Hell's Bells: A Neural Network Pipeline for Ternary Fast Matrix Multiplication Algorithms

Erik Mårtensson[1,2,3], Paul Stankovski Wagner[1], and Joshua Stapleton[4,5]

[1]Lund University, Lund, Sweden, paul.stankovski_wagner@eit.lth.se
[2]Advenica AB, Malmö, Sweden
[3]Linköping University, Linköping, Sweden, erik.martensson@liu.se
[4]Independent Researcher, joshua.stapleton.ai@gmail.com
[5]Department of Computing, Imperial College London, UK

Abstract

We present a neural network-based pipeline for efficiently generating fast matrix multiplication (FMM) algorithms of small but arbitrary dimensions (n, m, k). Our neural network is general and tunable to output FMM schemes with specific properties, and in this paper we specifically target aspects that are useful and important in practical implementation, such as ternarity (coefficients in $\{-1, 0, 1\}$), sparseness and a low number of additions after optimization (addition reduction carried out separately).

We generate and optimize thousands of FMM algorithms and show that our generation method is beneficial in terms of performance across the entire FMM pipeline (both the FMM generation itself and optimization of additions). We discuss performance metrics and utilize heatmaps to visualize and understand this performance.

We achieve record-low arithmetic (additive) complexity for various combinations of dimensions. For $(n, m, k) = (2, 2, k)$, our method performs particularly well. Our improvement compared to previous results increases with k.

We show that the (addition) optimization process can behave very differently depending on the dimensions considered, indicating how further improvements (beyond our results) can be targeted. In particular, in the $(n, m, k) = (2, 2, k)$ setting, we show evidence of structural FMM properties coming into play, concretely showing that FMM generation with a minimal number of additions is sometimes suboptimal with respect to the entire FMM pipeline.

Finally, we make our neural network implementation, our generated FMM schemes, heatmap utilities and datasets publicly available.

1 Introduction

This work focuses on minimizing the number of additions for fast matrix multiplication (FMM) schemes of fairly small dimensions. As argued in [20], minimizing the arithmetic (additive) complexity of matrix multiplication algorithms has a practical impact similar to minimizing rank.

We take the holistic perspective from [21] and consider the entire FMM pipeline, as illustrated in Figure 1. As suggested in the illustration,
`FMM Scheme Generation` and `Addition Reduction` are essentially separate processes in the production of implementation-efficient and addition-optimized FMM schemes.

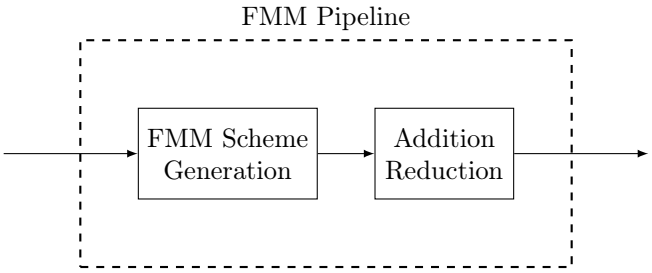


Figure 1: Pipeline for generation of implementation-efficient FMM Algorithms.

There are several different ways of generating FMM schemes, and there are also several ways of post-processing FMM schemes for optimization (addition reduction). In this paper we ask the following question: How can a holistic view of the FMM Pipeline improve production of *optimized* FMM schemes when using a neural network-based FMM generator to skew addition density probability distributions?

We provide a new neural network for ternary (coefficients in $\mathbb{T} = \{-1, 0, 1\}$) FMM algorithm generation that performs better than previous generation methods – in a holistic sense, and we concretely assess pipeline efficiency.

If you are somewhat familiar with neural networks and FMM algorithms, you can easily understand our neural network as fully connected, as illustrated in Figure 2 for the simpler case of multiplying two 2×2 matrices

$$\mathbf{AB} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \mathbf{C}.$$

When we examine the number of additions in FMM schemes, comparing them before and after optimization, we will see in Section 4 that other generation methods generate smooth bell-like probability distributions. In contrast, our neural network generates bell-shapes that are much more irregular, rather devilishly twisted and torn, as if drawn directly from Hell; we thus call these the Hell's Bells. Subsequently, we therefore also refer to our neural network as

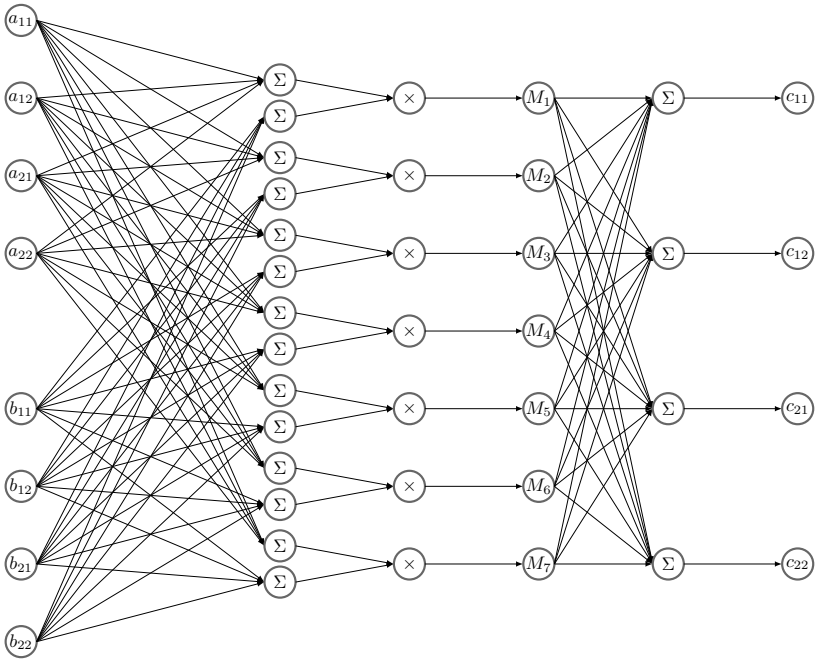


Figure 2: Illustration of the neural network for the case $n = m = k = 2$.

the Hell's Bells (neural) network, and to the FMM Pipeline when instantiated with the Hell's Bells network as the Hell's Bells Pipeline.

A bird's-eye interpretation of our method is that we tweak probability distributions in different ways in favor of improving the production quality of optimized FMM algorithms, aiming at practical implementation.

In a general context, multiplying 2×2 matrices directly from the definition requires eight multiplications and four additions. In 1969 Strassen [33] showed how to reduce the number of multiplications to seven, at the cost of increasing the number of additions to 18[1]. The number of multiplications is also called the rank of the algorithm. The rank seven was shown to be optimal in [36]. Strassen's algorithm reduces the asymptotic complexity of multiplying $n \times n$ matrices from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^{\log_2(7)})$. One branch of research has worked on minimizing the asymptotic complexity, culminating in the current record of $\mathcal{O}(n^{2.371339})$ [1]. This branch almost exclusively consists of galactic algorithms. This work only considers the other branch, that is, asymptotically fast, practically viable FMM algorithms.

For rank 7 algorithms for multiplying 2×2 matrices, the number of additions was reduced to 15 in [4]. The number of additions required is also referred to as the arithmetic complexity of the algorithm. Using a change of basis, Karstadt and Schwartz improved this number to 12 and showed it to be optimal in [13]. In

[1]When referring to additions, we implicitly include both additions and subtractions.

terms of computational complexity, performing this change of basis introduces lower-order terms that are small compared to the cost of performing an addition.

Now consider multiplying an $n \times m$ matrix by an $m \times k$ matrix. For most shapes with at least one dimension larger than 2, both the optimal rank and the optimal arithmetic complexity of matrix multiplication are not known. For three collections of low-rank schemes found for a large number of dimensions, see [17, 27, 29]. Note that [17] points out that results are beaten so frequently that publishing an exhaustive list in a paper becomes pointless as the list very quickly becomes outdated.

1.1 The 3×3 Case

For multiplying 3×3 matrices, Laderman found a rank 23 algorithm by hand in 1976 [18]. After almost half a century of applying increasingly sophisticated search methods, this rank has not been improved. The arithmetic complexity on the other hand has recently seen a number of improvements in quick succession.

In terms of the number of additions required for rank 23 algorithms for multiplying 3×3 matrices, Smirnov found an algorithm that naively requires 84 additions [30]. Smirnov pointed out that the number of additions for his algorithm can be lowered, but he did not perform such an optimization. Karstadt and Schwartz reduced the number of additions to 75 in [13]. Beniamini et al. reduced this number further to 61 using a change of basis in [3]. Without performing a change of basis, Mårtensson and Stankovski Wagner showed how to perform Laderman's original algorithm using only 62 additions [20], almost matching the result of Beniamini et al.

Recently, Stapleton developed a method for generating low-rank matrix multiplication schemes using a neural network [32]. By generating many schemes of rank 23 and reducing the number of additions using the implementations of Mårtensson and Stankovski Wagner, he found a 60-addition algorithm. In this work, we further combine Stapleton's method of generating fast matrix multiplication schemes with Mårtensson's and Stankovski Wagner's method for reducing the number of additions to find a scheme requiring only 59 additions. In an earlier preprint of this work, we give the full details of this scheme [22]. This includes the naive (unreduced) form of the scheme using 110 additions, as well as the optimized form, which requires only 59 additions. This also includes a specification of the scheme on a form readable by the software implementations of Mårtensson [19] and Stankovski Wagner [31]. We briefly comment on how our work relates to some recent improvements from [14, 24, 35] in Section 1.3.

1.2 Other Dimensions

In this work, we also show how to generalize the method introduced by Stapleton to arbitrary dimensions (n, m, k). For various dimensions, we improve the arithmetic complexity of previous constructions. Our method performs particularly well for dimensions of the form $(2, 2, k)$, where it outperforms previous methods more, the larger k becomes.

1.3 Related Works

The first example of finding fast matrix multiplication using machine learning is from Elser [6], who managed to recover Strassen's algorithm, as well as a rank-23 algorithm for multiplying 3×3 matrices. These results were replicated with a similar method in [2], where the authors argued heuristically why a rank-22 scheme for multiplying 3×3 matrices is unlikely to exist. Using significantly more computational resources and a different approach, DeepMind found efficient matrix multiplication schemes in [7, 23].

Recently, Perminov published a series of preprints relevant for this area of work [24, 25, 26]. In [25], he searched the flip graph for schemes with ternary coefficients. The dimensions where he makes improvements are too large for (at least the current version) of our approach to apply. In [26], he builds upon the method of [20] for reducing the additive complexity of matrix multiplication schemes, slightly improving the additive complexity in some settings. Finally, in [24] he traverses the flip graph of dimensions $(n, m, k) = (3, 3, 3)$, applying the reduction method of [26] until he finds an algorithm with the number of additions reduced to 58[2]. For more discussion, comparing our two results, see Section 4.5.

Starting from Perminov's scheme, Sun applies a transformation of it and then uses an exhaustive search among all straight-line programs (SLPs) to optimize the number of additions [35][3]. This way he achieves a scheme requiring only 56 additions. By applying another transformation of Perminov's scheme, combined with SLP, Karunaratne and Idamekorala reduce the number of additions to 55 [14].

Note that the reductions in [14, 35] both go outside the framework of both our work and the work by Perminov, both of which limit the search space to common subexpression elimination (CSE). Their works are compatible with generating many schemes with our method as well as the flip graph search of Perminov, and then reducing the number of additions. However, note that their methods are non-obvious to generalize, as they use the very particular structure of the scheme in question to be able to perform exhaustive SLP. See Section 2.2 for a more detailed introduction to the difference between the two reduction frameworks.

For sets of dimensions other than $(3, 3, 3)$ we compare with previous work in [3, 10, 13, 20, 21, 26, 28].

Acknowledgments

We thank Gemini 3.6 Flash and Chat GPT 5.6 for proofreading this manuscript, although all writing and modification of this manuscript were done by hand.

[2] A result he published concurrently with an early preprint of this work [22].

[3] The fact that the search is exhaustive follows by studying his script [34]. To be more precise, he applies exhaustive SLP to two out of three search problems, as the search space is too vast for the third one.

2 Notation and Background

We frequently use the compact indexing notation

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

We let $\mathcal{U}(a, b)$ denote the continuous, uniform distribution taking values between a and b. We let $\mathcal{N}(\mu, \sigma^2)$ denote a normal distribution with mean μ and variance σ^2. Given a matrix $\mathbf{A} \in R^{n \times m}$, we define its Frobenius norm as

$$\|A\|_F = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} |A_{ij}|^2}.$$

2.1 Strassen-type Matrix Multiplication Schemes

Given $\mathbf{A} \in R^{n \times m}$, $\mathbf{B} \in R^{m \times k}$ and consider $\mathbf{C} = \mathbf{AB} \in R^{n \times k}$ for some ring R. A Strassen-type algorithm of rank r computes $\mathbf{C}$ as

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

where $\alpha_{ij}^{(l)}, \beta_{ij}^{(l)}, \gamma_{ij}^{(l)} \in K$. In this work we consider ternary coefficients, that is, we let $K = \mathbb{T} = \{-1, 0, 1\}$. For some dimensions – most famously $n = m = k = 4$ – it is possible to improve the rank by letting $R = K = \mathbb{Z}_2$ [7]. For other settings it is possible to improve the rank by considering a larger set of coefficients for K, such as integer [15], rational [16, 30] or even complex coefficients [12, 23][4].

For a comparison of how the rank of ternary schemes compare against schemes with other coefficients, see [17, 27].

As detailed in [21], generating FMM schemes with low arithmetic complexity can be divided up into two parts or processes; generation of many schemes of minimized rank, and reduction of the number of additions of each scheme. Let us first briefly discuss the addition reduction part.

[4]The rank-48 scheme for the $n = m = k = 4$ setting from [23] was later simplified to only using rational coefficients in [5].

2.2 Reducing Additions in an FMM Scheme

Consider a system of sums to compute, as defined in (1). Compared to computing all additions and (scalar multiplications) naively, it is possible to save a significant number of operations by performing the computations in a clever order. Computing all 'A'-factors, 'B'-factors and 'C'-terms boils down to three instances of problems of the following format. Given $a_1, \ldots, a_q$, compute

$$s_1 = g_{11}a_1 + \cdots g_{1q}a_q, \tag{2}$$

$$\vdots \tag{3}$$

$$s_p = g_{p1}a_1 + \cdots g_{pq}a_q, \tag{4}$$

using a minimum number of additions and scalar multiplications. In this work, we consider schemes where $g_{ij} \in \mathbb{T}$. Thus, we restrict the operations to additions.

2.2.1 Common Subexpression Elimination

The works of Mårtensson-Stankovski Wagner and Perminov both restrict the search space to Common Subexpression Elimination (CSE). CSE works by repeatedly finding a sum/difference of two variables that exists in at least two of the p sums in (2)-(4). This is called a common subexpression. The method eliminates all occurrences of this subexpression by replacing it with a new variable. Then the process continues until there are no common subexpressions left. Once there are no common subexpressions left, then the remaining sums are computed naively.

The challenge in each step lies in deciding which of the common subexpressions to choose. In this work, we perform CSE from Mårtensson-Stankovski Wagner [20] to optimize the additions of the schemes that we generate.

2.2.2 Straight-Line Programming

A more general approach of computing (2)-(4) is by using Straight-Line Programming (SLP). Here we compute all the sums using t additions by forming

$$\begin{aligned} a_{q+1} &= a_{i_1} \pm a_{j_1}, \\ &\vdots \\ a_{q+t} &= a_{i_t} \pm a_{j_t}, \end{aligned}$$

where for all s_l, there is a k, such that $a_{q+k} = s_l$. Here, of course, $i_k, j_k \in \{1, \ldots, q+k-1\}$. Note that we can consider even more general ways of SLP, such as forming new partial sums by multiplying an existing variable by a scalar. However, this is likely not useful in our setting with ternary coefficients.

In [21] a toy example where SLP can outperform CSE for computing a system like in (2) was introduced. Whether this applies for problem instances coming from matrix multiplication schemes was left as an open question. Very soon afterwards, Sun [35] reduced the number of additions for multiplying 3×3 matrices from 58 to 56, by performing exhaustive SLP. This was then further improved to 55 in [14]. Both these works exploited the favorable properties of the scheme in question. We leave the general questions of how to efficiently perform SLP and how it performs compared to CSE for future research. Note that our method of generating schemes is compatible with any method for reducing their additions.

2.3 Generation of Many Low-Rank Schemes

The main contribution of this paper is a new method for generating schemes of low rank with good potential for low optimized arithmetic complexity. We describe our method in detail in Section 3.

3 The Hell's Bells Neural Network and Pipeline

Let us describe our neural network-based method for discovering fast matrix multiplication schemes. For more precise details on our method we refer to the implementation [31].

We consider the problem of finding the coefficients in (1) as the problem of finding the weights of a neural network. The values for the matrices $\mathbf{A}$ and $\mathbf{B}$ are considered as inputs to the network. The values $M_1, \ldots, M_r$ constitute a hidden layer. Finally, the values for the matrix $\mathbf{C}$ constitute the output of the neural network.

For a concrete illustration of the approach for dimensions $n = m = k = 2$, see Figure 2. The non-trivial weights correspond to all the edges that go into a sum/the symbol Σ, in the figure. The (trivial) weights for all other edges are implicitly equal to 1. The process of finding a matrix multiplication algorithm simply consists of figuring out the values for the non-trivial weights.

Our method of finding the solution consists of two steps; finding a real-valued solution, and turning the real-valued solution into a solution with ternary coefficients.

3.1 Network Weight Initialization

The network starts by independently assigning the random value to each non-trivial weight w as

$$w = \begin{cases} 1 & \text{with probability } 0.05, \\ -1 & \text{with probability } 0.05, \\ \sim \mathcal{U}\left(-\sqrt{6/f_{\text{in}}}, \sqrt{6/f_{\text{in}}}\right) & \text{with probability } 0.9. \end{cases} \tag{5}$$

Here, f_{in} is the fan-in, the number of inputs to that layer of the neural network. Concretely, this is equal to $r(nm + mk)$ for $\alpha_{ij}^{(l)}$ and $\beta_{ij}^{(l)}$, and for $\gamma_{ij}^{(l)}$ it is equal to rnk.

3.2 Sampling Matrices for Training and Testing

We create sets of matrices that are used during training and testing. We sample matrices in the following way. Each entry is sampled i.i.d. from $\mathcal{N}(0, 1)$. The matrix is then normalized to unit Frobenius norm. In this way, optimization is driven by inputs on a product of spheres rather than entrywise-independent matrices. Empirically, this induces a non-uniform distribution over converged ternary schemes, suggesting that the method currently explores a restricted but practically valuable subset of the full ternary search space[5].

3.3 Hell's Bells network Phase 1: Solution in $\mathbb{R}$

Loop the following steps a predetermined number of times (epochs).

1. Generate a batch of $b = 256$ random matrix pairs[6] $\mathbf{A}_i, \mathbf{B}_i$ and compute their products $\mathbf{C}_i = \mathbf{A}_i\mathbf{B}_i$.
2. Compute the corresponding approximations of the matrix products $\mathbf{C}'_i$ using the current network weights.
3. Compute the loss ℓ as the mean squared error of the approximation;

$$\ell = \text{MSE}(\{\mathbf{C}'_i, \mathbf{C}_i\}) = \frac{1}{b}\sum_{i=1}^{b} \|\mathbf{C}'_i - \mathbf{C}_i\|_F^2$$

4. If $\ell < \varepsilon$, then return `REAL SOLUTION CONVERGED`.
5. Update the network weights using ℓ.

If no solution was found, then return `REAL SOLUTION DID NOT CONVERGE`.

Network weights are updated using the (PyTorch) optimizer AdamW, which can be viewed as a gradient step decision maker in the (multi-dimensional) network weight search space.

3.4 Hell's Bells network Phase 2: Ternarization

If the process from Section 3.3 finds a real-valued solution, then we try to turn it into a solution with ternary coefficients. The process is essentially the same as in Section 3.3, with the main exception being how we penalize deviations from

[5] While it may be possible to initialize the weights to more precisely match that of a matrix multiplication schemes, it is unlikely to have a significant impact on the performance of our approach, which for specific values (n, m, k) draws values in a fairly narrow interval.

[6] this batch size empirically performs particularly well.

ternary weights. Loop the following steps for the remaining number of steps (epochs).

1. Generate a batch of $b = 256$ random matrix pairs[7] $\mathbf{A}_i, \mathbf{B}_i$ and compute their products $\mathbf{C}_i = \mathbf{A}_i\mathbf{B}_i$.

2. Compute the corresponding approximation of the matrix product $\mathbf{C}'_i$ using the current weights.

3. Compute the loss ℓ, defined in the following way.

$$\ell = \mathrm{MSE}(\{\mathbf{C}'_i, \mathbf{C}_i\}) + \lambda_{\text{sparse}} \sum_{w \in W} |w| + \lambda_{\text{ternary}} \sum_{w \in W} \Psi_T(w). \tag{6}$$

   Here λ_{sparse}, λ_{ternary} and T are real values that can be modified epoch by epoch, and

$$\Psi_T(w) = -T \ln \left(e^{-\frac{|w|}{T}} + e^{-\frac{|w-1|}{T}} + e^{-\frac{|w+1|}{T}} \right).$$

   The second term of (6) encourages the network to minimize the total weight. This is in line with [21], which observed that schemes with lower naive arithmetic complexity tend to have lower optimized arithmetic complexity.

   Each term in the sum of the third component of (6) makes the network value weights being close to $\mathbb{T} = \{-1, 0, 1\}$. Thus, the term as a whole makes the network value produce weights approaching this set. In fact, there is nothing magical about the set $\mathbb{T}$. We can generalize it to making the model approach any alphabet $\mathcal{A}$, by defining

$$\Psi_{T,\mathcal{A}}(w) = -T \ln \left(\sum_{a \in \mathcal{A}} e^{-\frac{|w-a|}{T}} \right).$$

4. To evaluate whether our solution is both correct and has coefficients close to the set $\{-1, 0, 1\}$ we compute

$$\ell_e = \mathrm{MSE}(\{\mathbf{C}'_i, \mathbf{C}_i\}) + 1 - D_{\mathbb{T}}(W),$$

   where

$$D_{\mathbb{T}}(W) = 1 - \frac{1}{|W|} \sum_{w \in W} \min\left(|w-1|, |w|, |w+1|\right). \tag{7}$$

   If $\ell_e < \varepsilon$, then we have a candidate solution and terminate the ternarization. If more than the pre-determined number of epochs have passed, then we also terminate and mark the run as `SOLUTION FAILED TO TERNARIZE`.

[7] this batch size empirically performs particularly well.

To generalize the process to check for closeness to an arbitrary alphabet $\mathcal{A}$, we would of course replace (7) with

$$D_{\mathcal{A}}(W) = 1 - \frac{1}{|W|} \sum_{w \in W} \min_{a \in \mathcal{A}} (|w - a|) .$$

5. Update the network weights using ℓ. Note that the loss value ℓ for updating the weights and the evaluation loss ℓ_e for checking if our solution is correct are different.

3.5 Verifying the Solution

If ternarization succeeds, then we make a hard snap, that is, we round each current weight w to the closest value in $\mathbb{T}$. We run a check to see that the algorithm we found indeed constitutes a correct matrix multiplication scheme. If we have indeed found a correct scheme, then as a post-processing step, we run the software of Mårtensson and Stankovski Wagner [20] to determine the arithmetic complexity of the solution.

3.6 Comments on our Particular Setting

Our setting is a bit unusual within the context of neural networks.

1. We can (cheaply) generate an unlimited amount of training data.
2. The neural network is an exact representation.

Despite these advantages, finding the correct weights of course remains a hard problem, since the problem of finding the lowest rank itself is hard, as showed by Håstad [8].

3.7 How Problem Size Grows with the Dimensions

Consider the general matrix multiplication problem for dimensions (n, m, k) and rank r. The number of non-trivial edge weights to compute, involving $'A'$ terms is rnm, the number of non-trivial edge weights to compute, involving $'B'$ terms is rmk, and the number of non-trivial edge weights to compute, involving $'M'$ terms is nkr. Thus, the total number of non-trivial edges to determine is equal to

$$r(nm + mk + nk). \tag{8}$$

This number is of course equal to the number of variables when stating the problem in terms of Brent equations.

3.8 Choice of Optimizer and Hyperparameters

For completeness, we discuss the choice of optimizer and hyperparameters in Appendix A.

4 Results

We first provide a summary of the additive complexities we find, before we go into our more detailed results relating to FMM generation efficiency and more structural observations.

4.1 Additive Complexity Summary

In Table 1 we summarize our results in terms of the arithmetic complexity that we find for various dimensions.

Table 1: Our additive complexities compared to the previous state-of-the-art for various dimensions.

			Previous best	
Dimensions	Rank	This paper	w/o CoB	w/ CoB
222	7	15	15 [28]	12 [13]
223	11	**17**	21 [20]	18 [3]
224	14	**26**	36 [20]	28 [3]
225	18	**28**	40 [20]	32 [3]
226	21	**38**	45 [21]	42 [10]
227	25	**41**	-	64 [10]
228	28	**51**	-	73 [10]
233	15	44	43 [21]	39 [3]
234	20	54	52 [21]	51 [3]
235	25	**70**	71 [26]	75 [10]
333	23	59	55 [14]	61 [3]

Notice that we limit ourselves to dimensions where $n \leq m \leq k$. Given a scheme for dimensions nmk of a specified rank and arithmetic complexity[8], it can be transformed into an algorithm of the same rank and arithmetic complexity for any permutation of the dimensions n, m and k. We reference the first publication at which the specific arithmetic complexity was demonstrated, not where the algorithm was first found on naive, non-optimized form. All cited comparisons refer to the best reference results that we can find, not necessarily the best result that can be achieved using known methods.

Generally, looking at the addition count distributions for the optimized schemes, it tends to look like a skewed normal distribution, with some noticeable exceptions.

4.2 How to Read Addition Reduction Heatmaps

We utilize addition reduction heatmaps to illustrate performance aspects of FMM algorithm generators. Recall that our main end goal is to skew the output

[8]For convenience we simply write a set of dimensions (n, m, k) as nmk in many places within this section.

probability distributions of the number of additions *after* reduction towards minimal values.

Our heatmaps plot how dense the resulting FMM algorithms are before (x-axis) and after (y-axis) addition reduction, in terms of the number of additions. The raw data, available in [31], is discrete and two-dimensional, with the minimum and maximum values displayed on both axes, so there are no values outside the displayed region. You may appropriately think of the raw data as a two-dimensional histogram, bucketing pairs of integers, but we have opted for a color coding with smearing for improved visual interpretation of the probability distributions.

In Figure 8, for example, we pair the heatmap with a histogram to clearly provide a direct end goal visual of the resulting probability distribution of the number of additions after optimization. This histogram corresponds to looking at the raw heatmap data from the y-axis.

4.3 The Hell's Bells Network is More Efficient than Uniform FMM Algorithm Generators

In Figure 3 we compare the performance of the Hell's Bells network to the FMM generator used by Heule et al. [9]. While it is very involved to clearly define what drawing an FMM algorithm uniformly at random actually means, we can still substantiate our efficiency claim by making this comparison.

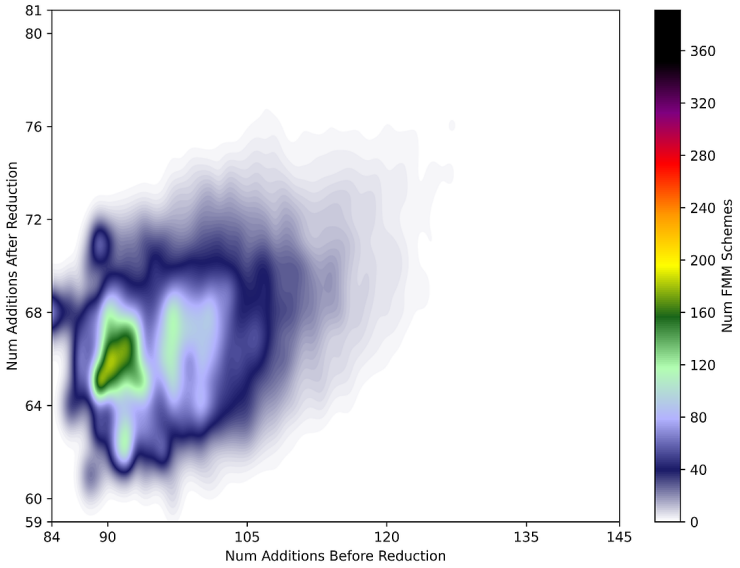
It is clear from looking at the near-perfect binomial bump in Figure 3b that the Heule FMM generator draws FMM algorithms close to uniformly at random in some (not technical but) intuitive and natural sense.

Note that the slightly stretched shape along the diagonal indicates a positive correlation between the number of naive and reduced additions, indicating that it is FMM pipeline-optimal to minimize additions during FMM generation in the $(n, m, k) = (3, 3, 3)$ setting.

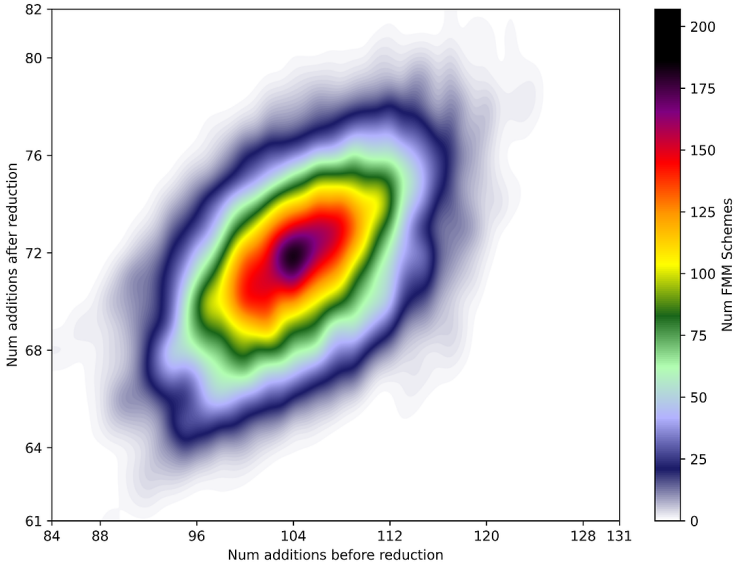
It is very clear that the Hell's Bells network delivers probability distributions that are much better in the sense that they are heavily skewed towards a smaller number of additions in the optimized schemes. The FMM schemes generated by the Hell's Bells network are biased to have lower naive number of additions, and, by extension, our full Hell's Bells pipeline delivers a lower optimized number of additions. This is visually apparent in Figure 3a, as the shape of the probability distribution is more comet-shaped, with the center of mass shifted significantly down to the left along the diagonal, compared to Figure 3b.

Looking at just the optimized number of additions in Figure 4, the difference in efficiency is similarly apparent[9].

[9] Aiming at a minimal number of additions after reduction, it is appropriate to compare the left tails of the probability distributions, whereas the middle and right parts of the distributions are not relevant in this setting.



(a) Heatmap for the Hell's Bells network.



(b) Heatmap from [21] for Heule et al.

Figure 3: Addition reduction heatmap comparison for our FMM generation method vs. that of Heule et al. for the 333 setting.

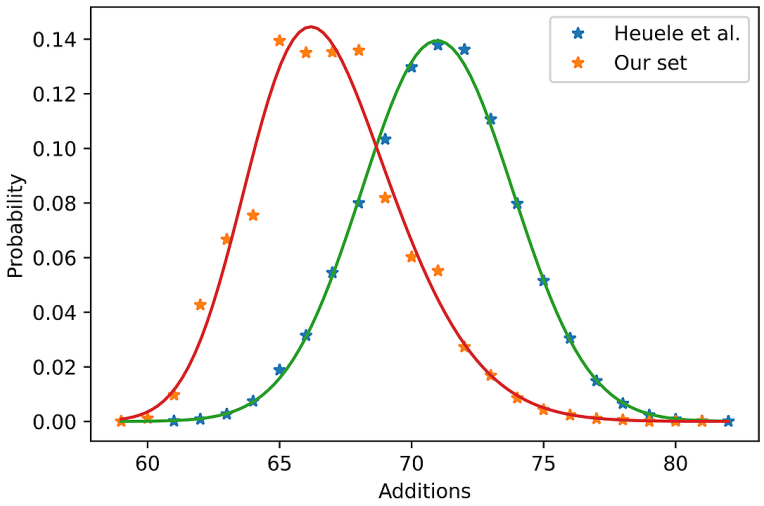


Figure 4: Probability distributions for the optimized number of additions in different sets of FMM algorithms.

4.4 Parameter Order May be Significant

It is not immediately clear that the Hell's Bells network treats the parameters (n, m, k) equally in a symmetric sense. That is, running the Hell's Bells network with parameters nmk or, e.g., the computationally equivalent kmn may potentially yield different results – different probability distributions with respect to additive weight.

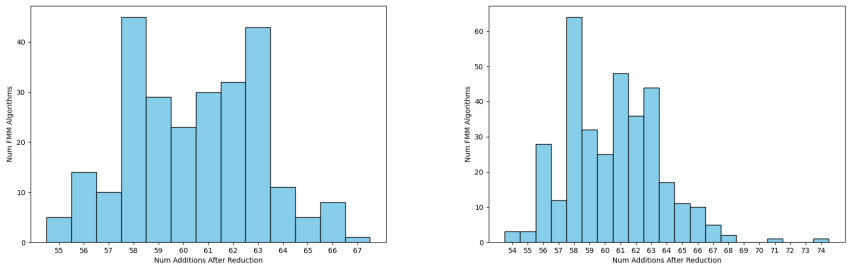
To examine this point further, we compare the case for 234 vs. 432, shown in Figure 5. We see a slightly higher variance for 432, which is beneficial when targeting minimality.

For 235 vs. 532 we observed a similar slight difference and opted to go with 532 in our simulations (reported as 235 below). However, the number of samples is not large enough for a conclusive result, so our conclusion is that parameter order *may* be significant for performance in the current implementation. More detailed investigations are left for future studies.

4.5 The 333 Setting

In the 333 setting we achieve 59 additions (see Figure 6, first reported in [22]), which was subsequently improved by, in turn, Perminov's 58 additions in [24], Sun's 56 additions [35] and Karunaratne's and Idamekorala's 55 additions. Let us comment on this chain of results in some more detail.

In [24], Perminov managed to find a scheme requiring only 58 additions. His approach was to start from a rank-23 scheme and then search the vast flip graph



(a) 234 setting (256 samples). (b) 432 setting (342 samples).

Figure 5: Addition count distribution comparison for the 234 and 432 settings. The Hell's Bells network shows some signs of asymmetric behavior with respect to parameter order nmk vs. kmn.

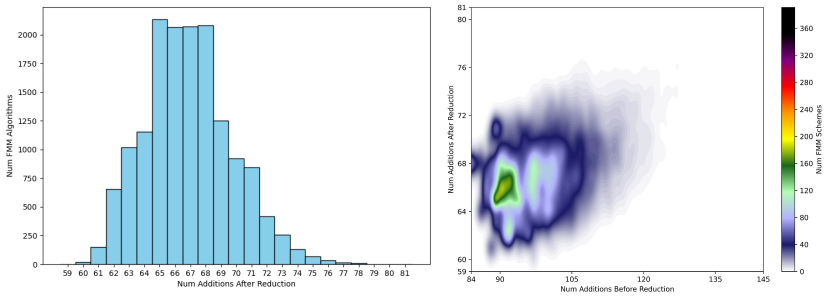


Figure 6: Addition distribution and reduction heatmap for the 333 setting with 15,316 sampled FMM algorithms.

in its neighborhood until he found a scheme with low arithmetic complexity. Compared to us, he explores a larger number of schemes, but the difference from one scheme to the next is very minor.

By transforming Perminov's scheme and applying SLP instead of CSE (see Appendix B of [21] for some further discussion), Sun found a scheme requiring only 56 additions. This was further improved to 55 additions using a similar approach in [14]. Note that these improvements come from performing reduction of additions more efficiently, compared to both our work and Perminov's, and not from generating schemes in a more efficient way.

While we do not deliver a new record in the 333 setting, we do leave the setting under-explored since we, contrary to Sun and Perminov, have neither applied local search around our FMM algorithm outputs, nor more/other/stronger addition reduction methods than those in [20]. We leave exploration of more advanced such FMM pipelines for further research.

4.6 The $(n, m, k) = (2, 2, k)$ Parameter Regime

Studying the results of Table 1 we find the $(n, m, k) = (2, 2, k)$ setting particularly interesting. The corresponding addition count distributions and heatmaps for dimensions 222 to 228 are found in Figures 7 to 13.

222

The 222 setting is trivial and is included for completeness. The respective numbers with and without change of basis here are known to be optimal. Note that the addition count distribution deviates a lot from a (skewed) normal distribution, and finding solutions that are equal to the optimal 15 is very easy.

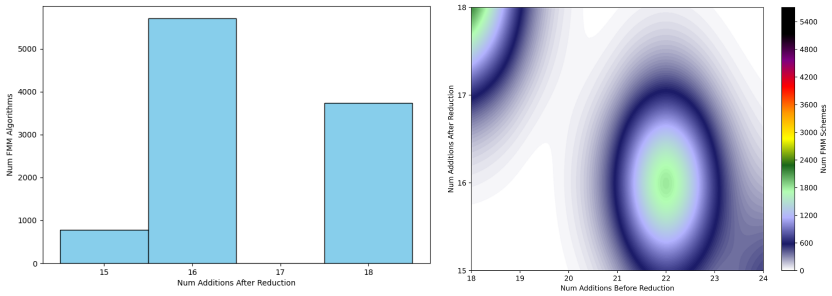


Figure 7: Addition distribution and reduction heatmap for the 222 setting with 10,227 sampled FMM algorithms.

223

Notice for the 223 setting, that our result can easily be matched by performing Strassen's algorithm on Winograd's form, together with a naive multiplication of a 2×2 matrix and a column vector. Nevertheless, this result has – as far as we know – not been published elsewhere. Note that the addition count distribution deviates quite a lot from a (skewed) normal distribution. Finding solutions that are close or equal to 17 (which might be optimal) is very easy.

224

Without a change of basis, 30 additions can easily be achieved by dividing the computation into two independent multiplications of 2×2 matrices. Note that we do significantly better than that with 26 additions.

225

Our 28 additions here can also be achieved by combining our results for the 224 setting with a naive multiplication of a 2×2 matrix and a column vector.

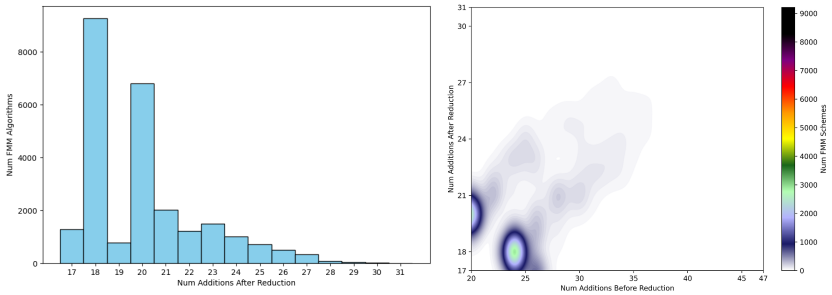


Figure 8: Addition distribution and reduction heatmap for the 223 setting with 25,625 sampled FMM algorithms.

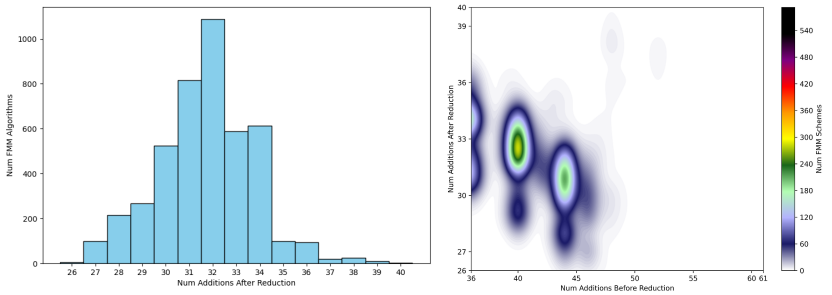


Figure 9: Addition distribution and reduction heatmap for the 224 setting with 4,464 sampled FMM algorithms.

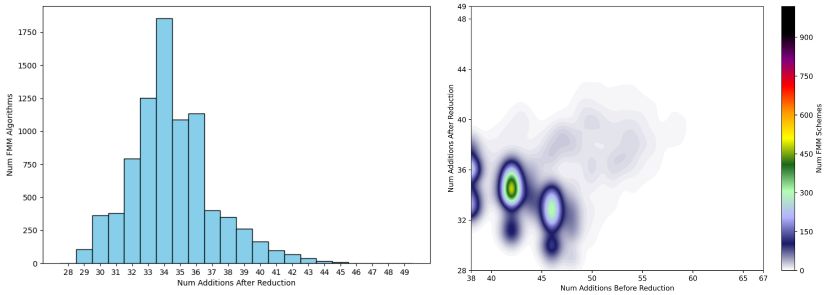


Figure 10: Addition distribution and reduction heatmap for the 225 setting with 8,378 sampled FMM algorithms.

226

Our 38 additions here constitute a non-trivial improvement compared to our own results of lower dimensions and against previous state-of-the-art. For ex-

ample, combining our results from the 224 setting with Strassen's algorithm on Winograd's form leads to 26 + 15 = 41 additions.

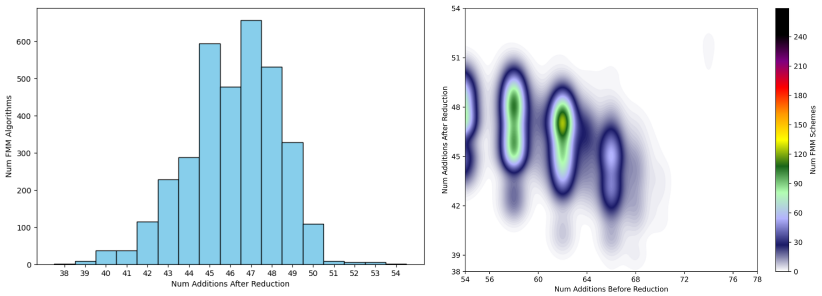


Figure 11: Addition distribution and reduction heatmap for the 226 setting with 3,438 sampled FMM algorithms.

227

Our 41 additions here constitute a non-trivial improvement compared to our own results of lower dimensions and against previous state-of-the-art. For example, combining our results from the 225 setting with Strassen's algorithm on Winograd's form leads to 28 + 15 = 43 additions. For this setting, we do not find any previous published results optimizing the number of additions in the setting without a change of basis.

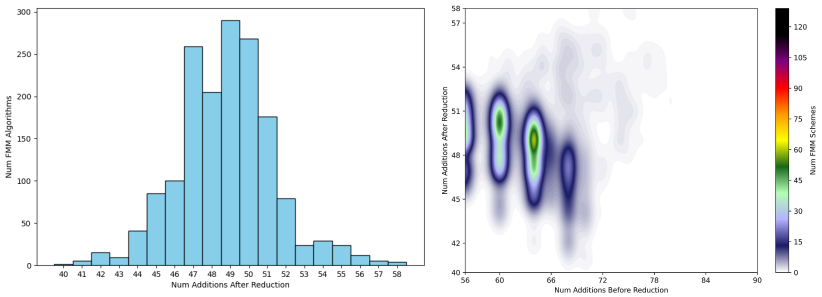


Figure 12: Addition distribution and reduction heatmap for the 227 setting with 1,631 sampled FMM algorithms.

228

Our 51 additions here constitute a non-trivial improvement compared to our own results of lower dimensions and against previous state-of-the-art. Combin-

ing our results from the 224 setting twice gives an algorithm with 52 additions. Combining our results from the 226 setting with Strassen's algorithm on Winograd's form leads to $38 + 15 = 53$ additions. For this setting, we do not find any previous published results optimizing the number of additions in the setting without a change of basis.

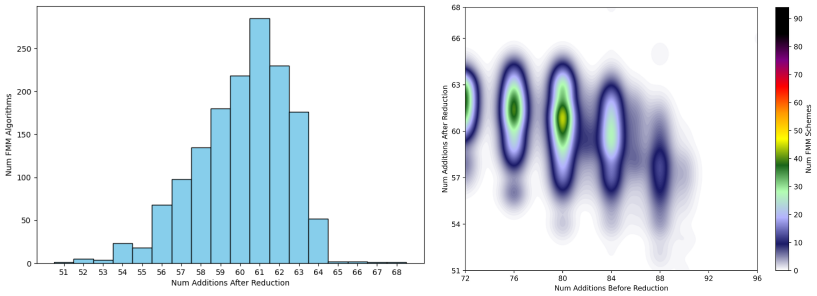


Figure 13: Addition distribution and reduction heatmap for the 228 setting with 1,499 sampled FMM algorithms.

4.7 Heatmap Clustering Reveals Structural Properties

The sequence of heatmaps in Figures 7 to 13 for the $22k$ schemes reveals further interesting insights. Firstly, the clustering indicates that there are different classes of FMM algorithms that fare better or worse in terms of number of additions after reduction. In other words, clustering reveals structural properties that are visible in the heatmap space.

This means that it is likely possible to extract the precise FMM algorithm feature or feature set that define a specific cluster. This may, of course, be useful for pushing the search for more efficient algorithms even further.

We note that no previous research efforts have explicitly targeted problem structure for efficient FMM algorithm generation, so this opens up very interesting targets for future research.

The clustering we see in the $22k$ setting seems to follow a pattern; 222 and 223 define two clusters, for 224 and 225 there are three, for 226 and 227 there are four, for 228 (and presumably 229) there are five.

4.8 Lower Weight is Suboptimal for Some Parameter Sets

Last but not least, the sequence of heatmaps for the $22k$ schemes also clearly shows that the FMM algorithms in the rightmost cluster (with highest naive weight) perform best in terms of the number of additions after reduction, which shows that weight minimization during FMM generation is suboptimal over the entire FMM pipeline for these parameter sets.

Regime $22k$ with $k > 8$ and Scaling Limits

We did not attempt to generate $22k$-schemes for $k > 8$, so we do not know the maximal k for which the Hell's Bells network can generate FMM algorithms with the current implementation. However, since pushing for low weight is suboptimal, it makes more sense to first modify the Hell's Bells network to target a specific weight to maximize the efficiency of FMM generation in the $22k$ parameter regime. This is suggested as further research.

4.9 The $(n, m, k) = (2, 3, k)$ Parameter Regime

The remaining results from Table 1 correspond to dimensions of the format $(2, 3, k)$, whose addition count distributions and heatmaps are found in Figures 14 to 17. Generally in these settings, we see some clustering around certain addition counts, making them more frequent than if drawing values from a (skewed) normal distribution.

233

In the 233 setting, with our 44 additions we manage to improve the 46 additions reported by Perminov in [26]. However, we do not improve the state-of-the-art as a solution with 43 additions was achieved in [21] by reducing the Hopcroft-Kerr scheme [11] in the equivalent 323 setting.

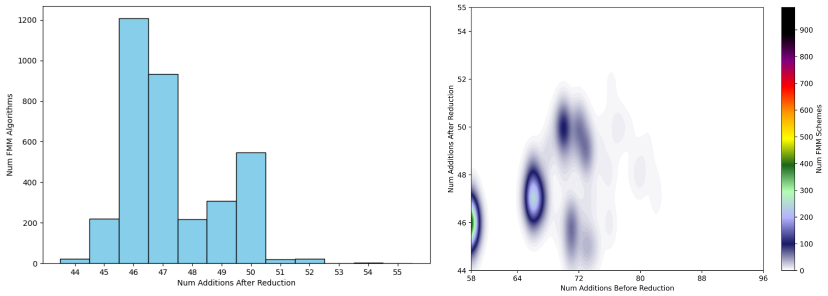


Figure 14: Addition distribution and reduction heatmap for the 233 setting with 3,500 sampled FMM algorithms.

234/432

In the 234 setting, our 54 additions improve the result of 58 additions presented by Perminov [26]. However, in [21] one of the equivalent 423 schemes was reduced to 52 additions.

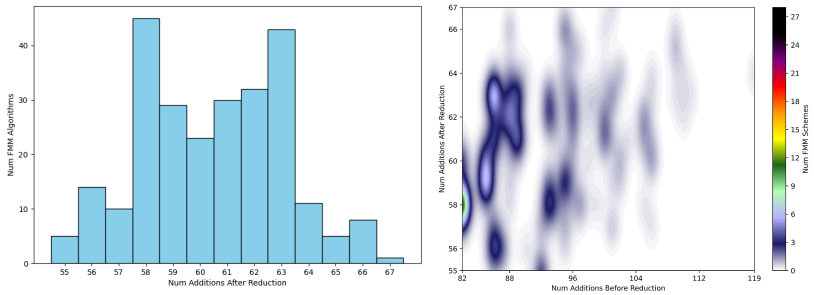


Figure 15: Addition distribution and reduction heatmap for the 234 setting with 256 sampled FMM algorithms.

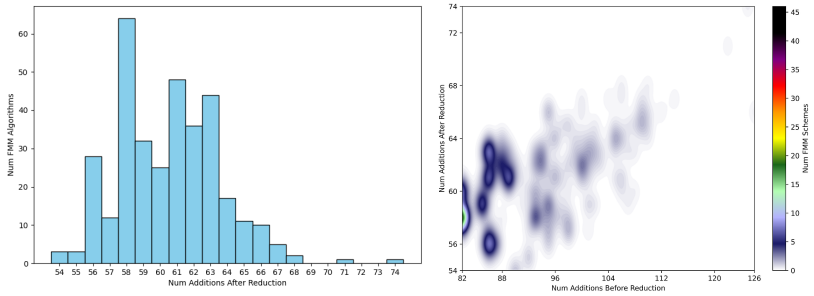


Figure 16: Addition distribution and reduction heatmap for the 432 setting with 342 sampled FMM algorithms.

235/532

In the 235 setting[10] our 70 additions improve the published state-of-the-art solutions both with and without a change of basis. Interestingly, this distribution is not at all Normal, and the minimally obtained score of 70 additions is also the most common addition count in this setting! Unlike the possible interpretations for some smaller dimension settings, like 222 and 223, we do not believe that this indicates that 70 additions is optimal.

4.10 Remaining Parameters

The subsections above cover the dimensions in which we managed to find solutions. For dimensions of type $22k$, we arbitrarily stopped at $k = 8$, while our method could have found solutions for larger values of k. For dimension types larger than reported here, such as 334, we did not manage to generate solutions.

[10] As indicated before, we technically study the computationally equivalent 532 setting.

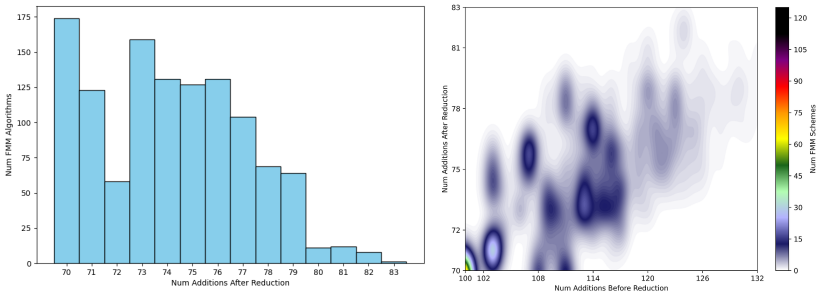


Figure 17: Addition distribution and reduction heatmap for the 532 setting with 1,023 sampled FMM algorithms.

We believe that our method should be able to find solutions also for larger dimensions, and the path forward in this direction is not only to improve the implementation by tweaking the hyperparameters, but also more fundamentally to incorporate structural information, evolving more intelligent neural networks that are targeted at specific parameters or parameter sets.

5 Summary and Future Work

We introduce a neural network-based pipeline for efficiently generating FMM algorithms of small dimensions (n, m, k), achieving record-breaking arithmetic complexities for various dimensions. By generating and analyzing thousands of schemes, we study the connection between the arithmetic complexity before and after reducing additions, revealing structural properties of FMM. By making the implementation, schemes and data sets available, we facilitate future research in this direction.

5.1 List of Future Work

There are many potential improvements and generalizations of the method presented in this paper, most of which are scattered throughout the paper. Here we compile them into a list for convenience.

- The hyperparameters, detailed in Table 2, were set to fit the 333 setting, and subsequently applied to other dimension settings. The only optimization made for specific settings was adjusting the number of epochs. Tailoring the hyperparameters for each specific parameter set nmk should yield improved results.

- As described in Section 3.4, we ternarize the solution to get coefficients in the set $\{-1, 0, 1\}$. As also discussed there, this method can be applied to an arbitrary, finite set of coefficients.

- Our method of generating many solutions can be combined with Perminov's method of searching the flip graph for a vast number of schemes in the neighborhood of a solution. It is also possible to consider transformations of solutions, as Sun did in [35], applying a transformation of the solution originally found by Perminov.
- As demonstrated by Sun [35] for some schemes, going beyond CSE substitutions can lead to a lower number of additions. Combining that idea with our scheme generation could also lead to improved results.
- The initial state of the optimization procedure is set to something fairly arbitrary and generic. It might be possible to improve by using knowledge of the structure of the matrix multiplication problem at hand.
- Currently, our scheme generation tries to minimize the total weight, that is, the naive additive complexity. In the $22k$ parameter regime, this is clearly suboptimal. Can we improve by optimizing for another heuristic? One possible idea here is to use the potential metric from [20].
- We train and test the network by letting it attempt to multiply a batch of pairs of matrices. Instead, it should be more efficient to train/test it against the general formula for matrix multiplication.
- Heatmaps show a clustering effect, as we saw in Section 4.7. How can this be exploited? In Figure 10 we see that 46 naive additions lead to lowest reduced number of additions. Is this specific to our method or does it reveal something about FMM for that dimension? For the $22k$ case, it seems as k increases that generating schemes with higher weights lead to lower optimized number of additions. This suggests that it might be beneficial to have the FMM generation process targeting FMM algorithms of a specific weight. An even higher goal is to understand how the underlying FMM structures induce the observed clustering.
- The $(2, 3, 5)$ setting stands out as odd with 70 additions being both the best and most common addition count score. Figuring out the mystery of this probability distribution has the potential of lowering the addition count for this setting and perhaps also for other settings.

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)*, pages 2005–2039. SIAM, 2025.

[2] P. Andreini, A. Bernardi, M. Bianchini, B. T. Corradini, S. Marziali, G. Nunziati, and F. Scarselli. Neural Learning of Fast Matrix Multiplica-

tion Algorithms: A StrassenNet Approach. arXiv:2602.21797 [math.AG], 2026.

[3] 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.

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

[5] 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.SC], 2025.

[6] V. Elser. A network that learns Strassen multiplication. *J. Mach. Learn. Res.*, 17(1):3964–3976, Jan. 2016.

[7] 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*, 610:47–53, 2022.

[8] J. Håstad. Tensor rank is NP-complete. In G. Ausiello, M. Dezani-Ciancaglini, and S. R. Della Rocca, editors, *Automata, Languages and Programming*, pages 451–460, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[9] M. J. H. Heule, M. Kauers, and M. Seidl. Local Search for Fast Matrix Multiplication. In M. Janota and I. Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 155–163, Cham, 2019. Springer International Publishing.

[10] 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)*, pages 322–334. SIAM, 2025.

[11] J. E. Hopcroft and L. R. Kerr. On Minimizing the Number of Multiplications Necessary for Matrix Multiplication. *SIAM Journal on Applied Mathematics*, 20(1):30–36, jan 1971.

[12] I. Kaporin. Finding Complex-Valued Solutions of Brent Equations Using Nonlinear Least Squares. *Comput. Math. and Math. Phys.*, pages 1881–1891, 2024.

[13] E. Karstadt and O. Schwartz. Matrix Multiplication, a Little Faster. *J. ACM*, 67(1), jan 2020.

[14] S. Karunaratne and A. Idamekorala. 55 Additions Suffice for 3x3 Matrix Multiplication at Rank 23. arXiv:2607.28676 [cs.CC], 2026.

[15] M. Kauers and J. 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.

[16] M. Kauers and I. Wood. Exploring the Meta Flip Graph for Matrix Multiplication. arXiv:2510.19787 [cs.SC], 2025.

[17] B. C. Lacelle. A catalog of fast matrix multiplication algorithms with frontier-closure search. arXiv:2606.13408 [cs.SC], 2026.

[18] J. D. Laderman. A Noncommutative Algorithm for Multiplying 3×3 Matrices Using 23 Multiplications. *Bulletin of the American Mathematical Society*, 82(1):126 – 128, 1976.

[19] E. Mårtensson. GitHub – Python code repository fmm_add_reduction, 2024. At `https://github.com/ErikMaartensson/fmm_add_reduction`, last accessed on 2026–08–14.

[20] E. Mårtensson and P. Stankovski Wagner. The Number of the Beast: Reducing Additions in Fast Matrix Multiplication Algorithms for Dimensions up to 666. In *2025 Proceedings of the Conference on Applied and Computational Discrete Algorithms (ACDA)*, pages 47–60. SIAM, 2025.

[21] E. Mårtensson and P. Stankovski Wagner. How and Why to Minimize the Arithmetic Complexity for Fast Matrix Multiplication Algorithms. Cryptology ePrint Archive, Paper 2026/849, 2026.

[22] E. Mårtensson, P. Stankovski 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.

[23] 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. arXiv:2506.13131 [cs.AI], 2025.

[24] A. I. Perminov. A 58-Addition, Rank-23 Scheme for General 3x3 Matrix Multiplication. arXiv:2512.21980 [cs.DS], 2025.

[25] A. I. Perminov. Fast Matrix Multiplication via Ternary Meta Flip Graphs. arXiv:2511.20317 [cs.SC], 2025.

[26] A. I. Perminov. Parallel Heuristic Exploration for Additive Complexity Reduction in Fast Matrix Multiplication. arXiv:2512.13365 [cs.SC], 2025.

[27] A. I. Perminov. Meta Flip Graph meets Serendipitous Product: new Fast Matrix Multiplication results. arXiv:2606.02480 [cs.SC], 2026.

[28] R. L. Probert. On the Additive Complexity of Matrix Multiplication. *SIAM Journal on Computing*, 5(2):187–203, 1976.

[29] A. Sedoglavic. Yet another catalogue of fast matrix multiplication algorithms., 2026. At `https://fmm.univ-lille.fr/`, last accessed on 2026–08–14.

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

[31] P. Stankovski Wagner. GitHub – C code repository fmm_add_reduction, 2024. At `https://github.com/werekorren/fmm_add_reduction`, last accessed on 2026–08–14.

[32] J. Stapleton. A 60-Addition, Rank-23 Scheme for Exact 3x3 Matrix Multiplication. arXiv:2508.03857 [cs.DS], 2025.

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

[34] Y. Sun. 3x3 matmul, rank 23, 56 additions, 2026. At `https://github.com/sunyinqi0508/3by3r23-56a`, last accessed on 2026–08–14.

[35] Y. Sun. An Exact 56-Addition, Rank-23 Scheme for General 3*3 Matrix Multiplication. arXiv:2604.27645 [cs.DS], 2026.

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

A Optimizer and Hyperparameters

For convenience, in Table 2 we compile all our hyperparameters and the values we use for them. The first category - Input - is simply input parameters that the algorithm itself does not set. The rank r is chosen as the best known rank for that particular set of dimensions (n, m, k).

We vary the number of epochs depending on the dimensions (n, m, k) to roughly maximize the expected number of ternary schemes found per unit of time. Having a batch size of 256 seems to be a good tradeoff.

As an optimizer we use AdamW. We specify hyperparameters relevant to AdamW in Table 2. Among them, the value 0.97 for β_2 is a non-standard value that we found to be effective. Once again, we refer to the implementation [31] for even more details.

Table 2: Algorithm hyperparameters

Category	Parameter	Values
Input	Matrix dimension (n)	Varies
	Matrix dimension (m)	Varies
	Matrix dimension (k)	Varies
	Algorithm rank (r)	Varies
Training	Epochs	Varies
	Batch Size	256
Optimization	Optimizer	AdamW
	Initial Learning Rate	$8 \cdot 10^{-3}$
	β_1 (Momentum)	0.9
	β_2	0.97
	Weight Decay	0.01
Ternarization and Sparsification	Ternary Trigger Loss	$3 \cdot 10^{-6}$
	Ternary λ	10^{-4}
	Sparsity λ	$3 \cdot 10^{-6}$
	Temperature T	10^{-3}