Multiplying 2×2 Sub-Blocks Using 4 Multiplications

Yoav Moran*
The Hebrew University of Jerusalem
Jerusalem, Israel
yoav.gross@mail.huji.ac.il

Oded Schwartz*
The Hebrew University of Jerusalem
Jerusalem, Israel
odedsc@cs.huji.ac.il

ABSTRACT

Fast parallel and sequential matrix multiplication algorithms switch to the cubic time classical algorithm on small sub-blocks as the classical algorithm requires fewer operations on small blocks. We obtain a new algorithm that can outperform the classical one, even on small blocks, by trading multiplications with additions. This algorithm contradicts the common belief that the classical algorithm is the fastest algorithm for small blocks. To this end, we introduce commutative algorithms that generalize Winograd's folding technique (1968) and combine it with fast matrix multiplication algorithms. Thus, when a single scalar multiplication requires ρ times more clock cycles than an addition (e.g., for 16-bit integers on Intel's Skylake microarchitecture, ρ is between 1.5 and 5), our technique reduces the computation cost of multiplying the small sub-blocks by a factor of $\frac{\rho+3}{2(\rho+1)}$ compared to using the classical algorithm, at the price of a low order term communication cost overhead both in the sequential and the parallel cases, thus reducing the total runtime of the algorithm. Our technique also reduces the energy cost of the algorithm. The ρ values for energy costs are typically larger than the ρ values for arithmetic costs. For example, we obtain an algorithm for multiplying 2×2 blocks using only four multiplications. This algorithm seemingly contradicts the lower bound of Winograd (1971) on multiplying 2×2 matrices. However, we obtain this algorithm by bypassing the implicit assumptions of the lower bound. We provide a new lower bound matching our algorithm for 2×2 block multiplication, thus showing our technique is optimal.

ACM Reference Format:
Yoav Moran and Oded Schwartz. 2023. Multiplying 2×2 Sub-Blocks Using 4 Multiplications. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3558481.3591083

1 INTRODUCTION

Matrix multiplication is a core operation in many algorithms in various fields, including numerical linear algebra, machine learning, data mining, image and signal processing, and in many graph algorithms. As such, many researchers try to improve its performance.

Fast matrix multiplication algorithms are a class of divide-and-conquer algorithms which use a base $\langle l, m, n; t \rangle$-algorithm recursively: an algorithm to multiply an $l \times m$ matrix by an $m \times n$ matrix using t scalar multiplications, where $l, m, n, t \in \mathbb{N}$. The base cases of fast algorithms use fewer multiplications than the classic algorithm. Therefore, they use less multiplications and linear operations asymptotically. However, they require more operations in the base case. As a result, efficient fast algorithm implementations call the classical algorithm on sufficiently small blocks (cf. [21, 31, 45]).

1.1 Commutative algorithms

A matrix multiplication algorithm is *commutative* if it requires that any two elements x, y of the input matrices satisfy $x \cdot y = y \cdot x$. Commutative algorithms trade up to half of the multiplications for additions compared to non-commutative ones. That is, they use fewer multiplications at the cost of more additions (see Theorem 3.3). This means that the run time of commutative algorithms may be better than their non-commutative counterparts when a single scalar multiplication costs more than an addition. However, matrix multiplication is not commutative in general ($A \cdot B \neq B \cdot A$). Therefore, commutative algorithms cannot be used recursively the same way fast algorithms are. Commutative algorithms can still be used at the base case of another recursive algorithm.

1.2 Previous works

In 1969, Strassen [45] discovered the first sub-cubic matrix multiplication algorithm. He did so using a bilinear algorithm for multiplying two 2×2 matrices, using 7 multiplications instead of 8. Following research split into two main directions. The first focuses on asymptotic improvements (cf. [1, 8, 15–17, 23, 41, 48]). This line of work includes theoretically appealing breakthroughs but typically renders astronomical constants hidden in the O-notation, which makes them impractical. The second research direction focuses on improving the run-time for multiplying reasonably sized matrices. This direction includes finding new algorithms with reasonable base case size and leading coefficients (cf. [20, 24, 25, 37, 38, 43]), improving the leading coefficients of existing algorithms (cf. [5, 6, 14, 31]), and reducing communication costs and memory footprint (cf. [3, 4, 7, 19, 32, 34]).

In 1968, Winograd [49] found the first algorithm that requires less than n^3 multiplications: a commutative algorithm for multiplying two $n \times n$ matrices using only $\frac{n^3}{2} + n^2$ multiplications instead of n^3. Since then, most of the work on improving commutative matrix multiplication has focused on trading multiplications with linear operations, either by creating new algorithms (cf. [2, 40, 47]) or by combining commutative algorithms with fast non-commutative ones (cf. [27, 28]). Lower bounds for commutative algorithms include [9, 10, 26, 50, 51]. The numerical stability of Winograd's

*Both authors contributed equally to this research.

SPAA '23, June 17–19, 2023, Orlando, FL, USA.

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591083

algorithm has been studied and shown to be compatible with the classical algorithm if carefully scaled [12].

1.3 Our contribution

We provide an alternative algorithm to the classical one for small block multiplications. We obtain this by introducing a method for composing a non-commutative algorithm with a commutative one (see Section 3.1). This composition reduces the arithmetic cost compared to the existing one. We thus save multiplications and linear operations compared to existing commutative algorithms while increasing the I/O-complexity by a low-order term only. Furthermore, we demonstrate a trade-off between the interprocessor I/O-complexity, the arithmetic complexity, and the communication between cache levels. Particularly, we can avoid the interprocessor I/O-complexity overhead at the cost of a slight increase in the arithmetic and the communication between cache levels.

Our technique attains the lower bound, and replaces half of the multiplications with additions, regardless the size of the sub-blocks, as long as the matrices are sufficiently large (see Tables 1 and 3). In contrast, existing commutative algorithms require the block size to be huge in order to attain this tradeoff. Particularly, for 2×2 blocks we show that our algorithm breaks existing lower bounds by sidestepping their assumptions and attains the new ones. In general, our algorithm is optimal when the classic algorithm is an optimal (non-commutative) algorithm, see Theorem 1.1.

Theorem 1.1. *Let $n \in \mathbb{N}$ be an even integer. Assume the classical $\langle n, n, n; n^3 \rangle$-algorithm has the lowest arithmetic cost among all $\langle n, n, n; t \rangle$-algorithms. Then our algorithm has the lowest arithmetic cost among all commutative algorithms for computing sub-blocks of dimensions $\langle n, n, n \rangle$ up to a low order term if the outer algorithm has duplicate rows.*

Many algorithms have duplicate rows (cf. [7, 24, 25, 37]), unfortunetly Strassen's algorithm is not one of them. Figure 1.1 compares the cost of using the classic algorithm, Winograd's, and ours for multiplying 20×20 blocks. The two sides differ in their outer algorithm. The left-hand graphs use one of the $\langle 3, 3, 3; 23 \rangle$-algorithms (out of over 17000) that [24] found, denoted there "i106w191c23ci-000". The right-hand graphs use the $\langle 6, 6, 6; 160 \rangle$-algorithm obtained by composing the classical $\langle 2, 1, 2; 4 \rangle$ with the $\langle 3, 6, 3; 40 \rangle$-algorithm of [43]. Our algorithm performs better than the classical one on 20×20 sub-blocks when the matrices are sufficiently large (see Figure 1.1).

Our algorithm improves the leading coefficient of the total arithmetic cost (recursive algorithm and block multiplications combined) while preserving the communication costs up to a low order term (see Table 2). To this end, we utilize duplicate rows in the encoding matrices of the outer algorithm to reduce the costs of computing the correction terms (multiplications that depend on only one of the inputs). We generalize the trilinear condition of [11] that characterizes non-commutative algorithms to commutative ones.

1.4 Paper organization

In Section 2, we recall some preliminaries regarding quadratic and recursive-bilinear algorithms. We also generalize Brent's [11] triple product condition to fit the commutative case. In Section 3, we provide a composition of non-commutative algorithms with commutative ones, that allows better arithmetic cost than the standard composition. In Section 4, we show how to utilize the theoretical results of Section 3 to obtain faster algorithms. Section 5 contains discussion and future work.

2 PRELIMINARIES

2.1 Notations

Notation 2.1. Let R be a ring and let $A, B \in R^k$, then $A \odot B$ denotes their Hadamard (elementwise) product.

Notation 2.2. Let R be a ring, let $k, l, n \in \mathbb{N}$ and let $A \in R^n$, $B \in R^l$, then $A \otimes B$ denotes their Kronecker product, and $A^{\otimes k} := \underbrace{A \otimes \ldots \otimes A}_{k \text{ times}}$ is the k^{th} Kronecker power of A.

Notation 2.3. Let R be a ring and let $A \in R^{n \times m}$ be a matrix. The vectorization of A is $\vec{A} = \left(a_{1,1} \ldots, a_{1,m}, \ldots, a_{n,1}, \ldots, a_{n,m}\right)^\top$.

Notation 2.4. Let R be a ring, let $a, b, x, y \in \mathbb{N}$. Let $A \in R^{ax \cdot by}$ be a vector, and let $\forall i \in [ab] : A^{(i)} := \left(A_{(xy)(i-1)+1}, \ldots, A_{xyi}\right)$. The block segmentation of A is $\hat{A} := \left(A^{(1)}, \ldots, A^{(ab)}\right) \in (R^{xy})^{ab}$.

2.2 Encoding and decoding matrices

Fact 2.5 ([46]). *Let $f : R^n \to R^m$ be a quadratic function over a ring R and let ALG be an arithmetic algorithm that uses t multiplications and divisions. There exist $U, V \in R^{t \times n}, W \in R^{t \times m}$ such that*
$\forall a \in R^n : f(a) = W^\top ((U \cdot a) \odot (V \cdot a))$.

That is, for any arithmetic algorithm ALG, there exists a quadratic algorithm ALG′ that does not use division, where the number of its multiplications is the same as the total number of multiplications and divisions as ALG. From here on we assume, without loss of generality, that any algorithm is of the form stated in Fact 2.5.

Definition 2.6. We refer to the $\langle U, V, W \rangle$ of a quadratic algorithm to compute a quadratic function as its encoding/decoding matrices (U and V are the encoding matrices and W is the decoding matrix).

Each row-triplet (one from U, one from V and one from W) represents one multiplication of the algorithm. Each column corresponds to one element (U, V correspond to the input elements and W corresponds to the output). For example, if $U_{1,7} = 3$ then the algorithm adds $3 \cdot a_7$ to the left-hand operand of the first multiplication.

Remark 2.7. Permuting the rows of $\langle U, V, W \rangle$ changes only the computation order, so it yields an equivalent algorithm.

Remark 2.8. Note that any bilinear function with inputs A, B is a quadratic function with the input $a = (A, B)$.

Remark 2.9. Denote $U = \left(U^A, U^B\right), V = \left(V^A, V^B\right)$. If the algorithm is non-commutative then $U^B, V^A = 0$. Therefore, the common notation is to ignore them and only specify $\left\langle U^A, V^B, W \right\rangle$.

Example 2.10. Let us look at the classic $\langle 2, 1, 2; 4 \rangle$-algorithm:

- Its representation using the non-commutative notation is

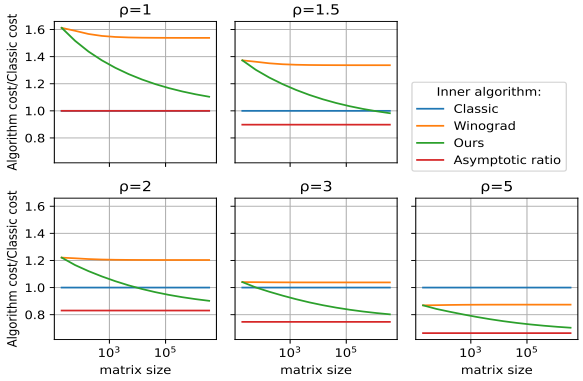
$$\langle U, V, W \rangle = \left\langle \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right\rangle$$

Case study: The arithmetic costs of a 2×2 block multiplication using various algorithms and corresponding lower bounds. The costs for computing a single block multiplication are amortized among all the block multiplications inside a large matrix multiplication. The costs include any computation except for the cost of the fast recursive algorithm that utilizes block multiplications at the base of the recursion. The $o(1)$ refers to an additional cost that is sublinear in the number of block multiplications. The assumptions column details the requirements and conditions for the algorithm or the lower bound.[1] We make three algorithmic assumptions: (1) the algorithm works as part of a large matrix multiplication algorithm, (2) scalar multiplication is commutative (namely, $a \cdot b = b \cdot a$), and (3) both encoding matrices (Definition 2.6) of the outer algorithm contain duplicate rows. For the lower bounds, we require (4) a homogenous algorithm. That is, the matrix multiplication algorithm uses the same inner algorithm for all block multiplications, and every main scalar multiplication appears in only one block multiplication. Our lower bounds allow assumptions (1) to (3) above but do not require them.

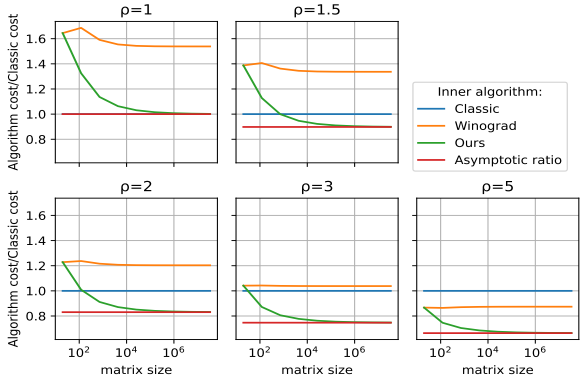
[1]The lower bounds in this paper apply to *arithmetic algorithms* [18], namely data-oblivious algorithms that use only the four operations +, −, ×, ÷.

Table 1: Amortized cost per 2×2 block multiplication

		Paper	×	+	Assumptions
Non-commutative	Algorithm	Classic	8	4	-
		[45]	7	18	-
		[52]	7	15	-
		[31]	7	$12 + o(1)$	Part of a recursive algorithm (1)
	Lower bound	[25]	7		Over $\mathbb{F}_2$, not part of a recursive algorithm
		[51]	7		Over a general field, not part of a recursive algorithm
		[39]		15	7 multiplications over $\mathbb{F}_2$, not part of a recursive algorithm
		[13]		15	7 multiplications over a general field, not part of a recursive algorithm
		[31]		12	7 multiplications
Commutative (2)	Algorithm	[49]	8	16	-
		[47]	7	23	-
		Ours	$4 + o(1)$	$8 + o(1)$	Part of a recursive algorithm (1), outer algorithm with duplicate rows (3)
	Lower bound	[51]	7		Over a general field, not part of a recursive algorithm
		Ours	4		Homogenous algorithm (4)
		Ours		8	4 multiplications



(a) Composing an outer $\langle 3, 3, 3; 23 \rangle$-algorithm with different inner algorithms



(b) Composing an outer $\langle 6, 6, 6; 160 \rangle$-algorithm with different inner algorithms

Figure 1.1: Case study: Simulation of the total arithmetic cost of composing an outer fast matrix multiplication algorithm with inner block multiplication.

Each point in the graph corresponds to the exact arithmetic count, including both the outer and the inner algorithms, for a given matrix dimensions. The inner algorithms (Classic/Winograd's/ours) are applied to blocks of size 20×20. The outer fast algorithm are: $\langle 3, 3, 3; 23 \rangle$-algorithm (left) from [24] and $\langle 6, 6, 6; 160 \rangle$-algorithm (right) made of the classical $\langle 2, 1, 2; 4 \rangle$ composed with $\langle 3, 6, 3; 40 \rangle$ from [43].[2] The x-axis represents the total size of the matrices multiplied. The y-axis represents the total arithmetic cost normalized by the arithmetic cost when applying the classical algorithm to the 20×20 blocks. ρ is the cost ratio between multiplication and addition.

[2]We apply the base transformation technique of [31] on all outer algorithms to improve their parameters. This change does not affect the point from which using our algorithm improves on using the classical one. This application will be presented in the full version of this paper.

ρ	1	1.5	2	3	5	50
$\langle 3,3,3;23\rangle$	1	0.907	0.843	0.761	0.677	0.521
$\langle 6,6,6;160\rangle$	1	0.910	0.847	0.766	0.681	0.522

Table 2: The ratio between the leading coefficient of total arithmetic cost (recursive algorithm and block multiplications, combined) when using our algorithm vs. the classical algorithm, using the same parameters as in Figure 1.1.

- Its representation using the general notation is $\langle U,V,W\rangle =$

$$\left\langle \left(\begin{array}{cc|cc} 1&0&0&0\\1&0&0&0\\0&1&0&0\\0&1&0&0 \end{array}\right), \left(\begin{array}{cc|cc} 0&0&1&0\\0&0&0&1\\0&0&1&0\\0&0&0&1 \end{array}\right), \left(\begin{array}{cccc} 1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1 \end{array}\right) \right\rangle$$

Definition 2.11. Denote the number of non-zero entries in a matrix A by $\text{nnz}(A) = \left|\left\{(i,j) : A_{i,j} \neq 0\right\}\right|$, and denote the number of non-singular entries in A by $\text{nns}(A) = \left|\left\{(i,j) : A_{i,j} \notin \{-1,0,1\}\right\}\right|$.

Claim 2.12 ([31]). Let $\langle U,V,W\rangle$ be the encoding/decoding matrices of a bilinear function $f : R^n \times R^m \to R^k$ and let q_u, q_v, q_w be the number of arithmetic operations performed by the encoding and decoding matrices, correspondingly. Then

$$\begin{aligned} q_u &= \text{nnz}(U) + \text{nns}(U) - t \\ q_v &= \text{nnz}(V) + \text{nns}(V) - t \\ q_w &= \text{nnz}(W) + \text{nns}(W) - k \end{aligned}$$

Claim 2.13 (Triple product condition). [11] Let R be a ring, let $U \in R^{t\times xy}, V \in R^{t\times yz}, W \in R^{t\times zx}$, and let $\delta_{i,j}$ be Kronecker's delta. $\langle U,V,W\rangle$ are encoding/decoding matrices of an $\langle x,y,z;t\rangle$-algorithm if and only if for every $i_1,i_2 \in [x], j_1,j_2 \in [y], k_1,k_2 \in [z]$ it holds that $\sum_{s=1}^{t} U_{s,(i_1,j_1)} V_{s,(j_2,k_1)} W_{s,(i_2,k_2)} = \delta_{i_1,i_2}\cdot\delta_{j_1,j_2}\cdot\delta_{k_1,k_2}$

We generalize the Claim 2.13 to fit the commutative case. Namely we provide a necessary and sufficient condition for a quadratic algorithm $\langle U,V,W\rangle$ to be commutative matrix multiplication algorithm:

Claim 2.14. Let R be a commutative ring, and let $U,V \in R^{t\times(x\cdot y+y\cdot z)}$, $W \in R^{t\times z\cdot x}$. Then $\langle U,V,W\rangle$ are encoding/decoding matrices of a commutative matrix multiplication algorithm of dimensions $\langle x,y,z\rangle$ if and only if for every $i_1,i_2,i_3 \in [x], j_1,j_2 \in [y], k_1,k_2,k_3 \in [z]$

$$\begin{gathered} \sum_{s=1}^{t} \left(U^A_{s,(i_1,j_1)} V^B_{s,(j_2,k_1)} + U^B_{s,(j_2,k_1)} V^A_{s,(i_1,j_1)}\right) W_{s,(i_2,k_2)} \\ = \delta_{i_1,i_2}\cdot\delta_{j_1,j_2}\cdot\delta_{k_1,k_2} \\ \sum_{s=1}^{t} U^A_{s,(i_1,j_1)} V^A_{s,(i_2,j_2)} W_{s,(i_3,k_1)} = 0 \\ \sum_{s=1}^{t} U^B_{s,(j_1,k_1)} V^B_{s,(j_2,k_2)} W_{s,(i_1,k_3)} = 0 \end{gathered}$$

The proof follows the proof of Brent's triple product condition ([11]). It will appear in the full version of this paper.

2.3 Algorithm composition

Fact 2.15 (Mixed product property). *[35] Let R be a ring and let $A \in R^{m\times n}, B \in R^{n\times p}, C \in R^{x\times y}, D \in R^{y\times z}$ be matrices over R. Then $(A\otimes C)\cdot(B\otimes D) = (A\cdot B)\otimes(C\cdot D)$.*

Fact 2.16. *Let $\left\langle U^O,V^O,W^O\right\rangle$ be an $\langle a,b,c;t\rangle$-algorithm, and let $\left\langle U^I,V^I,W^I\right\rangle$ be an $\langle x,y,z;t'\rangle$-algorithm. Then the standard composition of $\left\langle U^O,V^O,W^O\right\rangle$ with $\left\langle U^I,V^I,W^I\right\rangle$ is*

$$\left\langle U^O,V^O,W^O\right\rangle \otimes \left\langle U^I,V^I,W^I\right\rangle = \left\langle U^O\otimes U^I, V^O\otimes V^I, W^O\otimes W^I\right\rangle$$

Remark 2.17. The equality in Fact 2.16 is algebraic, but the computation graphs are different. This difference affects the arithmetic cost as well as the communication complexity.

Corollary 2.18. *Let $\langle U,V,W\rangle$ be an $\langle x,y,z;t\rangle$-algorithm. Calling $\langle U,V,W\rangle$ recursively k times yields the $\left\langle x^k,y^k,z^k;t^k\right\rangle$-algorithm $\langle U,V,W\rangle^{\otimes k}$ with the encoding/decoding matrices $\left\langle U^{\otimes k},V^{\otimes k},W^{\otimes k}\right\rangle$.*

High performance fast recursive matrix multiplication implementations operate according to the pseudo-code in Algorithm 1. They generally use the classic algorithm for ALG_I.

Algorithm 1 Standard Composition

Input $A \in R^{a^k x\times b^k y}, B \in R^{b^k y\times c^k z}, k \in \mathbb{N}$
Output $C = A\cdot B \in R^{a^k x\times c^k z}$

```
function Composed_algorithms(A, B, k)
    if k = 0 then
        return ALG_I(A, B)
    else
        Ã = U^O · Â                ▷ transform the input (Notation 2.4)
        B̃ = V^O · B̂
        for i = 1, ..., t do
            C̃_i = Composed_algorithms(Ã_i, B̃_i, k − 1)
        return (W^O)^⊤ · C̃         ▷ transform the output
```

This pseudo-code is equivalent to composing $\left\langle U^O,V^O,W^O\right\rangle^{\otimes k}$ with ALG_I where $\left\langle U^O,V^O,W^O\right\rangle$ is an $\langle a,b,c;t\rangle$-algorithm and ALG_I is an algorithm of dimentions $\langle x,y,z\rangle$.

Claim 2.19 ([6]). Let $\langle U,V,W\rangle$ be the encoding/decoding matrices of an $\langle x,y,z;t\rangle$ and let q_u,q_v,q_w be the number of arithmetic operations performed by the encoding and decoding matrices, correspondingly. Let $k \in \mathbb{N}$ and denote $X = x^k, Y = y^k, Z = z^k$. Then the number of arithmetic operations $U^{\otimes k}$ performs is $\frac{q_u}{t-xy}\left(t^k - XY\right)$ and similarly $V^{\otimes k}$ performs $\frac{q_v}{t-yz}\left(t^k - YZ\right)$ operations and $W^{\otimes k}$ performs $\frac{q_w}{t-xz}\left(t^k - XZ\right)$ operations.

2.4 Communication models

We use two communication models, the sequential model and the parallel model, as in [3, 29]:

Definition 2.20 (The sequential model)**.** [29] The sequential model is a communication model of a two-level machine - a fast memory of size M and an unbounded slow memory. The I/O-complexity is the number of words moved between the levels.

Definition 2.21. Let ALG be an algorithm whose I/O-complexity depends on the parameters $x_1,\ldots,x_n$. We denote its I/O-complexity by $IO_{\text{ALG}}(x_1,\ldots,x_n)$.

Definition 2.22 (The parallel model)**.** [3] The parallel model is a communication model of a distributed-memory machine with P identical processors, each with a local memory of size M. The I/O-complexity is the number of words sent and received along the cririca path as defined in [53].

Definition 2.23. Let ALG be an algorithm whose I/O-complexity depends on the parameters $x_1, \ldots, x_n$. We denote its I/O-complexity by $IO_{\text{ALG}}(x_1, \ldots, x_n)$.

Remark 2.24. The model (parallel or sequential) is always clear from the context. The number of processors P appears only in the parallel model.

Example 2.25. Let CLS be the classical algorithm. Then $IO_{CLS}(M, x)$ is the I/O-complexity of computing the classical algorithm in the sequential model for square matrices of dimesions $x \times x$ using fast memory of size M. $IO_{CLS}(P, M, x)$ is the I/O-complexity of computing the classical algorithm in the parallel model for square matrices of dimesions $x \times x$ using P processors each with local memory of size M.

3 MAIN RESULTS

This section contains a new way to use commutative algorithms for the innermost layer of a recursive bilinear algorithm. We also show a tight lower bound on the number of multiplications and the number of arithmetic operations commutative algorithms require. For the sake of simplicity, we analyze only the square matrices case. The rectanglular case follows similarly.

To obtain the new algorithmic method, we utilize duplicate rows in the encoding matrices of the outer algorithm. With duplicate rows, blocks from A and B appear in multiple block multiplications. We then reuse correction terms as they depend on only one of the inputs. That is, we compute the correction terms only once.

3.1 Improved composition

We next show how to compose a non-commutative algorithm with a commutative one, in a more efficient way, compared to the standard composition in Fact 2.16. We split the multiplications of each commutative algorithm into two types: main multiplications and correction terms. We reuse correction terms in multiple block multiplications in order to save arithmetic operations.

3.1.1 Main intuition. Our algorithm generalizes Winograd's algorithm ([49]). We start with describing Winograd's construction for multiplying $n \times n$ matrices.

Theorem 3.1 ([49]). *Let n be an even integer. There is a commutative algorithm that multiplies two $n \times n$ matrices using $\frac{n^3}{2} + n^2$ multiplications and $\frac{3n^3}{2} + 2n^2 - n$ additions.*

Proof idea. The algorithm is obtained by folding the classic algorithm. That is, it uses the identity $a_1b_1 + a_2b_2 = (a_1 + b_2)(a_2 + b_1) - a_1a_2 - b_1b_2$. For every $j \in [n]$, the classic algorithm contains the computation $a_{11}b_{1j} + a_{12}b_{2j}$. Using the folding identity, each of these computations uses the multiplication $a_{11}a_{12}$, so instead of computing it n times, one can compute it once. The same idea also works for the multiplications that use only elements from B. □

Note that the classic algorithm is a non-commutative one. We generalize Winograd's folding technique to general non-commutative algorithms. Let $\langle U, V, W \rangle$ be an $\langle m, m, m; t \rangle$-algorithm such that U has d_u distinct rows and V has d_v distinct rows where $d_u, d_v < t$. Consider an algorithm ALG that has k recursive calls of $\langle U, V, W \rangle$, and then a single call to Winogard's algorithm (Figure 3.1(b)). This algorithm incurs considerable overhead compared to the classic algorithm (Figure 3.1(a)) for small blocks.

As U has only d_u distinct rows, the combined algorithm multiplies the same sub-block of A by several sub-blocks of B. We can therefore improve the algorithm by computing correction terms that depend on A only just once. This improvement reduces the overhead significantly (see Figures 3.1(c) and 3.1(d).)

3.1.2 Formal description. First, we generalize the uniting-of-terms tecnique from [37] to reduce the number of multiplications the algorithm uses. Then we show how to compose a non-commutative algorithm with a commutative one. Finally, we combine the two methods to obtain algorithms that use fewer arithmetic operations.

Claim 3.1 ([37]). Let $\langle U, V, W \rangle$ be a bilinear algorithm that computes $f : R^n \times R^m \to R^k$, using t multiplications. If the last two rows of U and V are duplicate, namely $U_t = U_{t-1}$ and $V_t = V_{t-1}$, then the tensor $\langle U', V', W' \rangle = \left\langle \begin{pmatrix} U_1 \\ \vdots \\ U_{t-1} \end{pmatrix}, \begin{pmatrix} V_1 \\ \vdots \\ V_{t-1} \end{pmatrix}, \begin{pmatrix} W_1 \\ \vdots \\ W_{t-1} + W_t \end{pmatrix} \right\rangle$ is equivalent to $\langle U, V, W \rangle$.

We generalize the previous claim to the commutative case and provide its effect on the number of linear operations.

Lemma 3.2. *Let $\langle U, V, W \rangle$ be a commutative algorithm that computes $f : R^n \to R^k$, using t multiplications and q_u, q_v, q_w linear operations in U, V, W respectively. If the last two rows of U and V are duplicate, namely $U_t = U_{t-1}$ and $V_t = V_{t-1}$, then the algorithm $\langle U', V', W' \rangle = \left\langle \begin{pmatrix} U_1 \\ \vdots \\ U_{t-1} \end{pmatrix}, \begin{pmatrix} V_1 \\ \vdots \\ V_{t-1} \end{pmatrix}, \begin{pmatrix} W_1 \\ \vdots \\ W_{t-1} + W_t \end{pmatrix} \right\rangle$ is equivalent to $\langle U, V, W \rangle$. Furthermore, $q'_u \le q_u, q'_v \le q_v, q'_w \le q_w$.*

Proof. The proof of the equivalence is identical to the proof from [37], so we show only the second part. Deleting a row from U reduces $nnz(U)$ without increasing $nns(U)$, thus by Claim 2.12

$$q'_u = nnz(U') + nns(U') - (t-1) \le nnz(U) + nns(U) - 1 - t + 1 = q_u$$

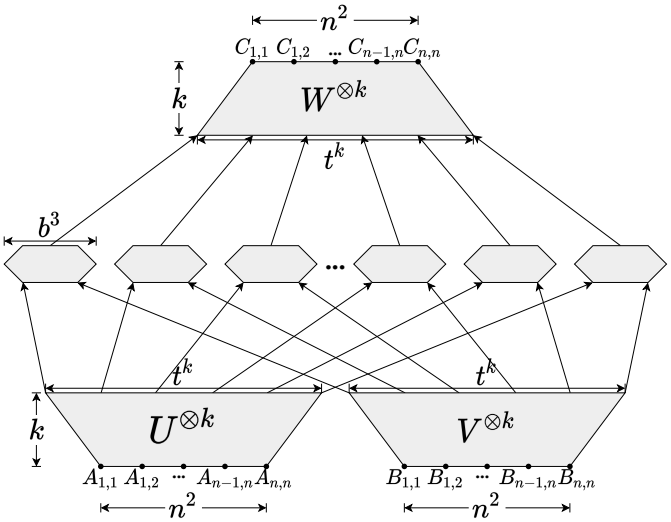
The same argument shows that $q'_v \le q_v$. As for W, merging rows does not increase $nnz(W) + nns(W)$, therefore

$$q'_w = nnz(W') + nns(W') - k \le nnz(W) + nns(W) - k = q_w \quad \square$$

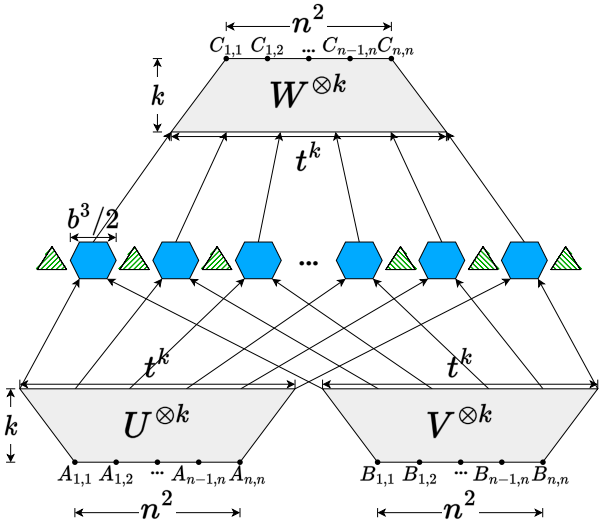
Note that we can view any commutative algorithm to compute bilinear function as 3 algorithms: ALG_A, ALG_B and ALG_{AB}. ALG_A dependent only on the first part of the input, ALG_B dependent only on the second part of the input, these are the correction terms, and the main multiplications ALG_{AB} dependent on both. After computing the three algorithms, we need only to combine them.

Definition 3.3. Let R be a commutative ring. We use the notation $\langle x, y, z; t_{AB}, t_A, t_B \rangle^C$ to represent a commutative matrix multiplication algorithm of the dimensions $\langle x, y, z \rangle$ that uses $t_{AB} + t_A + t_B$

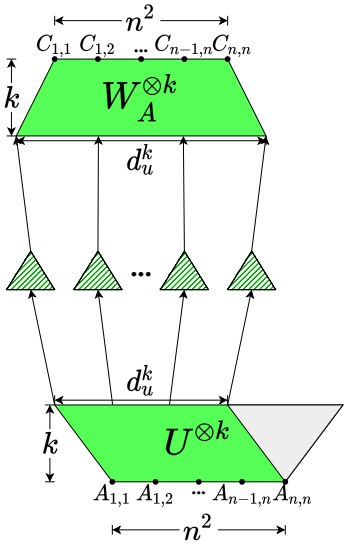
(a) Existing hybrid fast matrix multiplication, switching to the classical algorithm on small blocks of size $b \times b$. The hexagons represent the classical algorithm.



(b) Winograds algorithm. The hexagons stand for the folded classical algorithm, each using $\frac{b^3}{2}$ multiplications. The triangles stand for the correction terms, namely the $\frac{b^2}{2}$ multiplications of the form $a_i \cdot a_j$ and the $\frac{b^2}{2}$ of the form $b_i \cdot b_j$. See Theorem 3.1.



(c) COAT's correction terms. Each triangle uses $\frac{b^2}{2}$ multiplications and returns a vector of length b.



(d) COAT (Commutative Optimal Algorithm for Tiny blocks)

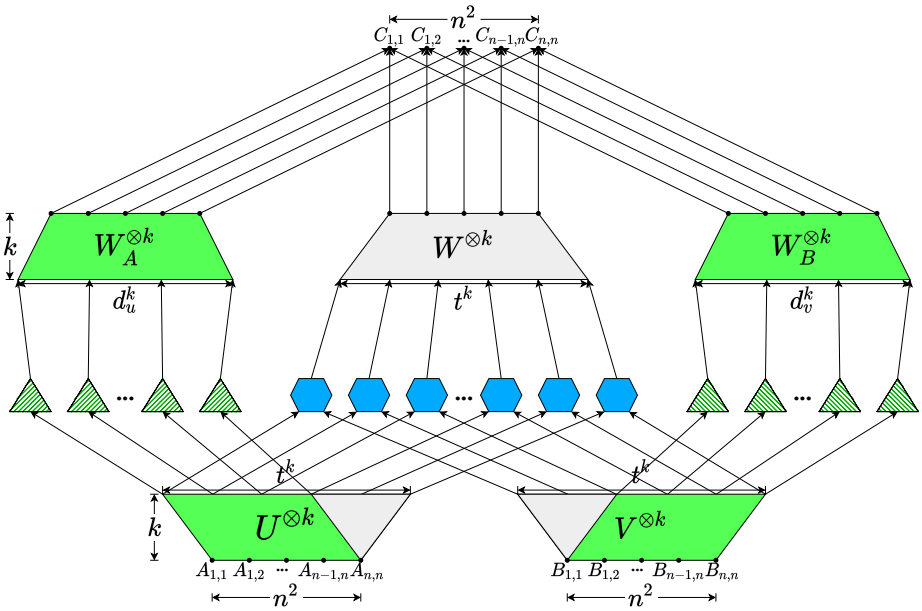


Figure 3.1: The computation graph of multiplying two $n \times n$ matrices using a fast recursive algorithm, then switching to $b \times b$ blocks multiplication using one of three base case algorithms: (a) the classic one, (b) Wingrad's, and our algorithm (c,d).

multiplications where ALG_A uses t_A of them, ALG_B uses t_B of them, and ALG_{AB} uses the remaining t_{AB} multiplications.

We split the linear operations similarly to $s = s_{AB} + s_A + s_B + s_M$, where ALG_{AB} uses s_{AB} of them, s_A, s_B are defined similarly, and s_M is the number of additions needed to combine the three algorithms.

Example 3.4. Consider the following $\langle 1, 2, 1; 1, 1, 1\rangle^C$-algorithm:

$$\begin{aligned} M_1 &= (a_1 + b_2)(a_2 + b_1) \\ M_2 &= a_1 a_2 \\ M_3 &= b_1 b_2 \\ c &= M_1 - M_2 - M_3 \end{aligned}$$

M_1 is ALG_{AB}, M_2 is ALG_A, and M_3 is ALG_B. The last row combines the three algorithms. We can also describe it by

$$\langle U,V,W\rangle = \left\langle \left(\begin{array}{cc|cc} 1&0&0&1\\ 1&0&0&0\\ \hline 0&0&0&1 \end{array}\right), \left(\begin{array}{cc|cc} 0&1&1&0\\ 0&1&0&0\\ \hline 0&0&1&0 \end{array}\right), \begin{pmatrix}1\\-1\\-1\end{pmatrix} \right\rangle$$

In this algorithm, $s_{AB}=2$, $s_A=s_B=0$, and $s_M=2$. Note that [49] uses this algorithm as a building block (see Theorem 3.1 above).

Definition 3.5. Let $\left\langle U^I,V^I,W^I\right\rangle$ be an $\langle x,y,z;t_{AB},t_A,t_B\rangle^C$-algorithm, and let $\left\langle U^O,V^O,W^O\right\rangle$ be an $\langle a,b,c;t\rangle$-algorithm. Then the standard composition of $\left\langle U^O,V^O,W^O\right\rangle$ with $\left\langle U^I,V^I,W^I\right\rangle$ is

$$\left\langle U^O,V^O,W^O\right\rangle \circledast \left\langle U^I,V^I,W^I\right\rangle :=$$
$$\left\langle \left(U^O\otimes U^I_A, V^O\otimes U^I_B\right), \left(U^O\otimes V^I_A, V^O\otimes V^I_B\right), W^O\otimes W^I\right\rangle$$

This composition is a generalization of the one in [27, 28]. [27] uses this composition with [52] as the outer algorithm and [49] as the inner one. [28] uses this composition with [33] as the outer algorithm and [49] as the inner one.

Remark 3.6. The intuitive composition $\left\langle U^O\otimes U^I, V^O\otimes V^I, W^O\otimes W^I\right\rangle$ does not work. The width of the encoding matrices of a commutative algorithm is the size of the input - $abxy+bcyz$, but the width of $U^O\otimes U^I$ is $ab\,(xy+yz)$.

Claim 3.7. Let all the parameters be as in Definition 3.5. Then $\left\langle U^O,V^O,W^O\right\rangle \circledast \left\langle U^I,V^I,W^I\right\rangle$ is an $\langle ax,by,cz;t\cdot t_{AB},t\cdot t_A,t\cdot t_B\rangle^C$-algorithm.

Proof idea. U^O operates on A and V^O operates on B, so they affect only the corresponding parts of $\left\langle U^I,V^I,W^I\right\rangle$. The complete proof will appear in the full version of this paper. □

Notation 3.8. Let $\left\langle U^O,V^O,W^O\right\rangle$ be an $\langle a,b,c;t\rangle$-algorithm. Denote by $\langle U',U',W_A\rangle$ the algorithm equivalent to $\left\langle U^O,U^O,W^O\right\rangle$ that Lemma 3.2 provides. We denote by q^A_w and q'_u the number of linear operations used to apply W_A and U' respectively. Similarly define q^B_w and q'_v.

We now state and prove our improved composition (COAT). The pseudo-code appears in Algorithm 2, and its computation graph appears in Figure 3.1. The main difference between COAT and the standard composition (Definition 3.5) is that the latter uses a black-box composition while COAT is more involved.

Theorem 3.2. *Let $\left\langle U^O,V^O,W^O\right\rangle$ be an $\langle m,m,m;t\rangle$-algorithm that uses q linear operations such that U^O has d_u distinct rows and V^O has d_v distinct rows, and let $\left\langle U^I,V^I,W^I\right\rangle$ be a $\langle b,b,b;t_{AB},t_A,t_B\rangle^C$-algorithm that uses $s_{AB}+s_A+s_B+s_M$ linear operations. Let $n=b\cdot m^k$, then Algorithm 2 is an $\left\langle n,n,n;t^k\cdot t_{AB},d_u^k\cdot t_A,d_v^k\cdot t_B\right\rangle^C$-algorithm that uses at most $\left(\frac{qb^2}{t-m^2}+s_{AB}\right)t^k+O\left(d_u^k+d_v^k\right)$ linear operations.*

Proof sketch. We use Claim 3.9 to compose $\left\langle U^O,V^O,W^O\right\rangle^{\otimes k}$ with $\left\langle U^I,V^I,W^I\right\rangle$. The complete proof will appear in the full version of this paper. □

Algorithm 2 Commutative Optimal Algorithm for Tiny blocks (COAT)

Input $A\in R^{a^kx\times b^ky}, B\in R^{b^ky\times c^kz}, k\in\mathbb{N}$
Output $C=A\cdot B\in R^{a^kx\times c^kz}$

function COAT(A,B,k)
  $C=\text{RCA}(A,B,k)$
  $A_F=\text{AFIX}(A,k)$
  $B_F=\text{BFIX}(B,k)$
  return $C+A_F+B_F$

function RCA(A,B,ℓ)
  if $\ell=0$ **then**
    return $\text{ALG}^I_{AB}(A,B)$
  else
    $\tilde{A}=U^O\cdot\hat{A}$ ▷ transform the input (Notation 2.4)
    $\tilde{B}=V^O\cdot\hat{B}$
    for $i=1,\ldots,t$ **do**
      $\tilde{C}_i=\text{RCA}(\tilde{A}_i,\tilde{B}_i,\ell-1)$
    return $\left(W^O\right)^\top\cdot\tilde{C}$ ▷ transform the output

function AFIX(A,ℓ)
  if $\ell=0$ **then**
    return $\text{ALG}^I_A(A)$
  else
    $\tilde{A}=U'\cdot\hat{A}$
    for $i=1,\ldots,t$ **do**
      $A'_i=\text{AFIX}(\tilde{A}_i,\ell-1)$
    return $(W_A)^\top\cdot A'$

This algorithm is our improved composition of an $\langle a,b,c;t\rangle$ algorithm $\left\langle U^O,V^O,W^O\right\rangle$ recursively with an $\langle x,y,z;t_{AB},t_A,t_B\rangle^C$ algorithm that split into $\text{ALG}^I_A,\text{ALG}^I_B$ and ALG^I_{AB}. Define BFIX similarly to AFIX.

Claim 3.9. Let all the parameters be as Theorem 3.2. Then there is an $\langle mb,mb,mb;t\cdot t_{AB},d_u\cdot t_A,d_v\cdot t_B\rangle^C$-algorithm that uses at most $ts_{AB}+d_u\cdot s_A+d_v\cdot s_B+\left(q+2q'_u+2q'_v+q^A_w+q^B_w\right)b^2+m^2s_M$ linear operations.

Proof. Let $\text{ALG}^I_A,\text{ALG}^I_B$ and ALG^I_{AB} be the three parts of the inner algorithm. Then their encoding/decoding matrices correspond to rows from $\left\langle U^I,V^I,W^I\right\rangle$. By Fact 2.5, without loss of generality

$$\left\langle U^I,V^I,W^I\right\rangle = \left\langle \left(U^I_A,U^I_B\right),\left(V^I_A,V^I_B\right),W^I\right\rangle$$
$$= \left\langle \begin{pmatrix} U^I_{MA}&U^I_{MB}\\ U^I_{FA}&0\\ 0&U^I_{FB}\end{pmatrix}, \begin{pmatrix} V^I_{MA}&V^I_{MB}\\ V^I_{FA}&0\\ 0&V^I_{FB}\end{pmatrix}, \begin{pmatrix} W^I_M\\ W^I_{FA}\\ W^I_{FB}\end{pmatrix}\right\rangle$$

where $\left\langle \left(U^I_{MA},U^I_{MB}\right),\left(V^I_{MA},V^I_{MB}\right),W^I_M\right\rangle$ are the encoding/decoding matrices of ALG^I_{AB}, $\left\langle \left(U^I_{FA},0\right),\left(V^I_{MA},0\right),W^I_{FA}\right\rangle$ are the encoding/decoding matrices of ALG^I_A, and $\left\langle \left(0,U^I_{MB}\right),\left(0,V^I_{MB}\right),W^I_{FB}\right\rangle$ are the encoding/decoding matrices of ALG^I_B. Composing $\left\langle U^O,V^O,W^O\right\rangle$

with $\langle U^I, V^I, W^I \rangle$ using Definition 3.5 yields

$$\left\langle \left(U^O \otimes U^I_A, V^O \otimes U^I_B\right), \left(U^O \otimes V^I_A, V^O \otimes V^I_B\right), W^O \otimes W^I \right\rangle$$

Splitting this algorithm into the three algorithms ALG_A, ALG_B and ALG_{AB} in this order gives us:

$$\left\langle \left(U^O \otimes U^I_{FA}, V^O \otimes 0\right), \left(U^O \otimes V^I_{FA}, V^O \otimes 0\right), W^O \otimes W^I_{FA} \right\rangle$$
$$\left\langle \left(U^O \otimes 0, V^O \otimes U^I_{FB}\right), \left(U^O \otimes 0, V^O \otimes V^I_{FB}\right), W^O \otimes W^I_{FB} \right\rangle$$
$$\left\langle \left(U^O \otimes U^I_{MA}, V^O \otimes U^I_{MB}\right), \left(U^O \otimes V^I_{MA}, V^O \otimes V^I_{MB}\right), W^O \otimes W^I_M \right\rangle$$

Note that $\text{ALG}_{AB} = \langle U^O, V^O, W^O \rangle \circledast \text{ALG}^I_{AB}$. Therefore computing ALG_{AB} uses $t \cdot t_{AB}$ multiplications. We can apply U^O and V^O, using $(q_u + q_v)\, b^2$ linear operations, then we apply ALG^I_{AB} to each of the t blocks using s_{AB} linear operations, and apply W^O to them using $q_w b^2$ linear operations. In total ALG_{AB} uses $t \cdot t_{AB}$ multiplications and $qb^2 + ts_{AB}$ linear operations.

ALG_A uses only elements from A, so we can remove the columns that depends on B from the encoding matrices, meaning that

$$\begin{aligned} \text{ALG}_A &= \left\langle U^O \otimes U^I_{FA}, U^O \otimes V^I_{FA}, W^O \otimes W^I_{FA} \right\rangle \\ &= \left\langle U^O, U^O, W^O \right\rangle \otimes \text{ALG}^I_A \end{aligned}$$

By Lemma 3.2 there is an algorithm $\langle U', U', W_A \rangle$ equivalent to $\langle U^O, U^O, W^O \rangle$ that uses only d_u multiplications, and less linear operations than $\langle U^O, U^O, W^O \rangle$. Thus, replacing ALG_A with $\text{ALG}'_A = \langle U', U', W_A \rangle \otimes \text{ALG}^I_A$ gives the same result using fewer operations.

This improvement allows us to compute the result of ALG_A using only $d_u t_A$ multiplications and $\left(2q'_u + q^A_w\right) b^2 + d_u s_B$ linear operations. The same argument shows that computing the result of ALG_B takes $d_v \cdot t_B$ multiplications and at most $\left(2q'_v + q^B_w\right) b^2 + d_v s_B$ linear operations.

Given the results from all three algorithms, we can compute the output using $m^2 s_M$ additions. In total, we used $t \cdot t_{AB} + d_u \cdot t_A + d_v \cdot t_B$ multiplications and at most $ts_{AB} + d_u \cdot s_A + d_v \cdot s_B + m^2 s_M + \left(q + 2q'_u + 2q'_v + q^A_w + q^B_w\right) b^2$ linear operations. □

Claim 3.10. Applying Algorithm 2 with Winograd's $\langle 2, 2, 2; 4, 2, 2 \rangle^C$-algorithm yields an algorithm that uses $4 + o(1)$ scalar multiplications and $8 + o(1)$ additions for each block multiplication.

Proof. By Claim 3.9, Algorithm 2 contains t^k block multiplications and $t_{AB} \cdot t^k + O\left(d^k_u + d^k_v\right)$ scalar multiplications. $t_{AB} = 4$, thus the algorithm uses $4 + o(1)$ scalar multiplications for each block multiplication. By Theorem 3.2, Algorithm 2 uses $s_{AB} t^k + O\left(d^k_u + d^k_v\right)$ linear operations outside the ones that are used for the recursive phase. By Claim 3.9 and Example 3.4, $s_{AB} = 4 \cdot 2 = 8$. Therefore, the algorithm uses $8 + o(1)$ additions for each block multiplication. □

3.2 Lower bounds and the optimality of our technique

In this section, we show that commutative algorithms cannot reduce the total number of operations, and can only save up to half of the multiplications. In Section 3.1 we showed how to obtain commutative algorithms from non-commutative ones by folding the multiplications. In this section, we use the inverse operation to provide lower bounds for commutative algorithms: we unfold commutative multiplications. We next show that such unfolding can only reduce the total number of operations.

Definition 3.11. We say that ALG′ is an unfolding of ALG, or that ALG unfolds to ALG′, if replacing each multiplication of the form $(A_1 + B_2)(A_2 + B_1)$ in ALG with $A_1B_1 + A_2B_2$ results in ALG′.

Theorem 3.3. *Let ALG be an $\langle l, m, n; t_{AB}, t_A, t_B \rangle^C$-algorithm. Let ALG′ be an unfolding of ALG. Then ALG′ is an $\langle l, m, n; 2t_{AB} \rangle$-algorithm that uses no more arithmetic operations than ALG.*

Proof. Recall that the correction terms are multiplications that depend only on one of the matrices. Let $A_1 \cdot A_2$ be a correction term. The unfolding of this multiplication is $A_1 \cdot 0 + A_2 \cdot 0 = 0$. Therefore the correction terms do not affect ALG′. Let us inspect the effect of unfolding on a given multiplication from ALG_{AB}. It is of the form $(A_1 + B_2)(A_2 + B_1)$. We can apply this row without commutativity using $A_1B_1 + A_2B_2$ instead since A_1A_2 and B_1B_2 do not appear in the result. Thus we obtain an $\langle a, b, c; 2t_{AB} \rangle$-algorithm.

Note that if at least one of the four elements A_1, A_2, B_1, B_2 is zero, then $A_1B_1 + A_2B_2$ consists of only one multiplication. In this case, the unfolding saves additions without increasing the number of multiplications. If all of A_1, A_2, B_1, B_2 are non-zeroes, then the unfolding trades one addition for one multiplication. Hence the total amount of arithmetic operations remains unchanged. □

Theorem 3.3 seems to imply that commutative algorithms can not improve the running time relative to non-commuatative algorithms. However, scalar multiplication often costs more than an addition, so the correct lower bound is the one of Corollary 3.12:

Corollary 3.12. *Assume that a scalar multiplication costs as much as ρ additions. Then no commutative algorithm can improve the run time of a non-commutative algorithm that uses t multiplications and q linear operations by a factor larger than $\frac{\frac{\rho+1}{2}t+q}{t\rho+q}$.*

Proof. Let ALG be an $\langle l, m, n; t \rangle$ algorithm that uses q linear operations. Let ALG′ be an $\langle l, m, n; t_{AB}, t_A, t_B \rangle^C$-algorithm that unfolds to ALG. By Theorem 3.3, ALG′ uses at least $t + q$ operations and at most $2t_{AB}$ multiplications. Therefore, ALG′ uses at least $\frac{t}{2}$ multiplications. Since $\rho \geq 1$, the arithmetic cost of computing ALG′ is at least $\rho\frac{t}{2} + t + q - \frac{t}{2} = \frac{\rho+1}{2}t + q$. On the other hand, the arithmetic cost of computing ALG is $t\rho + q$. The lower bound on the costs ratio follows. □

We conclude that if the classic algorithm is the optimal non-commutative algorithm for a specific size, then COAT with Winograd's algorithm ([49]) is the optimal commutative algorithm for sub-blocks of the same size. We call this algorithm *generalized folded algorithm (GFA)*.

Theorem 1.1. *Let $n \in \mathbb{N}$ be an even integer. Assume the classical $\langle n, n, n; n^3 \rangle$-algorithm has the lowest arithmetic cost among all $\langle n, n, n; t \rangle$-algorithms. Then GFA has the lowest arithmetic cost among all commutative algorithm for computing sub-blocks of dimensions $\langle n, n, n \rangle$ up to a low order term if $d_u, d_v < t$ in the outer algorithm.*

Table 3: Arithmetic cost per block

Algorithm	Requirements	Previous composition		COAT(here)	
		Multiplications	Linear operations	Multiplications	Linear operations
Classic		n^3	n^3-n^2	n^3	n^3-n^2
[49]	$n \mid 2$	$\frac{n^3}{2}+n^2$	$\frac{3n^3}{2}+2n^2-2n$	$\frac{n^3}{2}+\frac{d_u^k+d_v^k}{2t^k}n^2$	$\frac{\frac{3n^3}{2}-n^2}{+O\left(\frac{d_u^k+d_v^k}{t^k}n^2\right)}$
[49]	$n \nmid 2$	$\frac{n^3}{2}+\frac{3n^2}{2}-n$	$\frac{3n^3}{2}+\frac{7n^2}{2}-5n$	$\frac{\frac{n^3}{2}+\frac{n^2}{2}+\frac{\left(d_u^k+d_v^k\right)(n-1)^2}{2t^k}}{}$	$\frac{\frac{3n^3}{2}-\frac{3n^2}{2}}{+O\left(\frac{d_u^k+d_v^k}{t^k}n^2\right)}$
[47]	$n \mid 2$	$\frac{n^3}{2}+n^2-\frac{n}{2}$	$\frac{3n^3}{2}+8n^2-\frac{27n}{2}+6$	$\frac{n^3}{2}+n^2-\frac{n}{2}$	$\frac{3n^3}{2}+8n^2-\frac{27n}{2}+6$
[40]	$n \nmid 2$	$\frac{n^3}{2}+n^2-\frac{n}{2}$	$\frac{3n^3}{2}+\frac{27n^2}{2}-\frac{57n}{2}-\frac{9}{2}$	$\frac{n^3}{2}+n^2-2n+\frac{3}{2}+\frac{3\left(d_u^k+d_v^k\right)(n-1)}{4t^k}$	$\frac{3n^3}{2}+\frac{25n^2}{2}-\frac{69n}{2}+\frac{21}{2}+O\left(\frac{d_u^k+d_v^k}{t^k}n^2\right)$
[27]	$n \mid 4$	$\frac{7n^3}{16}+\frac{7n^2}{4}$	$\frac{21n^3}{16}+8n^2-7n$	$\frac{7n^3}{16}+\frac{7\left(d_u^k+d_v^k\right)n^2}{8t^k}$	$\frac{\frac{21n^3}{16}+\frac{11n^2}{4}}{+O\left(\frac{d_u^k+d_v^k}{t^k}n^2\right)}$
[28]	$n \mid 9, n \nmid 2$	$\frac{23n^3}{54}+\frac{23n^2}{6}-\frac{23n}{3}$	$\frac{23n^3}{18}+\frac{119n^2}{6}-\frac{115n}{3}$	$\frac{\frac{23n^3}{54}+\frac{23n^2}{18}+\frac{23\left(d_u^k+d_v^k\right)\left(\frac{n}{3}-1\right)^2}{2t^k}}{}$	$\frac{\frac{23n^3}{18}+\frac{127n^2}{18}}{+O\left(\frac{d_u^k+d_v^k}{t^k}n^2\right)}$

The arithmetic costs of several commutative algorithms when used as the inner algorithms of a recursive algorithm. The outer algorithm has the same parameters as the one in Theorem 3.2. n represents the size of the sub-blocks. We also assume $d_u, d_v < t$

Both [47] and [40] use less multiplications than [49] but their total arithmetic cost is higher. Moreover, COAT does not improve their arithmetic cost as much as [49], meaning [49] is faster in practice. As for [27, 28], they both use a fast algorithm composed with [49].

Proof. Assume that, for a fixed n, the classical algorithm has the lowest arithmetic cost among all $\langle n,n,n;t\rangle$-algorithms. Then the lower bound of Theorem 3.12 is the lowest when applying it to the classical algorithm. Setting $t=n^3$ and $q=n^3-n^2$, we deduce that any commutative algorithm for multiplying $n\times n$ blocks costs at least $\frac{\rho+1}{2}n^3+n^3-n^2=\frac{\rho+3}{2}n^3-n^2$. By Corollary 4.2, the arithmetic cost of multiplying each pair of blocks using GFA is

$$\frac{\frac{\rho n^3t^k}{2}+\left(\frac{3n^3}{2}-n^2\right)t^k+O\left(d_u^k+d_v^k\right)n^2}{t^k}$$
$$=\frac{\rho+3}{2}n^3-n^2+O\left(\frac{d_u^k+d_v^k}{t^k}n^2\right)$$

which, when $d_u, d_v < t$, is only slightly larger than the lower bound of $\frac{\rho+3}{2}n^3-n^2$. □

3.2.1 Tight bounds for 2×2 multiplication. GFA multiplies 2×2 blocks using $4+o(1)$ multiplications and $8+o(1)$ additions. We show matching lower bounds both for multiplications and additions. Note that the existing lower bounds for commutative matrix multiplication implicitly assume that the multiplication is not part of a recursive multiplication of larger matrices. Our bounds do not make this assumption and therefor hold for sub-block multiplication.

We only assume that the algorithm is homogenous, namely, that the same algorithm is used for all block multiplications, and that every main multiplication impacts only one block multiplication.

Theorem 3.4 ([51]). *Every algorithm for multiplying two 2×2 matrices requires at least 7 multiplications.*

Theorem 3.5. *Every homogenous commutative algorithm for multiplying 2×2 blocks requires at least 4 scalar multiplications.*

Proof. By Theorems 3.4 and 3.3, any 2×2 block multiplication algorithm requires at least 3.5 multiplications. Each block multiplication uses the same number of multiplications, so the amortized cost of all the block multiplications must be integral. The lower bound follows. □

Given this tight lower bound on the number of multiplications, we can prove a lower bound on the number of linear operations.

Claim 3.13. Every commutative algorithm for multiplying 2×2 blocks must use at least 12 arithmetic operations

Theorem 3.6 ([31]). *Any non-commutative algorithm that multiplies 2×2 blocks using 7 multiplications requires at least 12 linear operations.*

Proof of Claim 3.13. By Theorem 3.3, it suffices to show that Claim 3.13 holds for non-commutative algorithms. Let ALG $=\langle U,V,W\rangle$ be a non-commutative algorithm to multiply 2×2 blocks using t multiplications. If $t=7$, then by Theorem 3.6, ALG uses at least $12+7=19$ arithmetic operations.

Assume that $t\geq8$. Each row of W has a non-zero, so there are at least t non-zeros in W. W has 4 columns, so applying it takes

at least $t-4$ linear operations. We conclude that ALG requires at least $t+t-4 \geq 2 \cdot 8 - 4 = 12$ arithmetic operations. □

4 APPLICATIONS

In this section, we show how to combine the results from Section 3.1 with existing algorithms in order to obtain faster algorithms.

4.1 General idea

Recall the $\langle 1,2,1;1,1,1\rangle^C$-algorithm we saw in Example 3.4:

$$\text{ALG} = \langle U,V,W\rangle = \left\langle \left(\begin{array}{cc|cc} 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right), \left(\begin{array}{cc|cc} 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \end{array}\right), \begin{pmatrix} 1 \\ -1 \\ -1 \end{pmatrix} \right\rangle$$

ALG is the primary building block in Winograd's algorithm [49]. In fact, for any $x,y,z \in \mathbb{N}$, Winograd's $\left\langle x,y,z;\frac{xyz}{2},\frac{xy}{2},\frac{yz}{2}\right\rangle^C$ algorithm is a modification on composing the classic $\left\langle x,\frac{y}{2},z;\frac{xyz}{2}\right\rangle$ algorithm with ALG using the results from Section 3.1. The algorithm we analyze during this section uses ALG in a similar way. However, we compose a general $\langle x,y,z;t\rangle$ algorithm with ALG.

4.2 Arithmetic complexity

In this section, we analyze the arithmetic and communication costs of GFA. We focus on GFA since, by Theorem 1.1, COAT is optimal when applied to Winograd's algorithm.

Claim 4.1 ([49]). Let $b \in \mathbb{N}$ be even. There is an $\left\langle b,b,b;\frac{b^3}{2},\frac{b^2}{2},\frac{b^2}{2}\right\rangle^C$-algorithm that uses $\frac{3}{2}b^3+2b^2-2b$ linear operations.

Corollary 4.2 (GFA). *Let $b,k \in \mathbb{N}$ where b is even and let $\langle U,V,W\rangle$ be an $\langle m,m,m;t\rangle$-algorithm that uses q linear operations. Denote $n=bm^k$ and assume that U,V have d_u,d_v distinct rows respectively. Then allpying GFA gives an $\left\langle n,n,n;\frac{b^3}{2}t^k,\frac{b^2}{2}d_u^k,\frac{b^2}{2}d_v^k\right\rangle^C$-algorithm that uses $\left(\frac{qb^2}{t-m^2}+\frac{3}{2}b^3-b^2\right)t^k+O(d_u^k+d_v^k)$ linear operations.*

Proof. Immidiate from Theorem 3.2 and Claim 4.1. □

Remark 4.3. Composing the same outer algorithm with the classic $\left\langle b,b,b;x^3\right\rangle$-algorithm, instead of GFA, results in an $\left\langle n,n,n;b^3t^k\right\rangle$-algorithm that uses $\left(\frac{qb^2}{t-m^2}+b^3-b^2\right)t^k+O(n^2)$ linear operations.

4.3 Communication costs

This section shows that the communication cost of the GFA algorithm is the same, up to a low order term, as the HYB algorithm, (the conventional hybrid fast matrix multiplication algorithm switching to the classical one at the same cutoff point). This holds both in the sequential and the parallel models. To this end, we use a reduction from the main multiplications of GFA to the classic algorithm. We then prove that the correction phase uses asymptotically less communication (smaller by a factor of $\frac{d_u^k+d_v^k}{t^k}$) [1].

In both models (sequential and parallel), we follow Algorithm 2. We first compute the main multiplications, then the correction terms, and finally combine the results. We assume that the parameters are the same as in Corollary 4.2. Namely, $b,k \in \mathbb{N}$ where the block size b is even, $\langle U,V,W\rangle$ is an $\langle m,m,m;t\rangle$-algorithm, and U,V have d_u,d_v distinct rows respectively. We start by analyzing the parallel model. The sequential model follows easily.

4.3.1 Parallel model. In order to minimize the interprocessor communication costs of GFA, we use two different parallelization methods. For RCA we use the same parallelization technique as the one in HYB to keep the I/O-complexity from changing. As for AFIX and BFIX, we use the BFS-DFS parallelization technique from [3] to give an upper bound on their I/O-complexity.

Theorem 4.1.

$$IO_{GFA}\left(P,M,m^k,b\right) = (1+o(1))\, IO_{HYB}\left(P,M,m^k,b\right)$$

To prove Theorem 4.1, we analize its three components seperately (recall Algorithm 2). We first show in Lemma 4.4 that the I/O-complexity of RCA is the same as that of HYB. We then prove in Lemma 4.6 that AFIX and BFIX use asymptotically less communication.

Lemma 4.4. *Let MAIN be the main multiplications in one block of GFA. If $IO_{CLS}(P,M,b) = \alpha\frac{b^3}{P\sqrt{M}}+\beta\left(\frac{b^3}{P}\right)^{\frac{2}{3}}+\gamma b^2+\delta M$. Then $IO_{MAIN}(P,M,b) \leq IO_{CLS}(P,M,b)$.*

Proof. We treat A, B, and C as if each block of size 2×2 is a single element. This reduces x by a factor of 2, M by a factor of 4, and increases the cost of sending an element by a factor of 4. Now we use the classical algorithm and obtain

$$\begin{aligned} IO_{\text{MAIN}}(P,M,b) &\leq 4\cdot IO_{\text{CLS}}\left(\frac{M}{4},P,\frac{b}{2}\right) \\ &\leq 4\left(\alpha\frac{b^3}{4P\sqrt{M}}+\beta\left(\frac{b^3}{8P}\right)^{\frac{2}{3}}+\gamma\frac{b^2}{4}+\delta\frac{M}{4}\right) \\ &= IO_{\text{CLS}}(P,M,b) \end{aligned}$$

□

Lemma 4.5. $IO_{AFIX}(M,P,1,b) = O\left(\frac{b^2}{P}\right)$

Proof. Note that the correction terms of A are $\mu = (A^e \odot A^o)\cdot \mathbf{1}$ where $\mathbf{1}$ is the all ones vector. First, reorgenize the elements of A such that each processor has $\frac{P}{b}$ rows. Then, each processor computes locally the corresponding elements from μ. Finally, if some rows are split, the processors sum their data.

The first step requires at most $O\left(\frac{b^2}{P}\right)$ communication. The second step does not require communication. The third step uses at most $O\left(\frac{b}{P}\right)$ communication since there are b elements μ (it is a vector). In total, this algorithm used $O\left(\frac{b^2}{P}\right)$ communication. □

We next show an upper bound on the I/O-complexity of AFIX, the same analysis shows that BFIX also uses $o\left(IO_{\text{HYB}}\left(P,M,m^k,b\right)\right)$.

Lemma 4.6. $IO_{AFIX}\left(P,M,m^k,b\right) = o\left(IO_{HYB}\left(P,M,m^k,b\right)\right)$.

[1] One can completely avoid interprocessor I/O-complexity overhead by using the standard composition (Definition 3.5) until the blocks fit a single processor's memory and then switch to COAT. This change slightly increases the I/O and the arithmetic complexity by reducing k.

Proof. AFIX performs k recursive steps before computing μ. We first consider the case where $b^2 \leq M$, namely an entire sub-block fits into one processor's local memory. Computing μ does not affect the I/O-complexity. Therefore the same analysis as in [3] show that $IO_{\text{AFIX}}\left(P, M, m^k, b\right) = O\left(\frac{M}{P}\left(\frac{bm^k}{\sqrt{M}}\right)^{\log_a d_u} + \frac{b^2 m^{2k}}{P^{\frac{2}{\log_m d_u}}}\right)$.

Otherwise (if $b^2 > M$), all processors are utilized to compute the d_u recursive calls to block multiplications of size $bm^{k-1} \times bm^{k-1}$. When the algorithm reaches the base case, a sub-block of size $b \times b$, by Lemma 4.5, computing μ takes $O\left(\frac{b^2}{P}\right)$ communication. Thus,

$$IO_{\text{AFIX}}\left(P, M, m^k, b\right) = \begin{cases} O\left(\frac{b^2}{P}\right) & \text{if } k = 0 \\ d_u IO_{\text{AFIX}}\left(M, P, m^{k-1}, b\right) & \text{otherwise} \end{cases}$$

Thus, $IO_{\text{AFIX}}\left(P, M, m^k, b\right) = O\left(d_u^k \cdot \frac{b^2}{P}\right)$. By Claim 4.7, in both cases $IO_{\text{AFIX}}\left(P, M, m^k, b\right) = o\left(IO_{\text{HYB}}\left(P, M, m^k, b\right)\right)$.

□

Claim 4.7 ([4, 36, 42, 44]). $IO_{\text{HYB}}\left(P, M, m^k, b\right) =$

$$\Omega\left(\left(\frac{bm^k}{\max\left(b, \sqrt{M}\right)}\right)^{\log_m t} \frac{\max\left(\frac{b^3}{\sqrt{M}}, M\right)}{P} + \frac{b^2 m^{2k}}{P^{\frac{2}{\log_m t}}}\right)$$

Proof of Theorem 4.1. let FIX denote computing both AFIX and BFIX. Now, by Lemma 4.4 and Lemma 4.6,

$$\begin{aligned} IO_{\text{GFA}}\left(P, M, m^k, b\right) &= IO_{\text{RCA}}\left(P, M, m^k, b\right) + IO_{\text{FIX}}\left(P, M, m^k, b\right) \\ &= (1 + o(1))\, IO_{\text{HYB}}\left(P, M, m^k, b\right) \end{aligned}$$

□

4.3.2 *The sequential model.*

Theorem 4.2. $IO_{GFA}\left(M, m^k, b\right) = (1 + o(1))\, IO_{HYB}\left(M, m^k, b\right)$.

The analysis is practically the same as in the parallel model, plugging in $P = 1$ and skipping the memory independent case. The proof will appear in the full version of this paper.

5 DISCUSSION

We improve the arithmetic complexity of the inner sub-blocks in a matrix multiplication algorithm by a factor of $\frac{\rho+3}{2\rho+2}$. To this end, we generalize Winograd's folding technique. The resulting algorithm beats the classic algorithm even on small blocks as long as the input matrices are sufficiently large. It also attains the lower bound, hence is optimal. COAT can also be combined with the base change technique of [31] to simultaneously speed up both the recursive and iterative parts of the algorithm.

The algorithm is energy-saving as well. As Table 2 in [30] shows, on Google's TPU, multiplication requires up to 49 times more energy than addition on a 45-nm chip and as much as $3 - 31$ additions on a 7-nm chip. This translates to $25 - 46\%$ energy saving on TPU-like implementations of COAT.

In addition, one can combine the algorithm with pipelining instructions when coding. Pipelining does not affect energy consumption. Further, considering the run time, the ρ values of pipelined instructions are larger than 1 in many cases. For example, on Intel's IceLake microarchitecture, ρ can go up to 4 for integers when taking pipeline into account (see [22]).

We have generalized the Brent equations to apply to commutative algorithms. The generalization makes it easier to verify automatically whether a triplet of matrices represents a commutative matrix multiplication algorithm.

Future work includes applying our technique to other outer algorithms. Promising candidates appear in [37] as they have a small exponent and many duplicate rows. Another application combines our technique with the basis transformation techniques of [6, 31]. Further study includes finding other recursive-bilinear algorithms that can benefit from our techniques.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant egreement No 818252).

REFERENCES

[1] Josh Alman and Virginia V. Williams. 2021. A refined laser method and faster matrix multiplication, In Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA). *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, 522–539. https://doi.org/10.1137/1.9781611976465.32

[2] N. Anderson and D. Manley. 1994. A matrix extension of Winograd's inner product algorithm. *Theoretical Computer Science* 131 (1994). Issue 2. https://doi.org/10.1016/0304-3975(94)90186-4

[3] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. 2012. Communication-optimal parallel algorithm for strassen's matrix multiplication, In Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures. *Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 193–204. https://doi.org/10.1145/2312005.2312044

[4] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Graph expansion and communication costs of fast matrix multiplication. *Annual ACM Symposium on Parallelism in Algorithms and Architectures* 59, 6 (2011), 1–23. https://doi.org/10.1145/1989493.1989495

[5] Gal Beniamini, Nathan Cheng, Olga Holtz, Elaye Karstadt, and Oded Schwartz. 2020. Sparsifying the Operators of Fast Matrix Multiplication Algorithms. *arXiv preprint arXiv:2008.03759* (2020).

[6] Gal Beniamini and Oded Schwartz. 2019. Faster matrix multiplication via sparse decomposition. *Annual ACM Symposium on Parallelism in Algorithms and Architectures* (6 2019), 11–22. https://doi.org/10.1145/3323165.3323188

[7] Austin R. Benson and Grey Ballard. 2015. A framework for practical parallel fast matrix multiplication. *ACM SIGPLAN Notices* 50 (2015). Issue 8. https://doi.org/10.1145/2858788.2688513

[8] Dario A. Bini. 1980. Relations between exact and approximate bilinear algorithms. Applications. *Calcolo* 17 (1980). Issue 1. https://doi.org/10.1007/BF02575865

[9] Markus Bläser. 1999. Lower bounds for the multiplicative complexity of matrix multiplication. *Computational Complexity* 8 (1999). Issue 3. https://doi.org/10.1007/s000370050028

[10] Markus Bläser. 2001. A $\frac{5}{2}n^2$-lower bound for the multiplicative complexity of $n \times n$-matrix multiplication. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2010. https://doi.org/10.1007/3-540-44693-1_9

[11] Richard P. Brent. 1970. *Algorithms for matrix multiplication.* Technical Report. Stanford University, CA. Department of computer science.

[12] Richard P. Brent. 1970. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Numer. Math.* 16 (1970). Issue 2. https://doi.org/10.1007/BF02308867

[13] Nader H. Bshouty. 1995. On the additive complexity of 2×2 matrix multiplication. *Inform. Process. Lett.* 56, 6 (1995), 329–335. Issue 6. https://doi.org/10.1016/0020-0190(95)00176-X

[14] Murat Cenk and M. Anwar Hasan. 2017. On the arithmetic complexity of Strassen-like matrix multiplications. *Journal of Symbolic Computation* 80 (2017), 484–501. https://doi.org/10.1016/j.jsc.2016.07.004

[15] Henry Cohn and Christopher Umans. 2003. A group-theoretic approach to fast matrix multiplication. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.* IEEE, 438–449.

[16] Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation* 9 (1990). Issue 3. https://doi.org/10.1016/S0747-7171(08)80013-2

[17] Alexander M. Davie and Andrew J. Stothers. 2013. Improved bound for complexity of matrix multiplication. *Proceedings of the Royal Society of Edinburgh Section A: Mathematics* 143 A (2013). Issue 2. https://doi.org/10.1017/S0308210511001648

[18] Hans F De Groote. 1987. *Lectures on the complexity of bilinear problems.* Vol. 245. Springer Science & Business Media.

[19] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. 2013. Communication-optimal parallel recursive rectangular matrix multiplication. *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013.* https://doi.org/10.1109/IPDPS.2013.80

[20] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610, 7930 (2022), 47–53.

[21] Patrick C Fischer. 1974. Further schemes for combining matrix algorithms. In *International Colloquium on Automata, Languages, and Programming.* Springer, 428–436.

[22] Agner Fog. 2022. Instruction tables. *Technical University of Denmark* (2022). https://www.agner.org/optimize/instruction_tables.pdf

[23] François Le Gall. 2014. Powers of tensors and fast matrix multiplication, In Proceedings of the 39th international symposium on symbolic and algebraic computation. *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC*, 296–303. https://doi.org/10.1145/2608628.2608664

[24] Marijn J.H. Heule, Manuel Kauers, and Martina Seidl. 2021. New ways to multiply 3×3-matrices. *Journal of Symbolic Computation* 104 (2021). https://doi.org/10.1016/j.jsc.2020.10.003

[25] John E Hopcroft and Leslie R Kerr. 1971. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM J. Appl. Math.* 20, 1 (1971), 30–36.

[26] Joseph JáJá. 1980. On the Complexity of Bilinear Forms with Commutativity. *SIAM J. Comput.* 9 (1980). Issue 4. https://doi.org/10.1137/0209056

[27] Larisa D. Jelfimova. 2019. A New Fast Recursive Matrix Multiplication Algorithm. *Cybernetics and Systems Analysis* 55 (7 2019), 547–551. Issue 4. https://doi.org/10.1007/s10559-019-00163-2

[28] Larisa D. Jelfimova. 2021. A Fast Recursive Algorithm for Multiplying Matrices of Order $n = 3^q$ (q > 1). *Cybernetics and Systems Analysis* 57 (2021). Issue 2. https://doi.org/10.1007/s10559-021-00345-x

[29] Hong Jia-Wei and Hsiang-Tsung Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing.* 326–333.

[30] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten lessons from three generations shaped Google's TPUv4i: Industrial product. *Proceedings - International Symposium on Computer Architecture* 2021-June. https://doi.org/10.1109/ISCA52012.2021.00010

[31] Elaye Karstadt and Oded Schwartz. 2020. Matrix Multiplication, a Little Faster. *Journal of the ACM (JACM)* 67, 1 (2020), 1–31.

[32] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–22.

[33] Julian D. Laderman. 1976. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. In *American Mathematical Society*, Vol. 82. 126–128.

[34] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-avoiding parallel Strassen: Implementation and performance. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC.* https://doi.org/10.1109/SC.2012.33

[35] Charles F. Van Loan. 2000. The ubiquitous Kronecker product. *Journal of computational and applied mathematics* 123, 1-2 (2000), 85–100. https://doi.org/10.1016/S0377-0427(00)00393-9

[36] Roy Nissim and Oded Schwartz. 2019. Revisiting the I/O-complexity of fast matrix multiplication with recomputations. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 482–490.

[37] Victor Y. Pan. 1980. New Fast Algorithms for Matrix Operations. *SIAM J. Comput.* 9 (1980). Issue 2. https://doi.org/10.1137/0209027

[38] Victor Y. Pan. 1982. Trilinear aggregating with implicit canceling for a new acceleration of matrix multiplication. *Computers and Mathematics with Applications* 8 (1982). Issue 1. https://doi.org/10.1016/0898-1221(82)90037-2

[39] Robert L Probert. 1976. On the additive complexity of matrix multiplication. *SIAM J. Comput.* 5, 2 (1976), 187–203.

[40] Andreas Rosowski. 2019. Fast Commutative Matrix Algorithm. *arXiv preprint arXiv:1904.07683* (4 2019). http://arxiv.org/abs/1904.07683

[41] Arnold Schönhage. 1981. Partial and Total Matrix Multiplication. *SIAM J. Comput.* 10 (1981). Issue 3. https://doi.org/10.1137/0210032

[42] Jacob Scott, Olga Holtz, and Oded Schwartz. 2015. Matrix multiplication I/O-complexity by path routing. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures.* 35–45.

[43] Alexey V. Smirnov. 2013. The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics* 53 (2013). Issue 12. https://doi.org/10.1134/S0965542513120129

[44] Lorenzo De Stefani. 2019. The I/O Complexity of Hybrid Algorithms for Square Matrix Multiplication. In *30th International Symposium on Algorithms and Computation, ISAAC 2019, December 8-11, 2019, Shanghai University of Finance and Economics, Shanghai, China (LIPIcs, Vol. 149)*, Pinyan Lu and Guochuan Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:16. https://doi.org/10.4230/LIPIcs.ISAAC.2019.33

[45] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13 (1969). Issue 4. https://doi.org/10.1007/BF02165411

[46] Volker Strassen. 1973. Vermeidung von Divisionen. *Journal fur die Reine und Angewandte Mathematik* 1973 (1973). Issue 264. https://doi.org/10.1515/crll.1973.264.184

[47] Abraham Waksman. 1970. On Winograd's Algorithm for Inner Products. *IEEE Trans. Comput.* C-19 (1970). Issue 4. https://doi.org/10.1109/T-C.1970.222926

[48] Virginia V. Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd, In Proceedings of the forty-fourth annual ACM symposium on Theory of computing. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 887–898. https://doi.org/10.1145/2213977.2214056

[49] Shmuel Winograd. 1968. A New Algorithm for Inner Product. *IEEE Trans. Comput.* C-17 (1968), 693–694. Issue 7. https://doi.org/10.1109/TC.1968.227420

[50] Shmuel Winograd. 1968. On the number of multiplications necessary to compute certain functions. *Communications on Pure and Applied Mathematics* 23 (1968). Issue 2. https://doi.org/10.1002/cpa.3160230204

[51] Shmuel Winograd. 1971. On multiplication of 2×2 matrices. *Linear Algebra and Its Applications* 4, 4 (1971), 381–388. Issue 4. https://doi.org/10.1016/0024-3795(71)90009-7

[52] Shmuel Winograd. 1976. Private communication with R. Probert [39].

[53] Cui Qing Yang and Barton P. Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. *Proceedings - International Conference on Distributed Computing Systems* 8. https://doi.org/10.1109/dcs.1988.12538