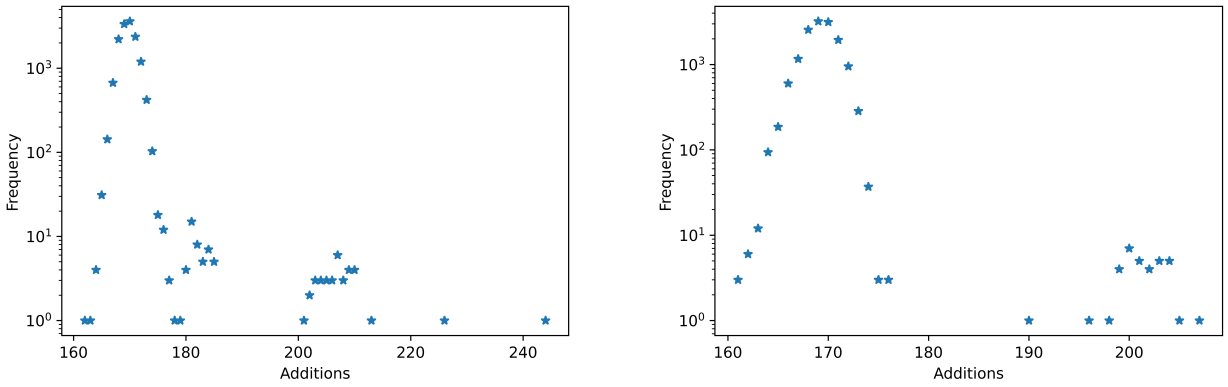
In Figure 3.3 we illustrate the connection between the arithmetic complexity of the Heuele et al. schemes of Section 3.2.1 before and after optimizing the number of additions. We particularly note the tendency that schemes of lower naive arithmetic complexity tend to also have lower optimized arithmetic complexity. Although the data plotted in Figure 3.3 is specific to the FMM generation used for the schemes of Heule et al., and a deeper investigation of the general behavior of various classes of generation algorithms is indeed warranted, we believe that it is likely that other FMM generation methods will exhibit the same behavior.

We conclude that it is advantageous to bias the FMM generation to output schemes of low (naive) arithmetic complexity, making it more likely to find a scheme of very low optimized arithmetic complexity. As for how to concretely minimize additions during FMM generation, we leave that for future work.

3.4 On the Implementation Runtime

Running our single-threaded, non-optimized Python implementation [27] with parameters according to Table 3.5 for the $(n, m, k) = (6, 6, 6)$ scheme from [20] takes takes around 1,000 seconds on a basic laptop. The runtime is significantly lower for the smaller problem sizes. Run-

Copyright © 2026
Copyright for this paper is retained by authors



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

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

Figure 3.1: Frequency distribution for the optimized number of additions needed in the Deepmind set of schemes for multiplying 4×4 matrices.

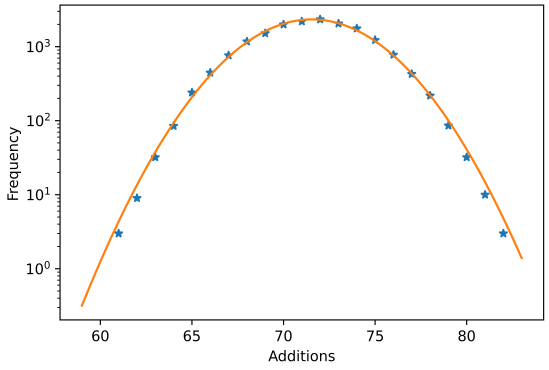


Figure 3.2: A frequency distribution for the optimized number of additions needed in the Heuele et al. set of schemes for multiplying 3×3 matrices [13]. The parabola is a normal distribution approximation of the frequencies.

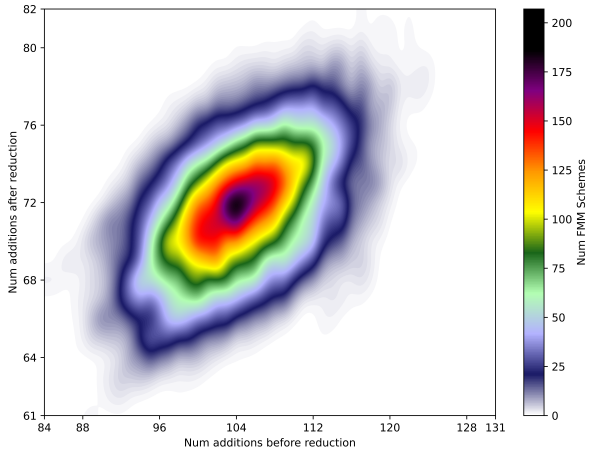


Figure 3.3: Addition reduction heatmap, showing how arithmetic complexities of FMM schemes before and after addition reduction relate.

ning our corresponding single-threaded, non-optimized C implementation [44] takes about 42 seconds. Parallelized and optimized implementations may be orders of magnitude faster.

3.5 Illustrating the Algorithm Performance

In Figure 3.4 we plot the performances of the algorithms from Tables 3.3, 3.4, 3.5 and 3.6. Each data point corresponds to one algorithm from [2, 20, 26] or [12, 30]. For an (n, m, k) algorithm the x-value is equal to $nk(m-1)$, which corresponds to the number of additions the naive algorithm uses. The y-values correspond to the number of additions used in naive and Greedy Potential-optimized implementations, respectively.

Figure 3.4a covers algorithms from Table 3.4, Figure 3.4b covers algorithms from Table 3.5, Figure 3.4c covers algorithms from Table 3.3 and Figure 3.4d covers algorithms from Table 3.6.

We notice in all cases that there is roughly a linear relation between the number of additions for the naive algorithm and the corresponding number of additions for the optimized 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. In both metrics,

Copyright © 2026
Copyright for this paper is retained by authors

both before and especially after optimization, the linear model is a better approximation for the Deepmind schemes than the flip graph schemes.

The slopes before/after optimization are 3.88/12.3 for Figure 3.4a, 3.83/12.5 for Figure 3.4b, 3.74/8.49 for Figure 3.4c and 3.59/9.16 for Figure 3.4d. Notice for both Figure 3.4a and Figure 3.4b that compared to the conference version of the paper [28], the slopes are less steep both before and after optimization. For Figure 3.4a, this is due adding the schemes of Moosbauer-Poole [26], Kauers-Wood [21] and modifications of these by Kemper [23]. For Figure 3.4b, this is due to adding the schemes of Moosbauer-Poole [26] and Kauers-Wood [21]. Furthermore, note for Figure 3.4a that all these added schemes have only ternary coefficients. Thus, the figure understates the reduction in arithmetic complexity that these schemes achieve.

Just like the recent trend of finding more efficient algorithms in terms of scalar multiplications in $\mathbb{Z}_2$, notice for both the flip graph schemes and the Deepmind schemes that the slopes are less steep over $\mathbb{Z}_2$ than over $\mathbb{Z}$. These two improvements over algorithms over $\mathbb{Z}$ are synergetic.

3.6 Illustrating Performance Differences Table 3.7 specifies the original Laderman algorithm for performing a multiplication of two 3×3 matrices using 23 multiplications and 98 additions. Table 3.8 shows how Greedy Vanilla reduces this addition count to 70. Finally, Table 3.9 shows how Greedy Potential reduces the addition count even further to 62. In Tables 3.7, 3.8 and 3.9 we use the compact indexing notation

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

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 the general trend that the classical method performs better when the dimensions of the matrices are higher.

For two different dimensions, we have studied two large sets of schemes with the same rank. In both settings we note that the optimized number of additions roughly follows a normal distribution, implying that we need to generate a large number of solutions in order to find one with a truly low arithmetic complexity.

To facilitate further research into arithmetic complexity, we provide a number of potential research directions. Some of these ideas have been scattered throughout the paper. We also list them here for convenience.

Applying the Method to More/Larger Instances. One obvious direction is simply to benchmark our methods by applying them to more/larger instances. Two obvious sets to consider are the schemes in Sedoglavic's table [41] and the recent works by Perminov [33, 34, 35]. Doing this might require optimizing the implementation and/or running it using more powerful resources.

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

Extended Optimization Framework. While it is possible to construct scheme structures for which our methods do not find the optimal solution (see Section 2.4 and Appendix B), a question is whether a corresponding extended framework is practically relevant to matrix multiplication. Within such an extended framework, it would be interesting to see if it is possible to similarly extend Theorem 2.5 to guide optimization.

Non-Ternary Coefficients. For schemes with non-ternary coefficients, our current approach aims to minimize the number of additions but ignores the costs of scalar multiplications. Many schemes of record-low rank have non-ternary coefficients, so studying the practical impact of these scalar multiplications and correspondingly realistic metrics is one path forward.

A More Relevant Metric Comparing Schemes. Currently fast matrix multiplication schemes are first compared in terms of rank and secondly in terms of arithmetic complexity as a tie-breaker. It would be interesting to develop a more relevant metric for practical comparison of schemes.

Better Heuristics for Reduction. In this work we introduce an efficient greedy approach and a slow exhaustive one. It would be interesting to find heuristic methods that outperform our greedy approach, while being faster than our exhaustive method. Perminov's heuristic in [34] is a first step in this direction.

Structured Matrices. In Section 3.1.3 we briefly mentioned the arithmetic complexity of performing a structured matrix multiplication. It is interesting to study the corresponding situation for all constructions in [24].

Extending the Sedoglavic Table. Sedoglavic' table for minimimum rank of matrix multiplication schemes currently does not include optimized arithmetic complexities. Extending it to include arithmetic complexities too would be helpful. This would allow us to make a plot, like the ones in Figure 3.4, which better captures the underlying arithmetic complexity of matrix multiplication as a function of the dimensions.

Building larger schemes by combining smaller

Copyright © 2026
Copyright for this paper is retained by authors

$$
\begin{aligned}
M_0 &= (A_0 + A_1 + A_2 - A_3 - A_4 - A_7 - A_8)B_4 \\
M_1 &= (A_0 - A_3)(-B_1 + B_4) \\
M_2 &= A_4(-B_0 + B_1 + B_3 - B_4 - B_5 - B_6 + B_8) \\
M_3 &= (-A_0 + A_3 + A_4)(B_0 - B_1 + B_4) \\
M_4 &= (A_3 + A_4)(-B_0 + B_1) \\
M_5 &= A_0B_0 \\
M_6 &= (-A_0 + A_6 + A_7)(B_0 - B_2 + B_5) \\
M_7 &= (-A_0 + A_6)(B_2 - B_5) \\
M_8 &= (A_6 + A_7)(-B_0 + B_2) \\
M_9 &= (A_0 + A_1 + A_2 - A_4 - A_5 - A_6 - A_7)B_5 \\
M_{10} &= A_7(-B_0 + B_2 + B_3 - B_4 - B_5 - B_6 + B_7) \\
M_{11} &= (-A_2 + A_7 + A_8)(B_4 + B_6 - B_7) \\
M_{12} &= (A_2 - A_8)(B_4 - B_7) \\
M_{13} &= A_2B_6 \\
M_{14} &= (A_7 + A_8)(-B_6 + B_7) \\
M_{15} &= (-A_2 + A_4 + A_5)(B_5 + B_6 - B_8) \\
M_{16} &= (A_2 - A_5)(B_5 - B_8) \\
M_{17} &= (A_4 + A_5)(-B_6 + B_8) \\
M_{18} &= A_1B_3 \\
M_{19} &= A_5B_7 \\
M_{20} &= A_3B_2 \\
M_{21} &= A_6B_1 \\
M_{22} &= A_8B_8
\end{aligned}
$$

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

Table 3.7: The original Laderman multiplication scheme with 98 additions. Here we use compact indexing notation.

$$
\begin{aligned}
t_0 &= A_0 - A_3 \\
t_1 &= A_0 - A_6 \\
t_2 &= A_2 - A_4 \\
t_3 &= A_7 + A_8 \\
t_4 &= A_1 + t_2 \\
t_5 &= A_7 - t_1 \\
u_0 &= B_0 - B_1 \\
u_1 &= B_0 - B_2 \\
u_2 &= B_4 + B_6 \\
u_3 &= B_5 - B_8 \\
u_4 &= B_3 - u_2 \\
u_5 &= B_5 + u_1 \\
M_0 &= (t_0 - t_3 + t_4)B_4 \\
M_1 &= t_0(-B_1 + B_4) \\
M_2 &= A_4(-u_0 - u_3 + u_4) \\
M_3 &= (A_4 - t_0)(B_4 + u_0) \\
M_4 &= (A_3 + A_4)(-u_0) \\
M_5 &= A_0B_0 \\
M_6 &= t_5u_5 \\
M_7 &= -t_1(B_2 - B_5) \\
M_8 &= (A_6 + A_7)(-u_1) \\
M_9 &= (-A_5 + t_4 - t_5)B_5 \\
M_{10} &= A_7(B_7 + u_4 - u_5) \\
M_{11} &= (-A_2 + t_3)(-B_7 + u_2) \\
M_{12} &= (A_2 - A_8)(B_4 - B_7) \\
M_{13} &= A_2B_6 \\
M_{14} &= t_3(-B_6 + B_7) \\
M_{15} &= (A_5 - t_2)(B_6 + u_3) \\
M_{16} &= (A_2 - A_5)u_3 \\
M_{17} &= (A_4 + A_5)(-B_6 + B_8) \\
M_{18} &= A_1B_3 \\
M_{19} &= A_5B_7 \\
M_{20} &= A_3B_2 \\
M_{21} &= A_6B_1 \\
M_{22} &= A_8B_8
\end{aligned}
$$

$$
\begin{aligned}
v_0 &= M_5 + M_{13} \\
v_1 &= M_1 + M_3 \\
v_2 &= M_6 + M_7 \\
v_3 &= M_{11} + M_{12} \\
v_4 &= M_{15} + M_{16} \\
C_0 &= M_{18} + v_0 \\
C_1 &= M_0 + M_3 + M_4 + M_{11} + M_{14} + v_0 \\
C_2 &= M_6 + M_8 + M_9 + M_{15} + M_{17} + v_0 \\
C_3 &= M_2 + v_0 + v_1 + v_4 \\
C_4 &= M_4 + M_5 + M_{19} + v_1 \\
C_5 &= M_{13} + M_{17} + M_{20} + v_4 \\
C_6 &= M_{10} + v_0 + v_2 + v_3 \\
C_7 &= M_{13} + M_{14} + M_{21} + v_3 \\
C_8 &= M_5 + M_8 + M_{22} + v_2
\end{aligned}
$$

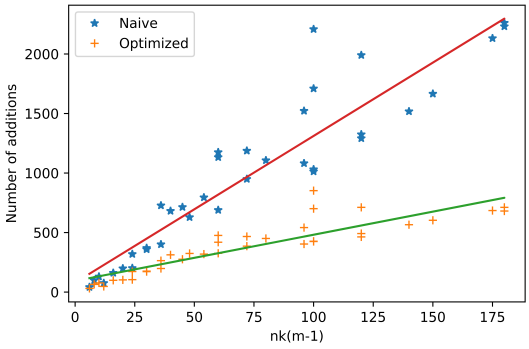
Table 3.8: Laderman multiplication scheme reduced from 98 to 70 additions with Greedy Vanilla. Here we use compact indexing notation.

Copyright © 2026
Copyright for this paper is retained by authors

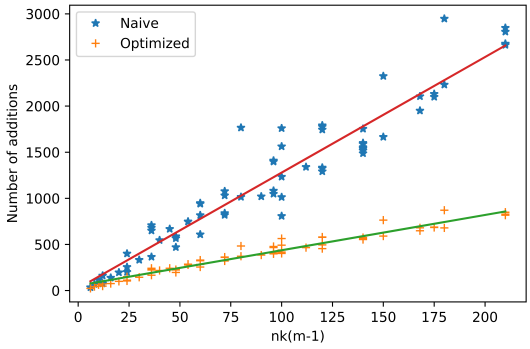
$$
\begin{array}{lll}
t_0 = A_0 - A_3 & M_2 = A_4(B_3 - u_4 - u_7) & M_{22} = A_8 B_8 \\
t_1 = A_0 - A_6 & M_3 = t_4 u_4 & \\
t_2 = A_2 - A_5 & M_4 = (A_3 + A_4)(-u_0) & v_0 = M_3 + M_5 \\
t_3 = A_2 - A_8 & M_5 = A_0 B_0 & v_1 = M_5 + M_6 \\
t_4 = A_4 - t_0 & M_6 = t_6 u_5 & v_2 = M_{11} + M_{13} \\
t_5 = A_4 - t_2 & M_7 = -t_1(B_2 - B_5) & v_3 = M_{13} + M_{15} \\
t_6 = A_7 - t_1 & M_8 = (A_6 + A_7)(-u_1) & v_4 = M_1 + v_0 \\
t_7 = A_7 - t_3 & M_9 = (A_1 - t_5 - t_6)B_5 & v_5 = M_7 + v_1 \\
 & M_{10} = A_7(B_3 - u_5 - u_6) & v_6 = M_{12} + v_2 \\
u_0 = B_0 - B_1 & M_{11} = t_7 u_6 & v_7 = M_{16} + v_3 \\
u_1 = B_0 - B_2 & M_{12} = t_3 u_2 & \\
u_2 = B_4 - B_7 & M_{13} = A_2 B_6 & C_0 = M_5 + M_{13} + M_{18} \\
u_3 = B_5 - B_8 & M_{14} = (A_7 + A_8)(-B_6 + B_7) & C_1 = M_0 + M_4 + M_{14} + v_0 + v_2 \\
u_4 = B_4 + u_0 & M_{15} = t_5 u_7 & C_2 = M_8 + M_9 + M_{17} + v_1 + v_3 \\
u_5 = B_5 + u_1 & M_{16} = t_2 u_3 & C_3 = M_2 + v_4 + v_7 \\
u_6 = B_6 + u_2 & M_{17} = (A_4 + A_5)(-B_6 + B_8) & C_4 = M_4 + M_{19} + v_4 \\
u_7 = B_6 + u_3 & M_{18} = A_1 B_3 & C_5 = M_{17} + M_{20} + v_7 \\
 & M_{19} = A_5 B_7 & C_6 = M_{10} + v_5 + v_6 \\
M_0 = (A_1 - t_4 - t_7)B_4 & M_{20} = A_3 B_2 & C_7 = M_{14} + M_{21} + v_6 \\
M_1 = t_0(-B_1 + B_4) & M_{21} = A_6 B_1 & C_8 = M_8 + M_{22} + v_5
\end{array}
$$

Table 3.9: 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 compact indexing notation.

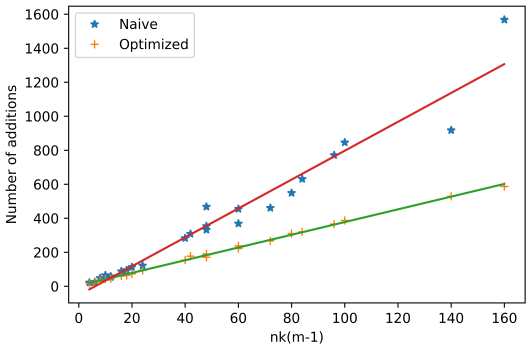
Copyright © 2026
Copyright for this paper is retained by authors



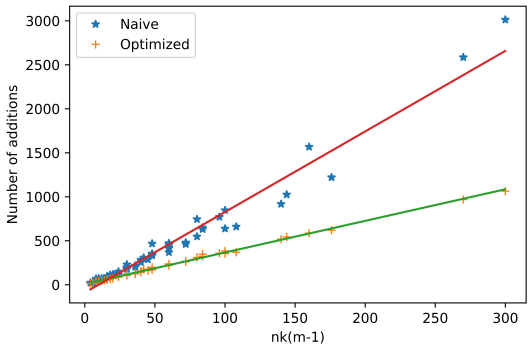
(a) Flip graph schemes with arithmetic over $\mathbb{Z}$.



(b) Flip graph schemes with arithmetic over $\mathbb{Z}_2$.



(c) Deepmind schemes with arithmetic over $\mathbb{Z}$.



(d) Deepmind schemes with arithmetic over $\mathbb{Z}_2$.

Figure 3.4: The number of additions for Strassen-type algorithms, as a function of the number of additions for a naive algorithm with the same dimensions.

schemes is used extensively in the Sedoglavic table. The performance of the arithmetic complexity of such composite constructions is currently unexplored.

Systematic Method for Finding Better Schemes. Finding an approach to generating a large pool of candidate solutions for a specific rank would be helpful in locating a scheme of particularly low arithmetic complexity. Perminov's work [34] is a step in this direction, where he does local search in the flip graph. For 3×3 matrices, a method for generating many solutions is studied in [29, 45]. One possible combined method is to generate good starting candidates as in [29, 45] and then search the flip graph locally around those candidates as in [34] to find even better schemes.

For such a hybrid approach, it is also useful to generate schemes with lower naive arithmetic complexity, as such schemes tend to have lower optimized arithmetic complexity (see Section 3.3).

Applications in Post-Quantum Cryptography. There are some examples applying FMM algorithms in post-quantum cryptography. Improving batch encryption and batch signature generation using Strassen's algorithm were studied in [7] and [47], respectively. This connection deserves further study.

Many matrix multiplications in post-quantum cryptography use matrices with various structures for efficiency reasons. Applying structured FMM algorithms (similar to the one in Section 3.1.3) can be considered.

Relation to Other Research Areas. The problem of minimizing the number of additions in linear systems 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] J. Alman, R. Duan, V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, *More asymmetry yields faster matrix multiplication*, in Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 2025, pp. 2005–2039.

Copyright © 2026
Copyright for this paper is retained by authors

[2] Y. Arai, Y. Ichikawa, and K. Hukushima, *Adaptive Flip Graph Algorithm for Matrix Multiplication*, in Proceedings of the 2024 International Symposium on Symbolic and Algebraic Computation, ISSAC '24, New York, NY, USA, 2024, Association for Computing Machinery, p. 292–298.

[3] G. V. Bard, *Algebraic Cryptanalysis*, Springer Publishing Company, Incorporated, 1st ed., 2009.

[4] G. Beniamini, N. Cheng, O. Holtz, E. Karstadt, and O. Schwartz, *Sparsifying the Operators of Fast Matrix Multiplication Algorithms*. arXiv:2008.03759 [cs.DS], 2020.

[5] G. Beniamini and O. Schwartz, *Faster Matrix Multiplication via Sparse Decomposition*, in The 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19'), New York, NY, USA, 2019, Association for Computing Machinery, p. 11–22.

[6] A. R. Benson and G. Ballard, *A framework for practical parallel fast matrix multiplication*, ACM SIGPLAN Notices, 50 (2015), p. 42–53.

[7] J. W. Bos, M. Ofner, J. Renes, T. Schneider, and C. van Vredendaal, *The matrix reloaded: Multiplication strategies in frodokem*, in Cryptology and Network Security, M. Conti, M. Stevens, and S. Krenn, eds., Cham, 2021, Springer International Publishing, pp. 72–91.

[8] N. H. Bshouty, *On the additive complexity of 2 × 2 matrix multiplication*, Information Processing Letters, 56 (1995), pp. 329–335.

[9] M. Cenk and M. A. Hasan, *On the arithmetic complexity of Strassen-like matrix multiplications*, Journal of Symbolic Computation, 80 (2017), pp. 484–501.

[10] J.-G. Dumas, C. Pernet, and A. Sedoglavic, *A non-commutative algorithm for multiplying 4x4 matrices using 48 non-complex multiplications*. arXiv:2506.13242 [cs.DS], 2025.

[11] ——, *A more accurate rational non-commutative algorithm for multiplying 4x4 matrices using 48 multiplications*. arXiv:2603.18699 [cs.DS], 2026.

[12] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli, *Discovering faster matrix multiplication algorithms with reinforcement learning*, Nature, 610 (2022), pp. 47–53.

[13] M. J. H. Heule, M. Kauers, and M. Seidl, *Local search for fast matrix multiplication*, in Theory and Applications of Satisfiability Testing – SAT 2019, M. Janota and I. Lynce, eds., Cham, 2019, Springer International Publishing, pp. 155–163.

[14] O. Holtz, A. Hsu, Y. Moran, O. Schwartz, and G. Wiernik, *Alternative bases for new fast matrix multiplication algorithms*, in 2025 Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA), SIAM, 2025, pp. 322–334.

[15] J. E. Hopcroft and L. R. Kerr, *On Minimizing the Number of Multiplications Necessary for Matrix Multiplication*, SIAM Journal on Applied Mathematics, 20 (1971), p. 30–36.

[16] J. Huang, T. M. Smith, G. M. Henry, and R. A. Van De Geijn, *Strassen's Algorithm Reloaded*, in SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 690–701.

[17] A. Joux, *Algorithmic Cryptanalysis*, Chapman & Hall/CRC, 1st ed., 2009.

[18] I. Kaporin, *Finding complex-valued solutions of brent equations using nonlinear least squares*, Comput. Math. and Math. Phys., (2024), pp. 1881–1891.

[19] E. Karstadt and O. Schwartz, *Matrix Multiplication, a Little Faster*, J. ACM, 67 (2020).

[20] M. Kauers and J. Moosbauer, *Flip Graphs for Matrix Multiplication*, in Proceedings of the 2023 International Symposium on Symbolic and Algebraic Computation (ISSAC '23), New York, NY, USA, 2023, Association for Computing Machinery, p. 381–388.

[21] M. Kauers and I. Wood, *Consequences of the moosbauer-poole algorithms*. arXiv:2505.05896 [cs.SC], 2025.

[22] ——, *Exploring the meta flip graph for matrix multiplication*. arXiv:2510.19787 [cs.SC], 2025.

[23] A. Kemper, *Matrix multiplication*. https://github.com/a1880/matrix-multiplication, 2025.

[24] K. Khoruzhii, P. Gelss, and S. Pokutta, *Faster algorithms for structured matrix multiplication via flip graph search*. arXiv:2511.10786 [cs.CS], 2025.

Copyright © 2026
Copyright for this paper is retained by authors

[25] J. D. Laderman, *A Noncommutative Algorithm for Multiplying* 3×3 *Matrices Using 23 Multiplications*, Bulletin of the American Mathematical Society, 82 (1976), pp. 126 – 128.

[26] J. Moosbauer and M. Poole, *Flip Graphs with Symmetry and New Matrix Multiplication Schemes*. arXiv:2502.04514 [cs.SC], 2025.

[27] E. Mårtensson, *GitHub – Python code repository fmm_add_reduction*. Available at github.com/ErikMaartensson/fmm_add_reduction, last accessed on 2026–04–23, 2024.

[28] E. Mårtensson and P. S. Wagner, *The number of the beast: Reducing additions in fast matrix multiplication algorithms for dimensions up to 666*. Cryptology ePrint Archive, Paper 2024/2063, 2024.

[29] E. Mårtensson, P. S. Wagner, and J. Stapleton, *A rank 23 algorithm for multiplying 3 x 3 matrices with an arithmetic complexity of 59*. arXiv:2601.05272 [cs.SC], 2025.

[30] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, *AlphaEvolve: A coding agent for scientific and algorithmic discovery*, 2025.

[31] V. Y. 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), 1978, pp. 166–176.

[32] A. I. Perminov, *A 58-addition, rank-23 scheme for general 3x3 matrix multiplication*. arXiv:2512.21980 [cs.DS], 2025.

[33] ——, *Fast matrix multiplication via ternary meta flip graphs*. arXiv:2511.20317 [cs.SC], 2025.

[34] ——, *Parallel heuristic exploration for additive complexity reduction in fast matrix multiplication*. arXiv:2512.13365 [cs.SC], 2025.

[35] ——, *Fast matrix multiplication in small formats: Discovering new schemes with an open-source flip graph framework*. arXiv:2603.02398 [cs.SC], 2026.

[36] R. L. Probert, *On the Additive Complexity of Matrix Multiplication*, SIAM Journal on Computing, 5 (1976), pp. 187–203.

[37] D. Rybin, Y. Zhang, and Z.-Q. Luo, xx^t *can be faster*. arXiv:2505.09814 [cs.DS], 2025.

[38] O. Schwartz, Y. Moran, and N. Vaknin, *Arithmetic and Communication Minimizing Fast Matrix Multiplication*. https://www.freepatentsonline.com/y2025/0013718.html, 2025. United States Patent Application 20250013718.

[39] O. Schwartz and N. Vaknin, *Pebbling Game and Alternative Basis for High Performance Matrix Multiplication*, SIAM Journal on Scientific Computing, 45 (2023), pp. C277–C303.

[40] A. Sedoglavic, *A non-commutative algorithm for multiplying (7 x 7) matrices using 250 multiplications*. working paper or preprint, Aug. 2017.

[41] ——, *Yet another catalogue of fast matrix multiplication algorithms.*, 2025. https://fmm.univ-lille.fr/.

[42] A. Smirnov, *The Bilinear Complexity and Practical Algorithms for Matrix Multiplication*, Comput. Math. and Math. Phys., 53 (2013), p. 1781–1795.

[43] ——, *Several Bilinear Algorithms for Matrix Multiplication*, 2017. Technical Report.

[44] P. Stankovski Wagner, *GitHub – C code repository fmm_add_reduction*. Available at github.com/werekorren/fmm_add_reduction, last accessed on 2026–04–23, 2024.

[45] J. Stapleton, *A 60-addition, rank-23 scheme for exact 3x3 matrix multiplication*, 2025.

[46] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13 (1969), p. 354–356.

[47] N. D. Türe and M. Cenk, *Efficient batch post-quantum signatures with crystals dilithium*, in Arithmetic of Finite Fields, S. Petkova-Nikova and D. Panario, eds., Cham, 2025, Springer Nature Switzerland, pp. 237–257.

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

Copyright © 2026
Copyright for this paper is retained by authors

A Illustrating Example This example illustrates the different parts of Section 2.3, using the example from Section 2.2.

The three problems refer to computing

$$
\begin{aligned}
& A_{11} \\
& \quad A_{12} \\
- & A_{11} \quad + A_{21} \\
& \qquad\quad A_{21} + A_{22} \\
- & A_{11} \quad + A_{21} + A_{22} \\
& A_{11} + A_{12} - A_{21} - A_{22} \\
& \qquad\qquad\quad A_{22}
\end{aligned}
$$

$$
\begin{aligned}
& B_{11} \\
& \quad B_{21} \\
& \quad B_{12} + \quad - B_{22} \\
- & B_{11} + B_{12} \\
& B_{11} - B_{12} + \quad + B_{22} \\
& \qquad\qquad B_{22} \\
- & B_{11} + B_{12} + B_{21} - B_{22}
\end{aligned}
$$

and

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

using a minimum number of additions and subtractions[10]. The three problems can be written on matrix form. For example, the last problem can be written as

$$
\begin{pmatrix} C_{11} \\ C_{12} \\ C_{21} \\ C_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \\ M_5 \\ M_6 \\ M_7 \end{pmatrix}. \quad \text{(A.1)}
$$

Now, performing the saving substitution $M_8 = M_1 + M_5$ means that we replace each occurrence of $M_1 + M_5$ by M_8. As the sum exists three times, this saves us two additions compared to computing the sum naively each time. The system resulting from applying the saving substitution to (A.1) can be written on matrix form as

$$
\begin{pmatrix} C_{11} \\ C_{12} \\ C_{21} \\ C_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \\ M_5 \\ M_6 \\ M_7 \\ M_8 \end{pmatrix}. \quad \text{(A.2)}
$$

Notice that this substitution added a column to the matrix with 1s for each row where the sum $M_1 + M_5$ occurred. Also notice that we now have zeros on the corresponding positions of columns 1 and 5.

Now, we see that the sum $M_3 + M_8$ occurs twice. Thus, performing the saving substitution $M_9 = M_3 + M_8$ on (A.2) saves one addition and results in the system

$$
\begin{pmatrix} C_{11} \\ C_{12} \\ C_{21} \\ C_{22} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ M_3 \\ M_4 \\ M_5 \\ M_6 \\ M_7 \\ M_8 \\ M_9 \end{pmatrix}. \quad \text{(A.3)}
$$

Studying (A.3) we see that there are no more saving substitutions. This completes our optimization process. Translating back to "additive" form we get

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

which matches the example from Section 2.2.

B An Example Where Saving Substitutions Cannot Find the Optimal Solution Consider the following problem[11]. Minimize the number of additions needed to compute

[10] Notice that the 'A' and 'B' cases include some single terms that require no actual computations, like A_{11}.

[11] We would like to thank an anonymous reviewer of the conference version of our paper [28] for finding this example where the optimal solution lies outside our framework.

Copyright © 2026
Copyright for this paper is retained by authors

$$\begin{array}{lcccccccc} C_1 & = & M_1 & + & M_2 & + & M_3 & + & M_4 \\ C_2 & = & M_1 & + & M_2 & + & M_3 & & \\ C_3 & = & M_1 & + & M_2 & & & + & M_4 \\ C_4 & = & M_1 & & & + & M_3 & + & M_4 \\ C_5 & = & & & M_2 & + & M_3 & + & M_4 \end{array} \tag{B.1}$$

Using saving substitutions, it is fairly easy to determine that we need seven additions to compute $C_1, \ldots, C_5$. However, only six additions and subtractions suffice in the following way.

$$\begin{array}{lcccc} T_1 & = & M_1 & + & M_2 \\ C_2 & = & T_1 & + & M_3 \\ C_1 & = & C_2 & + & M_4 \\ C_3 & = & C_1 & - & M_3 \\ C_4 & = & C_1 & - & M_2 \\ C_5 & = & C_1 & - & M_1 \end{array} \tag{B.2}$$

The secret to the improvement lies in computing C_1 in the third variable substitution and then using the fact that C_4 and C_5 can be computed by subtracting M_2 and M_1 from C_1 respectively. There we use a sum (C_1) and remove the respective terms that were not present in C_4 and C_5.

The methodology of using saving substitutions cannot find this type of saving, as it only builds sums by adding/subtracting terms that are explicitly present in the sum.

Also notice that computing C_1 does not make computing C_3 cheaper, as we can also compute C_3 as $T_1 + M_4$.

Copyright © 2026
Copyright for this paper is retained by authors