Matrix Multiplication, a Little Faster

ELAYE KARSTADT and ODED SCHWARTZ, The Hebrew University of Jerusalem

Strassen's algorithm (1969) was the first sub-cubic matrix multiplication algorithm. Winograd (1971) improved the leading coefficient of its complexity from 6 to 7. There have been many subsequent asymptotic improvements. Unfortunately, most of these have the disadvantage of very large, often gigantic, hidden constants. Consequently, Strassen-Winograd's $O(n^{\log_2 7})$ algorithm often outperforms other fast matrix multiplication algorithms for all feasible matrix dimensions. The leading coefficient of Strassen-Winograd's algorithm has been generally believed to be optimal for matrix multiplication algorithms with a 2×2 base case, due to the lower bounds by Probert (1976) and Bshouty (1995).

Surprisingly, we obtain a faster matrix multiplication algorithm, with the same base case size and asymptotic complexity as Strassen-Winograd's algorithm, but with the leading coefficient reduced from 6 to 5. To this end, we extend Bodrato's (2010) method for matrix squaring, and transform matrices to an alternative basis. We also prove a generalization of Probert's and Bshouty's lower bounds that holds under change of basis, showing that for matrix multiplication algorithms with a 2×2 base case, the leading coefficient of our algorithm cannot be further reduced, and is therefore optimal. We apply our method to other fast matrix multiplication algorithms, improving their arithmetic and communication costs by significant constant factors.

CCS Concepts: • **Mathematics of computing → Computations on matrices**; • **Computing methodologies → Linear algebra algorithms**;

Additional Key Words and Phrases: Fast matrix multiplication, bilinear algorithms

ACM Reference format:
Elaye Karstadt and Oded Schwartz. 2020. Matrix Multiplication, a Little Faster. *J. ACM* 67, 1, Article 1 (January 2020), 31 pages.
https://doi.org/10.1145/3364504

A preliminary version of this paper appeared in Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'17) [41].
This research is supported by grants 1878/14, and 1901/14 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities) and grant 3-10891 from the Ministry of Science and Technology, Israel. This research was also supported by the Einstein Foundation and the Minerva Foundation; the PetaCloud industry-academia consortium; by a grant from the United States-Israel Bi-national Science Foundation, Jerusalem, Israel; and the HUJI Cyber Security Research Center in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office. We acknowledge PRACE for awarding us access to Hazel Hen at GCS@HLRS, Germany. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 818252).
Authors' addresses: E. Karstadt and O. Schwartz, The Rachel and Selim Benin, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Rothberg Family Buildings, The Edmond J. Safra Campus, 9190416 Jerusalem, Israel; emails: elaye.karstadt@mail.huji.ac.il, odedsc@cs.huji.ac.il.

0004-5411/2020/01-ART1 $15.00
https://doi.org/10.1145/3364504

1 INTRODUCTION

Strassen's algorithm [58] was the first sub-cubic matrix multiplication algorithm, with arithmetic complexity $O(n^{\log_2 7})$. Subsequently, Winograd [61] reduced the leading coefficient of the arithmetic complexity from 7 to 6, by decreasing the number of additions and subtractions from 18 to 15. Many fast matrix multiplication algorithms followed and improved the asymptotic complexity (e.g., [12, 21, 22, 44, 50, 51, 53, 54, 57, 59, 60]). However, most of these improvements have come at the cost of very large, often gigantic, hidden constants. The leading coefficient of Strassen-Winograd's algorithm has generally been believed to be optimal, due to lower bounds of Probert [52] and Bshouty [16]. We present a new method which enables pushing beyond those lower bounds, and also apply our method to improve the leading coefficient of other fast matrix multiplication algorithms.

1.1 Previous Work

In order for fast matrix multiplication algorithms to be practically useful, they need to have a reasonably small base case size, and manageable hidden constants that do not hinder performance. The following efforts have been previously made to improve these issues.

Small base cases. There are a number of known fast matrix multiplication algorithms with reasonably small base case size, both for square and for rectangular matrices [36, 39, 43, 50, 51, 56]. Some of these are asymptotically faster than Strassen's algorithm and the fastest is Pan's $\langle 44, 44, 44; 36133\rangle$-algorithm,[1] with arithmetic complexity of $O(n^{\omega_0})$ where $\omega_0 = \log_{44} 36133 < 2.7734$. In practice, due to the hidden constants [10], Strassen-Winograd's algorithm often outperforms asymptotically faster algorithms, although some have achieved a performance similar to that of Strassen-Winograd's algorithm [61] (e.g., Kaporin's [40] implementation of an $\langle 48, 48, 48; 46464\rangle$-algorithm with $\omega_0 = \log_{48} 46464 < 2.776$, based on Laderman et al.'s algorithm [43]).

Computer aided search for new algorithms. Smirnov presented several fast matrix multiplication algorithms derived by computer aided search [56]. These include a $\langle 6, 3, 3; 40\rangle$-algorithm, which translates (using [37]) to a $\langle 54, 54, 54; 40^3\rangle$-algorithm, namely, asymptotic arithmetic complexity of $O(n^{3\log_{54} 40})$, with exponent $\omega_0 < 2.7743$, which is asymptotically faster than Strassen's algorithm but slower than Pan's $\langle 40, 40, 40; 36133\rangle$-algorithm [51]. Ballard and Benson [10] found several additional fast matrix multiplication algorithms, using computer aided optimization tools based on Johnson and McLoughlin, and Smirnov's work [38, 56]. They implemented selected algorithms, including Smirnov's $\langle 6, 3, 3; 40\rangle$-algorithm, on a shared-memory architecture, and demonstrated that in practice, these could outperform classical matrix multiplication, on modestly sized problems (up to $n \approx 13{,}000$). Even so, Strassen's algorithm still outperformed Smirnov's algorithm in some of the cases.

Intermediate representation. Bodrato [14] introduced the intermediate representation method, for repeated squaring and for chain matrix multiplication computations. This enables the number of additions between consecutive multiplications to be decreased. In this way, he obtained an algorithm with a 2×2 base case, which uses seven multiplications, and has an arithmetic complexity of $5n^{\log_2 7} - 4n^2$ instead of $6n^{\log_2 7} - 5n^2$ for chain multiplication and for repeated squaring, for every multiplication outside the first one. Bodrato also presented an invertible linear function which recursively transforms a $2^k \times 2^k$ matrix to and from the intermediate transformation.

Non-uniform recursion. Very recently, Cenk and Hasan [17] described a clever way to apply Strassen-Winograd's algorithm directly to $n \times n$ matrices by forsaking the uniform

[1] See Section 2.1 for definition.

Table 1. 2×2 Fast Matrix Multiplication Algorithms[2]

Algorithm	Additions	Arithmetic Complexity	IO-Complexity
Strassen [58]	18	$7n^{\log_2 7} - 6n^2$	$12 \cdot M \left(\sqrt{3} \cdot \frac{n}{\sqrt{M}}\right)^{\log_2 7} - 18n^2$
Strassen-Winograd [61]	15	$6n^{\log_2 7} - 5n^2$	$10.5 \cdot M \left(\sqrt{3} \cdot \frac{n}{\sqrt{M}}\right)^{\log_2 7} - 15n^2$
Ours	12	$5n^{\log_2 7} - 4n^2 + 3n^2 \log_2 n$	$9 \cdot M \left(\sqrt{3} \cdot \frac{n}{\sqrt{M}}\right)^{\log_2 7} + 9n^2 \cdot \log_2 \left(\sqrt{2} \cdot \frac{n}{\sqrt{M}}\right)$

divide-and-conquer pattern. Instead, their algorithm splits Strassen-Winograd's algorithm into two linear divide-and-conquer algorithms, which recursively perform all pre-computations, followed by vector multiplication of their results, and finally by linear post-computations to calculate the output. Their method enables reuse of sums, resulting in a matrix multiplication algorithm with an arithmetic complexity of $5n^{\log_2 7} + 0.5n^{\log_2 6} + 2n^{\log_2 5} - 6.5n^2$, which is only slightly more expensive than ours. However, this comes at the cost of a large increase in communication costs and memory footprint.

Communication costs. The communication costs (IO-complexity) of fast matrix multiplication algorithms are a measure of the data transferred within a processor's memory hierarchy, and between processors. Both often take significantly more time than the arithmetic costs. Communication costs are measured as a function of the matrix dimension n, the number of processors P, and the size M of the local memories. For fast matrix multiplication with arithmetic cost of $\Theta(n^{\omega_0})$, the communication cost is bounded below by $\Omega((\frac{n}{M^{1/2}})^{\omega_0} \frac{M}{P})$ [7, 11, 55]. The technique in [7] allows memory independent lower bound of $\Omega(\frac{n^2}{P^{2/\omega_0}})$ [6]. These bounds apply to the sequential case as well, with $P = 1$.

In the sequential model, these lower bounds are attained by the basic block-recursive implementation, as detailed by Strassen [58]. Some of the parallelizations do not attain the communication lower bounds [26, 32, 46], while others do [5, 47] (up to a logarithmic factor in P), and hence minimize communication costs. In practice, parallel versions of Strassen's algorithm, which minimize communication cost, outperform the well tuned classical matrix multiplication, both in shared-memory [10, 23, 42] and distributed-memory architectures [5, 45].

1.2 Our Contribution

We improve the leading coefficient of Strassen's and other fast matrix multiplication algorithms. This was achieved by extending Bodrato's [14] intermediate representation method for matrix squaring to an *alternative basis matrix multiplication* method (see Section 3). While basis transformation is generally as expensive as matrix multiplications, some can be performed very fast (e.g., Hadamard in $O(n^2 \log n)$ using FFT [20]). Fortunately, this is also the case for our basis transformation (see Section 3.1). Thus, our method represents a worthwhile tradeoff where the leading coefficient is reduced in exchange for a small, asymptotically insignificant, overhead (see Section 3.2). We also describe how to efficiently compose alternative basis matrix multiplication algorithms in order to create new algorithms for various base case sizes (see Section 4), and discuss how these

[2] A previous version of this work [41] claimed smaller leading coefficients for the IO-complexity, due to a miscalculation.

Table 2. Alternative Basis Algorithms

Algorithm	Linear Operations	Improved Linear Operations	Arithmetic Leading Coefficient	Improved Leading Coefficient	Computations Saved
$\langle 2, 2, 2; 7\rangle$ [61]	15	12	6	5	16.6%
$\langle 3, 2, 3; 15\rangle$ [56]	64	52	9.61	7.94	17.37%
$\langle 2, 3, 4; 20\rangle$[56]	78	58	8.8	7.46	16.18%
$\langle 3, 3, 3; 23\rangle$ [56]	87	75	7.21	6.57	8.87%
$\langle 6, 3, 3; 40\rangle$[56]	1246	198	55.63	9.36	83.17%

constants are affected and the impact on both arithmetic and IO-complexity. While this is not the first time that linear transformations have been applied to matrix multiplication, the main focus of previous research on the subject was to improve the asymptotics rather than improving the leading coefficient [19, 33].

In addition, we discuss the problem of finding alternative bases to improve fast matrix multiplication algorithms (see Section 6), and present several improved variants of existing algorithms. The most notable of these improvements are the alternative basis for Strassen's $\langle 2, 2, 2; 7\rangle$-algorithm which reduces the number of additions from 15 to 12 (see Section 3.3), and the one for Smirnov's $\langle 6, 3, 3; 40\rangle$-algorithm with leading coefficient reduced by about 83.9%.[3]

Theorem 1.1 (Probert and Bshouty's Lower Bounds [16, 52]). *15 additions are necessary for any $\langle 2, 2, 2; 7\rangle$-algorithm.*

Probert proved this bound for any $\langle 2, 2, 2; 7\rangle$-algorithms over $\mathbb{F}_2$, and Bshouty [16] later obtained the same lower bound over an arbitrary ring, using a different method. Although our result seemingly refutes these lower bounds, there is actually no contradiction: our *alternative basis matrix multiplication* method bypasses Probert's and Bshouty's implicit assumption that the input and output are represented in the standard basis. Furthermore, we extend the lower bound of Probert and Bshouty to apply to our alternative basis matrix multiplication (see Section 5).

Theorem 1.2 (Basis Invariant Lower Bound). *12 additions are necessary for any matrix multiplication algorithm that uses a recursive-bilinear algorithm with a 2×2 base case with seven multiplications, regardless of basis.*

Our alternative basis variant of Strassen's algorithm performs 12 additions in the base case, matching the lower bound in Theorem 1.2. Hence, it is optimal. It reduces the arithmetic complexity by 16.66%, and shrinks the IO-complexity by 14.29%, compared to Strassen-Winograd's algorithm, predicting a performance in improvement in the range of 14–16% on a shared-memory machine.

2 PRELIMINARIES

2.1 Fast Matrix Multiplication Algorithms

Fast matrix multiplication algorithms are recursive divide-and-conquer algorithms, which utilize a base-case $\langle n_0, m_0, k_0; t_0\rangle$-algorithm: multiplying an $n_0 \times m_0$ matrix by an $m_0 \times k_0$ matrix using t_0 scalar multiplications, where n_0, m_0, k_0, and t_0 are positive integers. When multiplying an $n \times m$ matrix by an $m \times k$ matrix, the algorithm splits them into blocks (each of size $\frac{n}{n_0} \times \frac{m}{m_0}$ and $\frac{m}{m_0} \times \frac{k}{k_0}$, respectively), and works block-wise, according to the base algorithm. Additions and subtractions

[3]Encoding/decoding matrices and the corresponding basis transformations can be found in Appendix B. Files with additional alternative basis algorithms can be found at https://github.com/elayeek/matmultfaster.

in the base-case algorithm are interpreted as block-wise additions and subtractions. Similarly, multiplication by a scalar is interpreted as multiplication of block matrix by a scalar. Multiplications of linear sums of matrix elements in the base-case algorithm are interpreted as block-wise multiplications via recursion. Throughout this work, we refer to an algorithm by its base case. Hence, an $\langle n, m, k; t\rangle$-algorithm may refer to either the algorithm's base case or the corresponding block recursive algorithm, as obvious from context.

2.2 Encoding and Decoding Matrices

Fact 2.1. *Let R be a ring, and let $f : R^n \times R^m \to R^k$ be a bilinear function which performs t multiplications. There exist $U \in R^{t\times n}$, $V \in R^{t\times m}$, $W \in R^{t\times k}$ such that*

$$\forall x \in R^n,\ y \in R^m\ f(x, y) = W^T((U \cdot x) \odot (V \cdot y)),$$

where $\odot$ is the element-wise vector product (Hadamard product).

Definition 2.2 (Encoding/Decoding Matrices). We refer to the $\langle U, V, W\rangle$ of a recursive-bilinear algorithm as its encoding/decoding matrices (where U, V are the encoding matrices and W is the decoding matrix).

Definition 2.3. Let R be a ring, and let $A \in R^{n\times m}$ be a matrix. We denote the vectorization of A by $\vec{A}$ and use the notation $U_{\ell,(i,j)}$ when referring to the element in the ℓ'th row on the column corresponding with the index (i, j) in the vectorization of A. For ease of notation, we sometimes write $A_{i,j}$ rather than $\vec{A}_{(i,j)}$.[4]

Fact 2.4 (Triple Product Condition [15]). *Let R be a ring, and let $U \in R^{t\times n\cdot m}$, $V \in R^{t\times m\cdot k}$, and $W \in R^{t\times n\cdot k}$. $\langle U, V, W\rangle$ are encoding/decoding matrices of an $\langle n, m, k; t\rangle$-algorithm if and only if*

$$\forall i_1, i_2 \in [n],\ k_1, k_2 \in [m],\ j_1, j_2 \in [k],$$

$$\sum_{r=1}^{t} U_{r,(i_1,k_1)} V_{r,(k_2,j_1)} W_{r,(i_2,j_2)} = \delta_{i_1,i_2}\delta_{k_1,k_2}\delta_{j_1,j_2},$$

where $\delta_{i,j} = 1$ if $i = j$ and 0 if otherwise.

Notation 2.5. *Denote the number of non-zero entries in a matrix by $nnz(A)$, and the number of rows/columns by $rows(A), cols(A)$.*

Remark 2.6. The number of linear operations used by a bilinear algorithm is determined by its encoding/decoding matrices. The number of additions performed by each of the encodings is

$$\texttt{AdditionsU} = nnz(U) - rows(U),$$
$$\texttt{AdditionsV} = nnz(V) - rows(V).$$

The number of additions performed by the decoding is

$$\texttt{AdditionsW} = nnz(W) - cols(W).$$

The number of scalar multiplications performed by each of the encodings/decodings is equal to the total number of matrix entries that are not 1, −1, or 0.

Remark 2.7. We assume that none of the rows of the U, V, and W matrices is all-zero. This is because any all-zero row in U, V is equivalent to an identically 0 multiplication, and any all-zero row in W is equivalent to a multiplication that is never used in the output. Hence, such rows can be omitted, resulting in asymptotically faster algorithms.

[4] All basis transformations and encoding/decoding matrices assume row-ordered vectorization of matrices.

3 ALTERNATIVE BASIS MATRIX MULTIPLICATION

Fast matrix multiplication algorithms are bilinear computations. The number of operations performed in the linear phases of such algorithms (for the application of their encoding/decoding matrices $\langle U, V, W\rangle$ in the case of matrix multiplication, see Definition 2.2) depends on the basis of the representation. In this section, we describe how alternative basis algorithms work and address the effects of using alternative bases on arithmetic complexity and IO-complexity.

Definition 3.1. Let R be a ring and let ϕ, ψ, υ be automorphisms of $R^{n\cdot m}$, $R^{m\cdot k}$, $R^{n\cdot k}$ (respectively). We denote a fast matrix multiplication algorithm which takes $\phi(A)$, $\psi(B)$ as inputs and outputs $\upsilon(A\cdot B)$ using t multiplications by $\langle n, m, k; t\rangle_{\phi,\psi,\upsilon}$-algorithm. If $n = m = k$ and $\phi = \psi = \upsilon$, we use the notation $\langle n, n, n; t\rangle_{\phi}$-algorithm. This notation extends the $\langle n, m, k; t\rangle$-algorithm notation as the latter applies when the three basis transformations are the identity map.

Given a recursive-bilinear, $\langle n, m, k; t\rangle_{\phi,\psi,\upsilon}$-algorithm, *ALG*, alternative basis matrix multiplication works as follows.

ALGORITHM 1: Alternative Basis Matrix Multiplication Algorithm

Input: $A \in R^{n\times m}$, $B^{m\times k}$
Output: $n \times k$ matrix $C = A \cdot B$

```
1: function Mult(A, B)
2:     Ã = φ(A)          ▷ R^{n×m} basis transformation
3:     B̃ = ψ(B)          ▷ R^{m×k} basis transformation
4:     C̃ = ALG(Ã, B̃)     ▷ ⟨n, m, k; t⟩_{φ,ψ,υ}-algorithm
5:     C = υ^{-1}(C̃)     ▷ R^{n×k} basis transformation
6: return C
```

Lemma 3.2. *Let R be a ring, let $\langle U, V, W\rangle$ be the encoding/decoding matrices of an $\langle n, m, k; t\rangle$-algorithm, and let ϕ, ψ, υ be automorphisms of $R^{n\cdot m}$, $R^{m\cdot k}$, $R^{n\cdot k}$ (respectively). $\langle U\phi^{-1}, V\psi^{-1}, W\upsilon^{T}\rangle$ are encoding/decoding matrices of an $\langle n, m, k; t\rangle_{\phi,\psi,\upsilon}$-algorithm.*

Proof. $\langle U, V, W\rangle$ are encoding/decoding matrices of an $\langle n, m, k; t\rangle$-algorithm. Hence, for any $A \in R^{n\times m}$, $B \in R^{m\times k}$

$$W^T((U\cdot\vec{A})\odot(V\cdot\vec{B})) = \overrightarrow{A\cdot B}.$$

Hence,

$$\upsilon(\overrightarrow{A\cdot B}) = \upsilon(W^T(U\cdot\vec{A}\odot V\cdot\vec{B})) = (W\upsilon^T)^T(U\phi^{-1}\cdot\phi(\vec{A})\odot V\psi^{-1}\cdot\psi(\vec{B})).$$ □

Corollary 3.3. *Let R be a ring, and let ϕ, ψ, υ be automorphisms of $R^{n\cdot m}$, $R^{m\cdot k}$, $R^{n\cdot k}$ (respectively). $\langle U, V, W\rangle$ are encoding/decoding matrices of an $\langle n, m, k; t\rangle_{\phi,\psi,\upsilon}$-algorithm if and only if $\langle U\phi, V\psi, W\upsilon^{-T}\rangle$ are encoding/decoding matrices of an $\langle n, m, k; t\rangle$-algorithm.*

3.1 Fast Basis Transformation

Definition 3.4. Let R be a ring and let $\psi : R^{n_0\times m_0} \to R^{n_0\times m_0}$ be a linear map. We recursively define a linear map $\psi^{k+1} : R^{n\times m} \to R^{n\times m}$ (where $n = n_0^{\ell_1}$, $m = m_0^{\ell_2}$ for some $\ell_1, \ell_2 \le k+1$) by $(\psi^{k+1}(A))_{i,j} = \psi^k(\psi(A))_{i,j}$, where $A_{i,j}$ are $\frac{n}{n_0}\times\frac{m}{m_0}$ sub-matrices.

Note that ψ^{k+1} is a linear map. For convenience, we omit the subscript of ψ when obvious from the context.

Claim 3.5. *Let R be a ring, let $\psi : R^{n_0 \times m_0} \to R^{n_0 \times m_0}$ be a linear map, and let $A \in R^{n \times m}$ (where $n = n_0^{k+1}$, $m = m_0^{k+1}$). Define $\tilde{A}$ by $(\tilde{A})_{i,j} = \psi^k(A_{i,j})$. Then $\psi(\tilde{A}) = \psi^{k+1}(A)$.*

Proof. ψ is a linear map. Hence, for any $i \in [n_0]$, $j \in [m_0]$, $(\psi(A))_{i,j}$ is a linear sum of elements of A. Therefore, there exist scalars $\{x_{r,\ell}^{(i,j)}\}_{r \in [n_0],\, \ell \in [m_0]}$ such that

$$(\psi^{k+1}(A))_{i,j} = \psi^k(\psi(A))_{i,j} = \psi^k\left(\sum_{r,\ell} x_{r,\ell}^{(i,j)} \cdot A_{r,\ell}\right).$$

By linearity of ψ^k

$$= \sum_{r,\ell} x_{r,\ell}^{(i,j)} \psi^k(A_{r,\ell}) = (\psi(\tilde{A}))_{i,j}. \qquad \square$$

Claim 3.6. *Let R be a ring, let $\psi : R^{n_0 \times m_0} \to R^{n_0 \times m_0}$ be an invertible linear map, and let ψ^{k+1} be as defined above. ψ^{k+1} is invertible and its inverse is $(\psi^{-(k+1)}(A))_{i,j} = \psi^{-k}(\psi^{-1}(A))_{i,j}$.*

Proof. Define $\tilde{A}$ by $(\tilde{A})_{i,j} = \psi^k(A_{i,j})$ and define $\psi^{-(k+1)}$ by $(\psi^{-(k+1)}(A))_{i,j} = \psi^{-k}(\psi^{-1}(A))_{i,j}$. Then

$$(\psi^{-(k+1)}(\psi^{k+1}(A)))_{i,j} = \psi^{-k}(\psi^{-1}(\psi^{k+1}(A)))_{i,j}.$$

By Claim 3.5

$$= \psi^{-k}(\psi^{-1}(\psi(\tilde{A})))_{i,j} = \psi^{-k}(\tilde{A})_{i,j}.$$

By definition of $\tilde{A}$

$$= \psi^{-k}(\psi^k(A)_{i,j}) = A_{i,j}. \qquad \square$$

We next analyze the arithmetic complexity and IO-complexity of fast basis transformations.

Claim 3.7. *Let R be a ring, let $\psi : R^{n_0 \times m_0} \to R^{n_0 \times m_0}$ be a linear map, and let $A \in R^{n \times m}$ where $n = n_0^k$, $m = m_0^k$. The arithmetic complexity of computing $\psi(A)$ is*

$$F_\psi(n, m) = \frac{q}{n_0 m_0} nm \cdot \log_{n_0 m_0}(nm),$$

where q is the number of linear operations performed by ψ.

Proof. Let $F_\psi(n, m)$ be the number of additions required by ψ. Each step of the recursion consists of computing $n_0 \cdot m_0$ sub-problems and performs q additions of sub-matrices. Therefore, $F_\psi(n, m) = n_0 m_0 F_\psi(\frac{n}{n_0}, \frac{m}{m_0}) + q(\frac{nm}{n_0 m_0})$ and $F_\psi(1, 1) = 0$. Thus,

$$\begin{aligned} F_\psi(n, m) &= n_0 m_0 F_\psi\left(\frac{n}{n_0}, \frac{m}{m_0}\right) + q\left(\frac{nm}{n_0 m_0}\right) \\ &= \sum_{k=0}^{\log_{n_0 m_0}(nm)-1} (n_0 m_0)^k \left(q \frac{nm}{(n_0 m_0)^{k+1}}\right) \\ &= \frac{q}{n_0 m_0} nm \cdot \sum_{k=0}^{\log_{n_0 m_0}(nm)-1} \left(\frac{n_0 m_0}{n_0 m_0}\right)^k = \frac{q}{n_0 m_0} nm \cdot \log_{n_0 m_0}(nm). \end{aligned} \qquad \square$$

Corollary 3.8. *Let R be a ring, let $\psi : R^{n_0 \times n_0} \to R^{n_0 \times n_0}$ be a linear map, and let $A \in R^{n \times n}$ where $n = n_0^k$. The arithmetic complexity of computing $\psi(A)$ is*

$$F_\psi(n) = \frac{q}{n_0^2} n^2 \cdot \log_{n_0} n,$$

where q is the number of linear operations performed by ψ.

Claim 3.9. *Let R be a ring and let $\psi : R^{n_0 \times m_0} \to R^{n_0 \times m_0}$ be a linear map, and let $A \in R^{n \times m}$ where $n = n_0^k$, $m = m_0^k$. The IO-complexity of computing $\psi(A)$ is*

$$IO_\psi(n, m, M) \leq \frac{3q}{n_0 m_0} nm \cdot \log_{n_0 m_0} \left(2 \cdot \frac{nm}{M}\right) + 4nm,$$

where q is the number of linear operations performed by ψ.

Proof. Each step of the recursion consists of computing $n_0 \cdot m_0$ sub-problems and performs q linear operations. The base case occurs when the problem fits entirely in the fast memory (or local memory in parallel setting), namely, $2n^2 \leq M$. Each addition requires a maximum of three data transfers (one for each input and one for writing the output). Hence, a basis transformation that performs q linear operations at each recursive step has the recurrence

$$IO_\psi(n, M) \leq \begin{cases} n_0 m_0 IO_\psi\left(\frac{n}{n_0}, \frac{m}{m_0}, M\right) + 3q \cdot \left(\frac{n}{n_0}\frac{m}{m_0}\right) & 2nm > M \\ 2M & otherwise. \end{cases}$$

Therefore,

$$\begin{aligned} IO_\psi(n, m, M) &\leq n_0 m_0 IO_\psi\left(\frac{n}{n_0}, \frac{m}{m_0}, M\right) + 3q\left(\frac{nm}{n_0 m_0}\right) \\ &= \sum_{k=0}^{\log_{n_0 m_0}(\frac{nm}{\frac{M}{2}})-1} (n_0 m_0)^k \left(3q \frac{nm}{(n_0 m_0)^{k+1}}\right) + 2M \cdot (n_0 m_0)^{\log_{n_0 m_0}(\frac{nm}{\frac{M}{2}})} \\ &= \frac{3q}{n_0 m_0} nm \cdot \sum_{k=0}^{\log_{n_0 m_0}\left(2 \cdot \frac{nm}{M}\right)-1} \left(\frac{n_0 m_0}{n_0 m_0}\right)^k + 4M \cdot \frac{nm}{M} \\ &= \frac{3q}{n_0 m_0} nm \cdot \log_{n_0 m_0}\left(2 \cdot \frac{nm}{M}\right) + 4nm. \end{aligned}$$

□

Corollary 3.10. *Let R be a ring and let $\psi : R^{n_0 \times n_0} \to R^{n_0 \times n_0}$ be a linear map, and let $A \in R^{n \times n}$ where $n = n_0^k$. The IO-complexity of computing $\psi(A)$ is*

$$IO_\psi(n, M) \leq \frac{3q}{n_0^2} n^2 \cdot \log_{n_0}\left(\sqrt{2} \cdot \frac{n}{\sqrt{M}}\right) + 4n^2,$$

where q is the number of linear operations performed by ψ.

3.2 Computing Matrix Multiplication in Alternative Basis

Claim 3.11. *Let ϕ, ψ, υ be automorphisms of $R^{n_0 \times m_0}, R^{m_0 \times k_0}, R^{n_0 \times k_0}$ (respectively), and let* ALG *be an $\langle n_0, m_0, k_0; t\rangle_{\phi,\psi,\upsilon}$ algorithm. For any $A \in R^{n \times m}$, $B \in R^{m \times k}$:*

$$ALG(\phi^\ell(A), \psi^\ell(B)) = \upsilon^\ell(A \cdot B),$$

where $n = n_0^\ell$, $m = m_0^\ell$, $k = k_0^\ell$.

Proof. Denote $\tilde{C} = ALG(\phi^{\ell+1}(A), \psi^{\ell+1}(B))$ and the encoding/decoding matrices of ALG by $\langle U, V, W\rangle$. We prove by induction on ℓ that $\tilde{C} = v^{\ell}(A \cdot B)$. For $r \in [t]$, denote

$$S_r = \sum_{i\in[n_0], j\in[m_0]} U_{r,(i,j)}(\phi(A))_{i,j},$$
$$T_r = \sum_{i\in[m_0], j\in[k_0]} V_{r,(i,j)}(\psi(B))_{i,j}.$$

The base case, $\ell = 1$, holds by Lemma 3.2 since ALG is an $\langle n_0, m_0, k_0; t\rangle_{\phi,\psi,v}$-algorithm. Note that this means that for any $i \in [n_0]$, $j \in [k_0]$

$$(v(AB))_{i,j} = (W^T((U \cdot \phi(A)) \odot (V \cdot \psi(B))))_{i,j} = \sum_{r\in[t]} W_{r,(i,j)}(S_r \cdot T_r).$$

Next, we assume the claim holds for $\ell \in \mathbb{N}$ and show for $\ell + 1$. Given input $\tilde{A} = \phi^{\ell+1}(A)$, $\tilde{B} = \psi^{\ell+1}(B)$, ALG performs t multiplications $P_1, \ldots, P_t$. For each multiplication P_r, its left-hand side multiplicand is of the form

$$L_r = \sum_{i\in[n_0], j\in[m_0]} U_{r,(i,j)}\tilde{A}_{i,j}.$$

By Definition 3.4, $(\phi^{\ell+1}(A))_{i,j} = \phi^{\ell}(\phi(A))_{i,j}$. Hence,

$$= \sum_{i\in[n_0], j\in[m_0]} U_{r,(i,j)}(\phi^{\ell}(\phi(A))_{i,j}).$$

From linearity of ϕ^{ℓ}

$$= \phi^{\ell}\left(\sum_{i\in[n_0], j\in[m_0]} U_{r,(i,j)}(\phi(A))_{i,j}\right)$$
$$= \phi^{\ell}(S_r).$$

And similarly, the right-hand multiplication R_r is of the form

$$R_r = \psi^{\ell}(T_r).$$

Note that for any $r \in [t]$, S_r, T_r are $n_0^{\ell} \times m_0^{\ell}$ and $m_0^{\ell} \times k_0^{\ell}$ matrices, respectively. Hence, by the induction hypothesis,

$$P_r = ALG(\phi^{\ell}(S_r), \psi^{\ell}(T_r)) = v^{\ell}(S_r \cdot T_r).$$

Each entry in the output $\tilde{C}$ is of the form

$$\tilde{C}_{i,j} = \sum_{r\in[t]} W_{r,(i,j)}P_r = \sum_{r\in[t]} W_{r,(i,j)}v^{\ell}(S_r \cdot T_r).$$

By linearity of v^{ℓ}

$$= v^{\ell}\left(\sum_{r\in[t]} W_{r,(i,j)}(S_r \cdot T_r)\right).$$

And, as noted in the base case,

$$(v(A \cdot B))_{i,j} = \left(\sum_{r\in[t]} W_{r,(i,j)}(S_r \cdot T_r)\right)_{(i,j)}.$$

Hence,

$$\tilde{C}_{i,j} = v^{\ell}(v(A \cdot B))_{i,j}.$$

Therefore, by Definition 3.4, $\tilde{C} = v^{\ell+1}(A \cdot B)$. □

We now present an analysis of the complexity of alternative basis matrix multiplication. For simplicity, we present the square matrix multiplication case. See Section 4 for the general case.

Notation 3.12. *When discussing an $\langle n_0, n_0, n_0; t\rangle_{\phi,\psi,v}$-algorithm, we denote $\omega_0 = \log_{n_0} t$.*

Claim 3.13. *Let ALG be an $\langle n_0, n_0, n_0; t\rangle_{\phi,\psi,v}$-algorithm which performs q linear operations at its base case. The arithmetic complexity of ALG is*

$$F_{ALG}(n) = \left(1 + \frac{q}{t - n_0^2}\right) n^{\omega_0} - \left(\frac{q}{t - n_0^2}\right) n^2.$$

Proof. Each step of the recursion consists of computing t sub-problems and performs q linear operations (additions/multiplication by scalar) of sub-matrices. Therefore, $F_{ALG}(n) = tF_{ALG}(\frac{n}{n_0}) + q(\frac{n}{n_0})^2$ and $F_{ALG}(1) = 1$. Thus,

$$\begin{aligned}
F_{ALG}(n) &= \sum_{k=0}^{\log_{n_0} n-1} t^k \cdot q \cdot \left(\frac{n}{n_0^{k+1}}\right)^2 + t^{\log_{n_0} n} \cdot F_{ALG}(1) \\
&= \frac{q}{n_0^2} n^2 \cdot \sum_{k=0}^{\log_{n_0} n-1} \left(\frac{t}{n_0^2}\right)^k + n^{\omega_0} \\
&= \frac{q}{n_0^2} n^2 \left(\frac{\left(\frac{t}{n_0^2}\right)^{\log_{n_0} n} - 1}{\frac{t}{n_0^2} - 1}\right) + n^{\omega_0} q \left(\frac{n^{\omega_0} - n^2}{t - n_0^2}\right) + n^{\omega_0}.
\end{aligned}$$

□

Claim 3.14. *Let ALG be an $\langle n_0, n_0, n_0; t\rangle_{\phi,\psi,v}$-algorithm which performs q linear operations at its base case. The IO-complexity of ALG is*

$$IO_{ALG}(n, M) \leq \left(3 + \frac{3q}{2\left(t - n_0^2\right)}\right) \cdot M \left(\sqrt{2} \cdot \frac{n}{\sqrt{M}}\right)^{\omega_0} - 3\left(\frac{q}{t - n_0^2}\right) n^2,$$

where $\omega_0 = \log_{n_0} t$.

Proof. Each step of the recursion consists of computing t sub-problems and performs q linear operations. The base case occurs when the problem fits entirely in the fast memory (or local memory in parallel setting), namely, $2n^2 \leq M$. Each addition requires a maximum of three data transfers (one of each input and one for writing the output). Hence, a recursive bi-linear algorithm that performs q linear operations at each recursive step has the recurrence

$$IO_{ALG}(n, M) \leq \begin{cases} t \cdot IO_{ALG}\left(\frac{n}{n_0}, M\right) + 3q \cdot \left(\frac{n}{n_0}\right)^2 & 2n^2 > M \\ 3M & otherwise. \end{cases}$$

$$\left\langle U_{opt},\, V_{opt},\, W_{opt} \right\rangle =$$

$$\left\langle \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ -1 & 1 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & -1 & 0 & -1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \right\rangle$$

Fig. 1. $\langle U_{opt}, V_{opt}, W_{opt} \rangle$ are the encoding/decoding matrices of our $\langle 2,2,2;7 \rangle_{\psi_{opt}}$-algorithm which performs 12 linear operations.

Therefore,

$$\begin{aligned} IO_{ALG}(n,M) &\leq \sum_{k=0}^{\log_{n_0} \frac{n}{\sqrt{\frac{M}{2}}}-1} t^k \cdot 3q\left(\frac{n}{n_0^{k+1}}\right)^2 + 3M \cdot t^{\log_{n_0}\left(\frac{n}{\sqrt{\frac{M}{2}}}\right)} \\ &= \frac{3q}{n_0^2} n^2 \sum_{k=0}^{\log_{n_0} \frac{n}{\sqrt{\frac{M}{2}}}-1} \left(\frac{t}{n_0^2}\right)^k + 3M\left(\sqrt{2}\frac{n}{\sqrt{M}}\right)^{\omega_0} \\ &= \frac{3q}{n_0^2} n^2 \left(\frac{\left(\frac{t}{n_0^2}\right)^{\log_{n_0} \frac{n}{\sqrt{\frac{M}{2}}}} - 1}{\frac{t}{n_0^2} - 1} \right) + 3M \cdot \left(\sqrt{2}\frac{n}{\sqrt{M}}\right)^{\omega_0} \\ &= q\left(\frac{\frac{3M}{2}\left(\sqrt{2}\cdot\frac{n}{\sqrt{M}}\right)^{\omega_0} - 3n^2}{t - n_0^2} \right) + 3M \cdot \left(\sqrt{2}\frac{n}{\sqrt{M}}\right)^{\omega_0} \\ &= \left(3 + \frac{3q}{2\left(t - n_0^2\right)}\right) \cdot M \left(\sqrt{2}\cdot\frac{n}{\sqrt{M}}\right)^{\omega_0} - 3\left(\frac{q}{t-n_0^2}\right) n^2. \end{aligned}$$

□

Remark 3.15. The above IO-complexity analysis holds for any recursive bilinear algorithm. Note, however, that the complexity of a given algorithm can sometimes be further improved via reuse of read or computed blocks. For example, in both encodings of our optimal $\langle 2,2,2;7 \rangle_{\psi_{opt}}$-algorithm (Figure 1), all three additions are sums of $A_{1,2}$ ($B_{1,2}$ in the second encoding) and another sub-matrix, hence, we can reuse the same sub-block, only reading the second summand into memory. We chose to present the general case (i.e., with no reuse) for an easier comparison between algorithms.

Corollary 3.16. *If* ALG *(Algorithm 1) performs q linear operations at the base case, then its arithmetic complexity is*

$$F_{ALG}(n) = \left(1 + \frac{q}{t - n_0^2}\right) n^{\omega_0} - \left(\frac{q}{t-n_0^2}\right) n^2 + O(n^2 \log n).$$

Proof. The number of flops performed by the algorithm is the sum of (1) the number of flops performed by the basis transformations (denoted ϕ, ψ, υ) and (2) the number of flops performed by the recursive bilinear algorithms *ALG*.

$$F_{ALG}(n) = F_{ALG}(n) + F_{\phi}(n) + F_{\psi}(n) + F_{v}(n).$$

The result follows immediately from Claim 3.8 and Claim 3.13. □

Corollary 3.17. *If* ALG *(Algorithm 1) performs q linear operations at the base case, then its IO-complexity is*

$$IO_{ALG}(n, M) \leq \left(3 + \frac{3q}{2(t - n_0^2)}\right) \cdot M \left(\sqrt{2} \cdot \frac{n}{\sqrt{M}}\right)^{\omega_0} - 3\left(\frac{q}{t - n_0^2}\right) n^2 + O\left(n^2 \log \frac{n}{\sqrt{M}}\right).$$

Proof. The IO-complexity is the sum of (1) the IO-complexity of the recursive bilinear algorithms *ALG* and (2) the IO-complexity of the basis transformations (denoted ϕ, ψ, v).

$$IO_{ALG}(n, M) = IO_{ALG}(n, M) + IO_{\phi}(n, M) + IO_{\psi}(n, M) + IO_{v}(n, M).$$

The result follows immediately from Claim 3.10 and Claim 3.14. □

3.3 Optimal ⟨2, 2, 2 ; 7⟩-Algorithm

We now present a basis transformation $\psi_{opt} : R^4 \to R^4$ and an $\langle 2, 2, 2; 7\rangle_{\psi}$-algorithm which performs only 12 linear operations.

Notation 3.18. *Let, ψ_{opt} refer to the following transformation:*

$$\psi_{opt} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & 0 & 1 \end{pmatrix}, \quad \psi_{opt}^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & -1 & 1 & 1 \end{pmatrix}.$$

For convenience, when applying ψ to matrices, we omit the vectorization and refer to the term as $\psi : R^{2\times 2} \to R^{2\times 2}$:

$$\psi_{opt}(A) = \psi_{opt}\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,1} & A_{1,2} - A_{2,1} + A_{2,2} \\ A_{21} - A_{2,2} & A_{1,2} + A_{2,2} \end{pmatrix},$$

where $A_{i,j}$ can be ring elements or sub-matrices. ψ_{opt}^{-1} is defined analogously. Both ψ_{opt} and ψ_{opt}^{-1} extend recursively as in Definition 3.4.

Claim 3.19. *$\langle U_{opt}, V_{opt}, W_{opt}\rangle$ are encoding/decoding matrices of an $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm.*

Proof. Observe that

$$\left\langle U_{opt} \cdot \psi_{opt}, V_{opt} \cdot \psi_{opt}, W_{opt} \cdot \psi_{opt}^{-T} \right\rangle = \left\langle \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 1 & -1 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 1 & -1 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & -1 & -1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & -1 & -1 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix} \right\rangle.$$

It is easy to verify that $\langle U_{opt} \cdot \psi_{opt}, V_{opt} \cdot \psi_{opt}, W_{opt} \cdot \psi_{opt}^{-T}\rangle$ satisfy the triple product condition in Fact 2.4. Hence, they are the encoding/decoding algorithm of an $\langle 2, 2, 2; 7\rangle$-algorithm. By Corollary 3.3, the claim follows. □

Claim 3.20. *Let R be a ring, and let $A \in R^{n\times n}$ where $n = 2^k$. The arithmetic complexity of computing $\psi_{opt}(A)$ is*

$$F_{\psi_{opt}}(n) = n^2 \log_2 n.$$

The same holds for computing $\psi_{opt}^{-1}(A)$.

Proof. Both ψ_{opt}, ψ_{opt}^{-1} perform $q = 4$ linear operations at each recursive step and has a base case size of $n_0 = 2$. The lemma follows immediately from Claim 3.8. □

Claim 3.21. *Let R be a ring, and let $A \in R^{n\times n}$ where $n = 2^k$. The I/O-complexity of computing $\psi_{opt}(A)$ is*

$$IO_{\psi_{opt}}(n, M) \leq 3n^2 \cdot \log_2\left(\sqrt{2} \cdot \frac{n}{\sqrt{M}}\right) + 4n^2.$$

Proof. Both ψ_{opt}, ψ_{opt}^{-1} perform $q = 4$ linear operations at each recursive step. The lemma follows immediately from Claim 3.10 with base case $n_0 = 2$. □

Corollary 3.22. *Our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm's arithmetic complexity is $F_{opt}(n) = 5n^{\log_2 7} - 4n^2$.*

Proof. Our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm has a 2×2 base case and performs seven multiplications. Applying Fact 2.6 to its encoding/decoding matrices $\langle U_{opt}, V_{opt}, W_{opt}\rangle$, we see that it performs 12 linear operations. The result follows immediately from Claim 3.13. □

Corollary 3.23. *Our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm's IO-complexity is*

$$IO_{opt}(n, M) \leq 9M\left(\sqrt{3} \cdot \frac{n}{\sqrt{M}}\right)^{\log_2 7} - 12n^2.$$

Proof. Our algorithm has a 2×2 base case and performs seven multiplications. By applying Fact 2.6 to its encoding/decoding matrices (as shown in Figure 1), we see that it performs 12 linear operations. The result follows immediately from Claim 3.14. □

Corollary 3.24. *The arithmetic complexity of alternative basis matrix multiplication (Algorithm 1) with our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm is*

$$F(n) = 5n^{\log_2 7} - 4n^2 + 3n^2 \log_2 n.$$

Proof. The proof is similar to that of Corollary 3.16. □

Corollary 3.25. *The IO-complexity of alternative basis matrix multiplication (Algorithm 1) with our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm is*

$$IO(n, M) \leq 9M\left(\sqrt{3} \cdot \frac{n}{\sqrt{M}}\right)^{\log_2 7} + 9n^2 \cdot \log_2\left(\sqrt{2} \cdot \frac{n}{\sqrt{M}}\right).$$

Proof. The proof is similar to that of Corollary 3.17. □

Theorem 3.26. *Our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm's sequential and parallel IO-complexity is bound by $\Omega((\frac{n}{\sqrt{M}})^{\omega_0} \frac{M}{P})$ (where P is the number of processors, 1 in the sequential case, and $\omega_0 = \log_2 7$).*

Proof. We refer to the undirected bipartite graph defined by the decoding matrix of our Strassen-like algorithm as its decoding graph (i.e., the edge (i, j) exists if $W_{i,j} \neq 0$). In [7], Ballard et al. proved that for any square recursive-bilinear Strassen-like algorithm with $n_0 \times n_0$ base case which performs t multiplications, that if the decoding graph is connected, then these bounds apply with $\omega_0 = \log_{n_0} t$. The decoding graph of our algorithm is connected. Hence, the claim is true. □

4 COMPOSING ALTERNATIVE BASIS MATRIX MULTIPLICATION ALGORITHMS

In this section,we apply our method to compositions of algorithms. That is, we show how to obtain alternative basis matrix multiplication algorithms consisting of any sequence of alternative basis matrix multiplication algorithms rather than repeatedly using the same base case.

One reason to compose algorithms is that the asymptotically fastest matrix multiplication algorithm may not always be optimal for small matrices. For example, the classical algorithm performs fewer computations for small matrix dimensions [3]. Another reason to compose algorithms is to better fit matrix sizes. Consider multiplying 6×6 matrices as an example. One way to solve the problem is by performing one step of an $\langle 2, 2, 2; 7\rangle$-algorithm, followed by a step of Laderman's $\langle 3, 3, 3; 23\rangle$-algorithm. Alternatively, the matrix may be padded with zeroes to round its size up to the nearest power of 2, then use Strassen-Winograd's algorithm on the resulting matrix, or peeling the matrix, removing a row or column to make it fit a power of two, accounting specially for the removed row or column afterwards; both of these last options result in more multiplications than the composition of fast matrix multiplication algorithms. Furthermore, many implementations of fast matrix multiplication algorithms take several fast recursive steps followed by a switch to the classical algorithm from a vendor-tuned library with extensive built-in optimizations, which improves their performance.

Definition 4.1 (Composition of Alternative Basis Matrix Multiplication Algorithms). Let R be a ring, let ALG_1 be an $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}$-algorithm, and ALG_2 be an $\langle n_2, m_2, k_2; t_2\rangle_{\phi_2,\psi_2,\upsilon_2}$-algorithm. We denote a fast matrix multiplication algorithm which takes $\phi_1 \circ \phi_2(A)$, $\psi_1 \circ \psi_2(B)$ (where $A \in R^{n_1 n_2 \times m_1 m_2}$, $B \in R^{m_1 m_2 \times k_1 k_2}$) as inputs and outputs $\upsilon_1 \circ \upsilon_2(A \cdot B)$ by computing one step of ALG_1's recursive bilinear algorithm, followed by one step of ALG_2's recursive bilinear algorithm by $ALG_1 \circ ALG_2$. See Definition 4.2 for composition of basis transformations.

Definition 4.2. Let R be a ring, let $\phi_1 : R^{n_1 \times m_1} \to R^{n_1 \times m_1}$, $\phi_2 : R^{n_2 \times m_2} \to R^{n_2 \times m_2}$ be linear maps, and let $A \in R^{n \times m}$ (where $n = n_1 \cdot n_2$, $m = m_1 \cdot m_2$). Define the composition $\phi_1 \circ \phi_2 : R^{n \times m} \to R^{n \times m}$ by $(\phi_1 \circ \phi_2(A)) = \phi_1(\tilde{A})$, where $\tilde{A}_{i,j} = \phi_2(A_{i,j})$ ($A_{i,j}$ are block sub-matrices).

As in the single algorithm case, all basis transformations are applied in the first and last phases, and the bilinear algorithms are applied in the middle phase.

We next define, prove, and analyze composite alternative basis matrix multiplication algorithms. In Section 4.1, we show how to compose basis transformations alternative basis algorithms and analyze their complexity. In Section 4.2, we analyze the complexity of the bilinear part of such algorithms, and present a correctness proof.

Claim 4.3 (Composition of Alternative Basis Matrix Multiplication Algorithms). *Let ALG_1 be an $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}$-algorithm, and let ALG_2 be an $\langle n_2, m_2, k_2; t_2\rangle_{\phi_2,\psi_2,\upsilon_2}$-algorithm. $ALG_1 \circ ALG_2$ is an $\langle n_1 n_2, m_1 m_2, k_1 k_2; t_1 t_2\rangle_{\phi_1\circ\phi_2,\psi_1\circ\psi_2,\upsilon_1\circ\upsilon_2}$-algorithm.*

Corollary 4.4. *Let $ALG_1, \ldots, ALG_\ell$ be $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}, \ldots, \langle n_\ell, m_\ell, k_\ell; t_\ell\rangle_{\phi_\ell,\psi_\ell,\upsilon_\ell}$-algorithms. Then $ALG_1 \circ \ldots \circ ALG_\ell$ is an $\langle \prod_{i=1}^{\ell} n_i, \prod_{i=1}^{\ell} m_i, \prod_{i=1}^{\ell} k_i; \prod_{i=1}^{\ell} t_i\rangle_{\phi_1\circ\ldots\circ\phi_\ell,\psi_1\circ\ldots\circ\psi_\ell,\upsilon_1\circ\ldots\circ\upsilon_\ell}$-algorithm.*

4.1 Composing Fast Basis Transformations

A composite alternative basis algorithm requires a composite basis transformation $\psi_1 \circ \ldots \psi_\ell$ (some of which can be the identity). In this section, we generalize our basis transformation (Definition 3.4) in order to make it applicable to the composition of basis transformations.

Example 4.5. Let R be a ring, $A \in R^{6\times 6}$, and let $\phi_1 : R^{2\times 2} \to R^{2\times 2}$, $\phi_2 : R^{3\times 3} \to R^{3\times 3}$ be linear maps, then

$$\phi_1 \circ \phi_2(A) = \phi_1 \begin{pmatrix} \phi_2(A_{1,1}) & \phi_2(A_{1,2}) & \phi_2(A_{1,3}) \\ \phi_2(A_{2,1}) & \phi_2(A_{2,2}) & \phi_2(A_{2,3}) \\ \phi_2(A_{3,1}) & \phi_2(A_{3,2}) & \phi_2(A_{3,3}) \end{pmatrix},$$

where $A_{i,j}$ are 2×2 block submatrices.

Claim 4.6. *Let R be a ring, let $\phi_1 : R^{n_1\times m_1} \to R^{n_1\times m_1}$, $\phi_2 : R^{n_2\times m_2} \to R^{n_2\times m_2}$ be invertible linear maps, and let $\phi_1 \circ \phi_2$ as defined above. Then $\phi_1 \circ \phi_2$ is invertible and its inverse is $((\phi_1 \circ \phi_2)^{-1}(A))_{i,j} = \phi_2^{-1}(\phi_1^{-1}(A))_{i,j}$.*

Proof. The proof is similar to that of Claim 3.6.

Define $\tilde{A}$ by $(\tilde{A})_{i,j} = \phi_2(A_{i,j})$ and define $(\phi_1 \circ \phi_2)^{-1}$ by $((\phi_1 \circ \phi_2)^{-1}(A))_{i,j} = \phi_2^{-1}(\phi_1^{-1}(A))_{i,j}$. Then

$$((\phi_1 \circ \phi_2)^{-1}(\phi_1 \circ \phi_2(A)))_{i,j} = \phi_2^{-1}(\phi_1^{-1}(\phi_1 \circ \phi_2(A)))_{i,j}.$$

By Definition 4.2

$$= \phi_2^{-1}(\phi_1^{-1}\phi_1(\tilde{A}))_{i,j} = \phi_2^{-1}(\tilde{A})_{i,j}.$$

By definition of $\tilde{A}$

$$= \phi_2^{-1}(\phi_2(A)_{i,j}) = A_{i,j}. \qquad \square$$

We next present the analysis of the Arithmetic and IO-complexity, for computing a composition of basis transformations.

Claim 4.7. *Let R be a ring, let $\phi_1 : R^{n_1\times m_1} \to R^{n_1\times m_1}$ be a linear map, and let $\phi_2 : R^{n_2\times m_2} \to R^{n_2\times m_2}$ be a linear map. The arithmetic complexity of computing $\phi_1 \circ \phi_2$ is*

$$F_{\phi_1\circ\phi_2}(n,m) = \frac{q_{\phi_1}}{n_1 m_1} nm + n_1 m_1 \cdot F_{\phi_2}\left(\frac{n}{n_1}, \frac{m}{m_1}\right),$$

where q_{ϕ_1} is the number of linear operations performed by ϕ_1.

Proof. The first step is to compute ϕ_1. We then divide the matrix into $n_1 \cdot m_1$ sub-problems of size $\frac{n}{n_1} \times \frac{m}{m_1}$, each of which is a basis transformation, as specified in Claim 4.3. Therefore,

$$\begin{aligned} F_{\phi_1\circ\phi_2}(n,m) &= F_{\phi_1}(n,m) + n_1 m_1 \cdot F_{\phi_2}\left(\frac{n}{n_1}, \frac{m}{m_1}\right) \\ &= \frac{q_{\phi_1}}{n_1 m_1} nm + n_1 m_1 \cdot F_{\phi_2}\left(\frac{n}{n_1}, \frac{m}{m_1}\right). \end{aligned} \qquad \square$$

Corollary 4.8. *Let R be a ring, let $\phi_1 : R^{n_1\times m_1} \to R^{n_1\times m_1}, \ldots, \phi_\ell : R^{n_\ell\times m_\ell} \to R^{n_\ell\times m_\ell}$ be linear maps, and denote the number of linear operations performed by ϕ_i by q_i. The arithmetic complexity of computing $\phi_1 \circ \ldots \circ \phi_\ell$ is*

$$\begin{aligned} F_{\phi_1\circ\ldots\circ\phi_\ell}(n,m) = {} & \frac{q_1 \cdot nm}{n_1 m_1} + n_1 m_1 \cdot \left(\frac{q_2 \cdot nm}{n_1 m_1 n_2 m_2}\right) \\ & + \ldots \\ & + \left(\prod_{i=1}^{\ell-2} n_i m_i\right)\left(\frac{q_{\ell-1} \cdot nm}{\prod_{j=1}^{\ell-1} n_j m_j}\right) \\ & + \left(\prod_{i=1}^{\ell-1} n_i m_i\right) \cdot F_{\phi_\ell}\left(\frac{n}{\prod_{j=1}^{\ell-1} n_j}, \frac{m}{\prod_{j=1}^{\ell-1} m_j}\right). \end{aligned}$$

Corollary 4.9. *Let R be a ring, let $\phi_1 : R^{n_1 \times m_1} \to R^{n_1 \times m_1}$ be a linear map, and let $\phi_2 : R^{n_2 \times m_2} \to R^{n_2 \times m_2}$ be a linear map. The arithmetic complexity of computing $\phi_1^\ell \circ \phi_2$ is*

$$F_{\phi_1^\ell \circ \phi_2}(n,m) = \frac{q_1}{n_1 m_1} nm \cdot \ell + (n_1 \cdot m_1)^\ell \cdot F_{\phi_2}\left(\frac{n}{n_1^\ell}, \frac{m}{m_1^\ell}\right),$$

where q_1 is the number of linear operations performed by ϕ_1.

Claim 4.10. *Let R be a ring, let $\phi_1 : R^{n_1 \times m_1} \to R^{n_1 \times m_1}$ be a linear map, and let $\phi_2 : R^{n_2 \times m_2} \to R^{n_2 \times m_2}$ be a linear map. The IO-complexity of computing $\phi_1 \circ \phi_2$ is*

$$IO_{\phi_1 \circ \phi_2}(n,m,M) \leq \begin{cases} \frac{2q_{\phi_1}}{n_1 m_1} nm + n_1 m_1 \cdot IO_{\phi_2}\left(\frac{n}{n_1}, \frac{m}{m_1}\right) & 2nm > M \\ 2M & otherwise \end{cases},$$

where q_{ϕ_1} is the number of linear operations performed by ϕ_1.

Proof. This proof is similar to the proof of Claim 4.7, accounting for memory size as in Claim 3.10. □

Corollary 4.11. *Let R be a ring, let $\phi_1 : R^{n_1 \times m_1} \to R^{n_1 \times m_1}, \ldots, \phi_\ell : R^{n_\ell \times m_\ell} \to R^{n_\ell \times m_\ell}$ be linear maps, and denote the number of linear operations performed by ϕ_i by q_i. The IO-complexity of computing $\phi_1 \circ \ldots \circ \phi_\ell$ is*

$$\begin{aligned} IO_{\phi_1 \circ \ldots \circ \phi_\ell}(n,m) \leq\ & 2q_1 \cdot \frac{nm}{n_1 m_1} \\ & + n_1 m_1 \cdot \left(2q_2 \cdot \frac{nm}{n_1 m_1 n_2 m_2}\right) \\ & + \ldots \\ & + 2\left(\prod_{i=1}^{\ell-2} n_i m_i\right)\left(\frac{q_{\ell-1} \cdot nm}{\prod_{j=1}^{\ell-1} n_j m_j}\right) \\ & + \left(\prod_{i=1}^{\ell-1} n_i m_i\right) \cdot IO_{\phi_\ell}\left(\frac{n}{\prod_{j=1}^{\ell-1} n_j}, \frac{m}{\prod_{j=1}^{\ell-1} m_j}\right). \end{aligned}$$

Corollary 4.12. *Let R be a ring, let $\phi_1 : R^{n_1 \times m_1} \to R^{n_1 \times m_1}$ be a linear map, and let $\phi_2 : R^{n_2 \times m_2} \to R^{n_2 \times m_2}$ be a linear map. The IO-complexity of computing $\phi_1^\ell \circ \phi_2$ is*

$$IO_{\phi_1^\ell \circ \phi_2}(n,m) \leq 2q_1 \cdot \frac{nm}{n_1 m_1} \cdot \ell + (n_1 \cdot m_1)^\ell \cdot IO_{\phi_2}\left(\frac{n}{n_1^\ell}, \frac{m}{m_1^\ell}\right),$$

where q_1 is the number of linear operations performed by ϕ_1.

4.2 Composing the Bilinear Phases

We next present a correctness proof for composed alternative basis matrix multiplication algorithms, followed by a runtime analysis. The following claim and corollary prove Claim 4.3 and Corollary 4.4, showing that the composition of alternative basis matrix multiplication algorithms is an alternative basis matrix multiplication algorithm.

Claim 4.13. *Let R be a ring, let $\phi_1, \psi_1, \upsilon_1$ be automorphisms of $R^{n_1 \times m_1}, R^{m_1 \times k_1}, R^{n_1 \times k_1}$ (respectively), and let ALG_1 be an $\langle n_1, m_1, k_1; t_1 \rangle_{\phi_1, \psi_1, \upsilon_1}$-algorithm. Furthermore, let $\phi_2, \psi_2, \upsilon_2$ be automorphisms of $R^{n_2 \times m_2}, R^{m_2 \times k_2}, R^{n_2 \times k_2}$ (respectively), and let ALG_2 be an $\langle n_2, m_2, k_2; t_2 \rangle_{\phi_2, \psi_2, \upsilon_2}$-algorithm. For any $A \in R^{n_1 \cdot n_2 \times m_1 \cdot m_2}$, $B \in R^{m_1 \cdot m_2 \times k_1 \cdot k_2}$,*

$$ALG_1 \circ ALG_2(\phi_1 \circ \phi_2(A), \psi_1 \circ \psi_2(B)) = \upsilon_1 \circ \upsilon_2(A \cdot B).$$

Corollary 4.14. *Let* $ALG_1, \ldots, ALG_\ell$ *be* $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}, \ldots, \langle n_\ell, m_\ell, k_\ell; t_\ell\rangle_{\phi_\ell,\psi_\ell,\upsilon_\ell}$*-algorithms. For any* $A \in R^{\prod_{j=1}^{\ell} n_j \times \prod_{j=1}^{\ell} m_j}$, $B \in R^{\prod_{j=1}^{\ell} m_j \times \prod_{j=1}^{\ell} k_j}$,

$$ALG_1 \circ \ldots \circ ALG_\ell(\phi_1 \circ \ldots \circ \phi_\ell(A), \psi_1 \circ \ldots \circ \psi_\ell(B)) = \upsilon_1 \circ \ldots \circ \upsilon_\ell(A \cdot B).$$

Proof of Claim 4.13. The proof is similar to the proof of Claim 3.11.

Denote $\tilde{C} = ALG_1 \circ ALG_2(\phi_1 \circ \phi_2(A), \psi_1 \circ \psi_2(B))$, and the encoding/decoding matrices of ALG_1 by $\langle U, V, W\rangle$. We prove that $\tilde{C} = \upsilon_1 \circ \upsilon_2(A \cdot B)$. For $r \in [t_1]$, denote

$$S_r = \sum_{i\in[n_1], j\in[m_1]} U_{r,(i,j)}(\phi_1(A))_{i,j},$$
$$T_r = \sum_{i\in[m_1], j\in[k_1]} V_{r,(i,j)}(\psi_1(B))_{i,j}.$$

Given input $\tilde{A} = \phi_1 \circ \phi_2(A)$, $\tilde{B} = \psi_1 \circ \phi_2(B)$, ALG_1 performs t_1 multiplications $P_1, \ldots, P_{t_1}$. For each multiplication P_r, its left-hand side multiplicand is of the form

$$L_r = \sum_{i\in[n_1], j\in[m_1]} U_{r,(i,j)}\tilde{A}_{i,j}.$$

By Definition 4.2, $(\phi_1 \circ \phi_2(A))_{i,j} = \phi_2(\phi_1(A))_{i,j}$. Hence,

$$= \sum_{i\in[n_1], j\in[m_1]} U_{r,(i,j)}(\phi_2(\phi_1(A))_{i,j}).$$

By linearity of ϕ_2

$$= \phi_2\left(\sum_{i\in[n_1], j\in[m_1]} U_{r,(i,j)}(\phi_1(A))_{i,j}\right)$$
$$= \phi_2(S_r).$$

And similarly, the right-hand multiplication R_r is of the form

$$R_r = \psi_2(T_r).$$

Note that for any $r \in [t_1]$, S_r, T_r are $n_2 \times m_2$ and $m_2 \times k_2$ matrices, respectively, and ALG_2 is an $\langle n_2, m_2, k_2; t_2\rangle_{\phi_2,\psi_2,\upsilon_2}$-algorithm. Hence,

$$P_r = ALG_2(\phi_2(S_r), \psi_2(T_r)) = \upsilon_2(S_r \cdot T_r).$$

Each entry in the output $\tilde{C}$ is of the form

$$\tilde{C}_{i,j} = \sum_{r\in[t]} W_{r,(i,j)}P_r = \sum_{r\in[t]} W_{r,(i,j)}\upsilon_2(S_r \cdot T_r).$$

By linearity of υ_2

$$= \upsilon_2\left(\sum_{r\in[t_1]} W_{r,(i,j)}S_r \cdot T_r\right).$$

ALG_1 is an $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}$-algorithm. Hence, by Lemma 3.2, for any $i \in [n_1]$, $j \in [k_1]$

$$(\upsilon_1(AB))_{i,j} = (W^T((U \cdot \phi_1(A)) \odot (V \cdot \psi_1(B))))_{i,j} = \sum_{r\in[t]} W_{r,(i,j)}(S_r \cdot T_r).$$

Hence,

$$\tilde{C}_{i,j} = \upsilon_2(\upsilon_1(A \cdot B))_{i,j}.$$

Therefore, by Definition 4.2, $\tilde{C} = \upsilon_1 \circ \upsilon_2(A \cdot B)$. □

Remark 4.15. By Fact 2.1, any bilinear algorithm is associated with encoding/decoding matrices $\langle U, V, W\rangle$. In the following proofs, we use the notations q_U, q_V, q_W to denote the number of linear operations a bilinear algorithm computes at its base case, where q_U, q_V are the number of linear operations used for the encoding (determined by U and V), and q_W is the number of linear operations used in the decoding (determined by W).

Claim 4.16. *Let ALG_1 be an $\langle n_1, m_1, k_1; t_1\rangle$-algorithm which performs q_U, q_V, q_W linear operations at the base case, and let ALG_2 be any matrix multiplication algorithm. The arithmetic complexity of computing $ALG_1 \circ ALG_2$ is*

$$F_{ALG_1\circ ALG_2}(n,m,k) = t_1 \cdot F_{ALG_2}\left(\frac{n}{n_1}, \frac{m}{m_1}, \frac{k}{k_1},\right) + q_U\left(\frac{nm}{n_1m_1}\right) + q_V\left(\frac{mk}{m_1k_1}\right) + q_W\left(\frac{nk}{n_1k_1}\right).$$

Proof. $ALG_1 \circ ALG_2$ performs the q_U, q_V, q_W linear operations of ALG_1, and computes t_1 subproblems using ALG_2. Therefore,

$$F_{ALG_1\circ ALG_2}(n,m,k) = t_1 F_{ALG_2}\left(\frac{n}{n_1}, \frac{m}{m_1}, \frac{k}{k_1},\right) + q_U\left(\frac{nm}{n_1m_1}\right) + q_V\left(\frac{mk}{m_1k_1}\right) + q_W\left(\frac{nk}{n_1k_0}\right). \qquad \square$$

Corollary 4.17. *Let $ALG_1, \ldots, ALG_\ell$ be $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}, \ldots, \langle n_\ell, m_\ell, k_\ell; t_\ell\rangle_{\phi_\ell,\psi_\ell,\upsilon_\ell}$-algorithms, and assume that ALG_i performs q_U^i, q_V^i, q_W^i linear operations at the base case. The arithmetic complexity of computing $ALG_1 \circ \ldots \circ ALG_\ell$ is*

$$\begin{aligned}
F_{ALG_1\circ\ldots\circ ALG_\ell}(n,m,k) = {} & q_U^1\left(\frac{nm}{n_1m_1}\right) + q_V^1\left(\frac{mk}{m_1k_1}\right) + q_W^1\left(\frac{nk}{n_1k_1}\right) \\
& + t_1\left(q_U^2\left(\frac{nm}{n_1n_1n_2m_2}\right) + q_V^2\left(\frac{mk}{m_1k_1m_2k_2}\right) + q_W^2\left(\frac{nk}{n_1k_1n_2k_2}\right)\right) \\
& + \ldots \\
& + \left(\prod_{j=1}^{\ell-2} t_j\right)\cdot\left(q_U^{\ell-1}\left(\frac{nm}{\prod_{j=1}^{\ell-1} n_jm_j}\right) + q_V^{\ell-1}\left(\frac{mk}{\prod_{j=1}^{\ell-1} m_jk_j}\right) + q_W^{\ell-1}\left(\frac{nk}{\prod_{j=1}^{\ell-1} n_jk_j}\right)\right) \\
& + \left(\prod_{j=1}^{\ell-1} t_j\right)\cdot F_{ALG_\ell}\left(\frac{n}{\prod_{j=1}^{\ell-1} n_j}, \frac{m}{\prod_{j=1}^{\ell-1} m_j}, \frac{k}{\prod_{j=1}^{\ell-1} k_j}\right).
\end{aligned}$$

Definition 4.18. Let ALG_1, ALG_2 be $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}, \langle n_2, m_2, k_2; t_2\rangle_{\phi_2,\psi_2,\upsilon_2}$ algorithms, respectively. We denote $\overbrace{ALG_1 \circ \ldots \circ ALG_1}^{\ell \text{ times}} \circ ALG_2$ by $ALG_1^\ell \circ ALG_2$.

Corollary 4.19. *Let ALG_1 be an $\langle n_1, m_1, k_1; t_1\rangle$-algorithm which performs q_U, q_V, q_W linear operations at the base case, and let ALG_2 be any matrix multiplication algorithm. The arithmetic complexity of computing $ALG_1^l \circ ALG_2$ is*

$$\begin{aligned}
F_{ALG_1^l\circ ALG_2}(n,m,k) = {} & \left[\frac{q_Unm}{(t_1-n_1m_1)(n_1m_1)^l} + \frac{q_Vmk}{(t_1-m_1k_1)(m_1k_1)^l} + \frac{q_Wnk}{(t_1-n_1k_1)(n_1k_1)^l}\right] t_1^l \\
& - \left[\left(\frac{q_U}{t_1-n_1m_1}\right)nm + \left(\frac{q_V}{t_1-m_1k_1}\right)mk + \left(\frac{q_W}{t_1-n_1k_1}\right)nk\right] \\
& + t_1^l F_{ALG_2}\left(\frac{n}{n_1^l}, \frac{m}{m_1^l}, \frac{k}{k_1^l}\right).
\end{aligned}$$

Corollary 4.20. *Let ALG_1 be an $\langle n_0, n_0, n_0; t_0\rangle$-algorithm which performs q linear operations at the base case, and let ALG_2 be any matrix multiplication algorithm. The arithmetic complexity of computing $ALG1^l \circ ALG_2$ is*

$$F_{ALG_1^l \circ ALG_2}(n) = \left[\frac{q \cdot n^2}{\left(t_0 - n_0^2\right)\left(n_0^2\right)^l}\right] t_0^l - \left(\frac{q}{t_0 - n_0^2}\right) n^2 + t_0^l F_{ALG_2}\left(\frac{n}{n_0^l}\right).$$

Claim 4.21. *Let ALG_1 be an $\langle n_1, m_1, k_1; t_1\rangle$-algorithm which performs q_U, q_V, q_W linear operations at the base case, and let ALG_2 be any matrix multiplication algorithm. The IO-complexity of computing $ALG_1 \circ ALG_2$ is*

$$IO_{ALG_1 \circ ALG_2}(n, m, k, M) \leq t_1 IO_{ALG_2}\left(\frac{n}{n_1}, \frac{m}{m_1}, \frac{k}{k_1}, M\right) + 3q_U\left(\frac{nm}{n_1 m_1}\right) + 3q_V\left(\frac{mk}{m_1 k_1}\right) + 3q_W\left(\frac{nk}{n_1 k_1}\right).$$

Proof. $ALG_1 \circ ALG_2$ performs the q_U, q_V, q_W linear operations of ALG_1, each of which incurs at most three cache misses per operation (two for reading the inputs, and one for writing the output), and computes t_1 subproblems using ALG_2. Therefore,

$$IO_{ALG_1 \circ ALG_2}(n, m, k, M) \leq t_1 F_{ALG_2}\left(\frac{n}{n_1}, \frac{m}{m_1}, \frac{k}{k_0}, M\right) + 3q_U\left(\frac{nm}{n_1 m_1}\right) + 3q_V\left(\frac{mk}{m_1 k_1}\right) + 3q_W\left(\frac{nk}{n_1 k_1}\right).$$

□

Corollary 4.22. *Let $ALG_1, \ldots, ALG_\ell$ be $\langle n_1, m_1, k_1; t_1\rangle_{\phi_1,\psi_1,\upsilon_1}, \ldots, \langle n_\ell, m_\ell, k_\ell; t_\ell\rangle_{\phi_\ell,\psi_\ell,\upsilon_\ell}$-algorithms, and assume that ALG_i performs q_U^i, q_V^i, q_W^i linear operations at the base case. The IO-complexity of computing $ALG_1 \circ \ldots \circ ALG_\ell$ is*

$$\begin{aligned} IO_{ALG_1 \circ \ldots \circ ALG_\ell}(n, m, k, M) \leq\ & 3q_U^1\left(\frac{nm}{n_1 m_1}\right) + 3q_V^1\left(\frac{mk}{m_1 k_1}\right) + 3q_W^1\left(\frac{nk}{n_1 k_1}\right) \\ & + 3t_1\left(q_U^2\left(\frac{nm}{n_1 n_1 n_2 m_2}\right) + q_V^2\left(\frac{mk}{m_1 k_1 m_2 k_2}\right) + q_W^2\left(\frac{nk}{n_1 k_1 n_2 k_2}\right)\right) \\ & + \ldots \\ & + 3\left(\prod_{j=1}^{\ell-2} t_j\right) \cdot \left(q_U^{\ell-1}\left(\frac{nm}{\prod_{j=1}^{\ell-1} n_j m_j}\right) + q_V^{\ell-1}\left(\frac{mk}{\prod_{j=1}^{\ell-1} m_j k_j}\right) + q_W^{\ell-1}\left(\frac{nk}{\prod_{j=1}^{\ell-1} n_j k_j}\right)\right) \\ & + \left(\prod_{j=1}^{\ell-1} t_j\right) \cdot IO_{ALG_\ell}\left(\frac{n}{\prod_{j=1}^{\ell-1} n_j}, \frac{m}{\prod_{j=1}^{\ell-1} m_j}, \frac{k}{\prod_{j=1}^{\ell-1} k_j}, M\right). \end{aligned}$$

Corollary 4.23. *Let ALG_1 be an $\langle n_1, m_1, k_1; t_1\rangle$-algorithm which performs q_U, q_V, q_W linear operations at the base case, and let ALG_2 be any matrix multiplication algorithm. The IO-complexity of computing $ALG_1^l \circ ALG_2$ is*

$$\begin{aligned} IO_{ALG_1^l \circ ALG_2}(n, m, k, M) \leq\ & \left[\frac{3q_U nm}{(t_1 - n_1 m_1)(n_1 m_1)^l} + \frac{3q_V mk}{(t_1 - m_1 k_1)(m_1 k_1)^l} + \frac{3q_W nk}{(t_1 - n_1 k_1)(n_1 k_1)^l}\right] t_1^l \\ & - 3\left[\left(\frac{q_U}{t_1 - n_1 m_1}\right) nm + \left(\frac{q_V}{t_1 - m_1 k_1}\right) mk + \left(\frac{q_W}{t_1 - n_1 k_1}\right) nk\right] \\ & + t_1^l F_{ALG_2}\left(\frac{n}{n_1^l}, \frac{m}{m_1^l}, \frac{k}{k_1^l}, M\right). \end{aligned}$$

Corollary 4.24. *Let ALG_1 be an $\langle n_0, n_0, n_0; t_0\rangle$-algorithm which performs q linear operations at the base case, and let ALG_2 be any matrix multiplication algorithm. The IO-complexity of computing $ALG1^l \circ ALG_2$ is*

$$IO_{ALG_1^l \circ ALG_2}(n, M) \leq 3\left[\frac{q \cdot n^2}{(t_0 - n_0^2)(n_0^2)^l}\right] t_0^l - \left(\frac{q}{t_0 - n_0^2}\right) n^2 + t_0^l IO_{ALG_2}\left(\frac{n}{n_0^l}, M\right).$$

5 BASIS-INVARIANT LOWER BOUND ON ADDITIONS FOR 2 × 2 MATRIX MULTIPLICATION

In this section, we prove Theorem 1.2 which says that 12 additions are necessary to compute 2×2 matrix multiplication recursively with a base case of 2×2 and seven multiplications, irrespective of basis transformations. Theorem 1.2 completes the lower bound of Probert [52] which says that for standard basis, 15 additions are required, when multiplying matrices over $\mathbb{F}_2$. Our proof is based on the Lower Bound of Bshouty [16]. We present an additional version of this proof, based on Probert's method [52] in Appendix A.

Proof (of Theorem 1.2). Corollary 5.3 shows that each encoding/decoding matrix must contain at least 10 non-zero entries. By Remark 2.6, the number of additions performed is

$$nnz(U) - rows(U) + nnz(V) - rows(V) + nnz(W) - cols(W).$$

Hence, 12 additions are necessary for a $\langle 2, 2, 2; 7\rangle_{\phi,\psi,\upsilon}$-algorithm irrespective of basis transformations ϕ, ψ, υ. □

Lemma 5.1. *[16] Let $\langle U, V, W\rangle$ be the encoding matrices of a $\langle 2, 2, 2; 7\rangle$-algorithm, then the rows of U are pairwise independent. The same holds for V and for W.*

Corollary 5.2. *Let R be a ring and let ϕ, ψ, $\upsilon : R^{2\times 2} \to R^{2\times 2}$ be invertible linear transformations. Let $\langle U, V, W\rangle$ be the encoding matrices of a $\langle 2, 2, 2; 7\rangle_{\phi,\psi,\upsilon}$-algorithm, then the rows of U are pairwise independent. The same holds for V and for W.*

Proof. By Corollary 3.3, $\langle U\phi, V\psi, W\upsilon^{-T}\rangle$ are encoding/decoding matrices of a $\langle 2, 2, 2; 7\rangle$-algorithm. Assume, by contradiction, that U contains two rows which are pairwise dependent. By injectivity of ϕ, $U\phi$ also contains two pairwise dependent rows. Then, $\langle U\phi, V\psi, W\upsilon^{-T}\rangle$ is a $\langle 2, 2, 2; 7\rangle$-algorithm with two pairwise dependent rows in $U\phi$, in contradiction to Lemma 5.1. □

Corollary 5.3. *Let $\langle U, V, W\rangle$ be the encoding matrices of a $\langle 2, 2, 2; 7\rangle$-algorithm; then U, V, and W have at least 10 non-zero entries each.*

Proof. Let R be a ring, and denote the standard basis of R^4 by $\{e_i\}$ (where the i'th entry is 1 and all other entries are 0). By Corollary 5.2, for any $1 \leq i \leq 4$, U can have at most one row of the form αe_i (for $\alpha \in R$). Hence, U can have at most four rows with a single non-zero entry, and the other three rows must have at least two non-zero entries each. Thus, U contains at least 10 non-zero entries. The same holds for V and W. □

6 OPTIMAL ALTERNATIVE BASES

To apply our alternative basis method to other fast matrix multiplication algorithms, it is necessary to find bases that reduce the number of linear operations performed by the algorithm. As we mentioned in Fact 2.6, the non-zero entries of the encoding/decoding matrices determine the number of linear operations performed by an algorithm. Hence, desirable encoding/decoding matrices

should be as sparse as possible, and ideally have entries restricted to −1, 0, or 1. From Lemma 3.2 and Corollary 3.3, we see that any $\langle n, m, k; t\rangle$-algorithm and dimension compatible basis transformations ϕ, ψ, υ, can be composed into an $\langle n, m, k; t\rangle_{\phi,\psi,\upsilon}$-algorithm. Therefore, the problem of finding a basis in which a Strassen-like algorithm performs the least amount of linear operations is equivalent, up to a factor of 2, to the Matrix Sparsification problem.

Problem 6.1. *Matrix Sparsification Problem (MS): Let U be an $m \times n$ matrix of full rank, find an invertible matrix A such that*

$$A = \underset{A \in GL_n}{\texttt{argmin}}(nnz(UA)).$$

That is, finding basis transformations for a fast matrix multiplication algorithm is similar to solving three independent MS problems. Unfortunately, MS is not only NP-Hard [48] to solve, but also NP-Hard to approximate to within a factor of $2^{\log^{\cdot 5-o(1)} n}$ [31] (over $\mathbb{Q}$, assuming NP does not admit quasi-polynomial time deterministic algorithms). There seem to be very few heuristics for matrix sparsification (e.g., [1, 18, 30, 49]), or algorithms that solve the problem under very limiting assumptions (e.g., [2, 35]). Nevertheless, for existing fast matrix multiplication algorithms with small base cases, the use of search heuristics to find bases which significantly sparsify the encoding/decoding matrices of several Strassen-like algorithms has proved useful. Our resulting alternative basis fast matrix multiplication algorithms are summarized in Table 2. Note, particularly, our alternative basis version of Smirnov's $\langle 6, 3, 3; 40\rangle$-algorithm, which is asymptotically faster than Strassen's, where we have reduced the number of linear operations in the bilinear-recursive algorithm from 1,246 to 198, thus reducing the leading coefficient by 83.17%. See Appendix B for the algorithm's encoding/decoding matrices and the alternative bases.

7 IMPLEMENTATION AND FURTHER APPLICATIONS

7.1 Performance Experiments

We next present performance results for our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm. All the experiments were conducted on a single compute node of HLRS's Hazel Hen, with two 12-core (24 threads) Intel Xeon CPU E5-2680 v3 and 128GB of memory. We used a straightforward implementation of both our algorithm and Strassen-Winograd's [61] algorithm using OpenMP. Each algorithm runs for a pre-selected number of recursive steps before switching to Intel's MKL DGEMM routine. Each DGEMM call uses all threads, and matrix additions are always fully parallelized. All results are the median obtained from six experiments.

Figure 2 shows that our algorithm outperforms Strassen-Winograd's, with the margin of improvement increasing with each recursive step and nearing the theoretical asymptotic improvement.

8 DISCUSSION AND FUTURE WORK

Our method yields novel variants of existing fast matrix multiplication algorithms, reducing the number of linear operations required. Our $\langle 2, 2, 2; 7\rangle_{\psi_{opt}}$-algorithm outperforms Strassen-Winograd's algorithm for any matrix dimension $n \geq 32$. Furthermore, we have obtained an alternative basis algorithm of Smirnov's $\langle 6, 3, 3; 40\rangle$-algorithm, that reduces the leading coefficient by 83.17%. While the problem of finding bases which optimally sparsify an algorithm's encoding/decoding matrices is NP-Hard (see Section 6), it is still solvable for many fast matrix multiplication algorithms with small base cases. Hence, in practice, basis transformations could be found by using search heuristics, leading to further improvements.

Strassen [58] observed that when recursion gets to a sufficiently small dimension m, switching to classical matrix multiplication improves the leading coefficient, with a leading coefficient smaller than 4.7 for $m = 16$. Fischer and Probert [28, 29] applied this to Strassen-Winograd's algorithm,

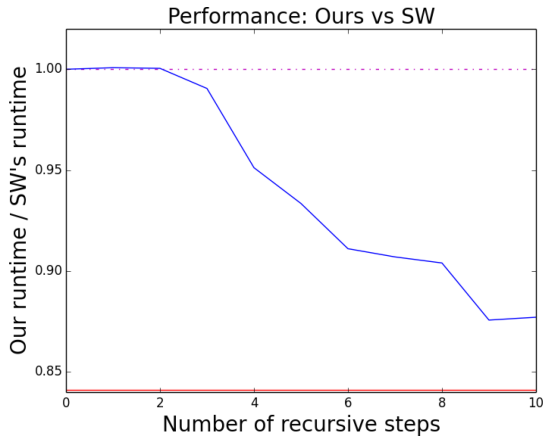

Fig. 2. Comparing the performance of our $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm to that of Strassen-Winograd's on square matrices of fixed dimension $N = 32{,}768$. The graph shows our algorithm's runtime, normalized by that of Strassen-Winograd's, as a function of the number of recursive steps taken before switching to Intel's MKL DGEMM. The top horizontal line (at 1) represents Strassen-Winograd's performance and the bottom horizontal line (at 0.83) represents the theoretical ratio of arithmetic complexities when taking the maximal number of recursive steps.

obtaining a minimal leading coefficient of approximately 3.73 for $m = 8$. We use Corollary 4.20, to compute the leading coefficient, with similar switch to classic matrix multiplication, of our $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm, yielding a leading coefficient of approximately 3.545 for $m = 8$, and a minimal leading coefficient of approximately 3.53 for $m = 6$, both smaller than that of Strassen-Winograd's.

Alternative Bases for Repeated Squaring and Chain Matrix Multiplication. Repeated squaring and chain matrix multiplication using alternative basis matrix multiplication yield a special case. Here, basis transformations may sometimes be omitted between consecutive multiplications, thus reducing the number of times the $O(n \log n)$ overhead is incurred. Bodrato [14] pointed out that given k matrices of size 2×2, $A_1, \ldots, A_k$, the partial products $R_m = \prod_{i=1}^{m}$ can be represented in alternative basis (which he referred to as intermediate representation). This requires that the input basis of the left multiplicand be the same as the basis of the previous iteration's output (e.g., chain multiplication using an $\langle n_0, n_0, n_0; t_0\rangle_{\phi,\psi,\phi}$-algorithm). For example, in our optimal $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm, all three bases used are the same. Naively, multiplying k matrices requires $3k$ basis transformations. However, this can also be achieved with only one basis transformation for each input matrix, and the inverse basis transformation is then used only once, after the last matrix multiplication. Hence, multiplying k matrices of size 2×2 can be accomplished using $k + 1$ basis transformations.

A second special case is matrix squaring. When both of the algorithm's input bases are the same (i.e., an $\langle n_0, n_0, n_0; t_0\rangle_{\phi,\phi,\upsilon}$-algorithm), the input only has to be transformed once. A third special case is repeated squaring. If all three bases are the same one, transformations between repeated iterations may be omitted. Therefore, repeated squaring utilizing alternative basis algorithms where all three bases are identical (e.g., our optimal $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm) require only two basis transformations: one before the first multiplication, and one at the end.

Finally, there is another special case of matrix squaring. When performing matrix squaring, it is possible that some of the partial sums used for left-hand multiplicands are also used for the right-hand multiplicands. For example, both our optimal $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm and Bodrato's algorithm [14], have left- and right-hand multiplicands that are identical, up to ordering (this can

be seen by the fact that our encoding matrices U, V are identical, up to a row permutation). When using such algorithms for squaring, we can save additions by computing the repeating sums once (in our $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm, this saves three additions). However, not all recursive multiplications are squares, yielding the recursion

$$F_{SQ}(n) = 4F_{SQ}\left(\frac{n}{2}\right) + 3F_{Mul}\left(\frac{n}{2}\right) + 9\left(\frac{n}{2}\right)^2,$$

whose solution is $F_{SQ}(n) = 5n^{\log_2 7} - 4n^2 - \frac{3}{4}n^2 \log_2 n$, which improves the low order terms, but not the leading coefficient. Bodrado [14] suggested a method to reduce the leading coefficient of repeated squaring from 5 to 4 by means of fairly robust bookkeeping. This is only possible for certain algorithms (e.g., Bodrato's $\langle 2,2,2;7\rangle$-algorithm and our $\langle 2,2,2;7\rangle_{\psi_{opt}}$-algorithm), and also increases memory footprint and communication costs.

Large scale implementations. Both kernels of our alternative basis algorithms (the basis transformation and the recursive-bilinear algorithms) are known to be highly parallelizable recursive divide-and-conquer algorithms. Furthermore, both kernels admit various communication minimizing parallelization techniques (e.g., [5, 47]). Therefore, alternative basis algorithms can replace the current fast matrix multiplication algorithm kernels both of single nodes (where processors have shared memory), and of distributed matrix multiplication libraries.

Alternative basis matrix algorithms reduce the cache misses. Thus, replacing local (single node) fast matrix multiplication with alternative basis multiplication saves both floating point operations and memory IO bandwidth. However, in distributed implementations, the inter-processor communication costs are also determined by the number of sub-problems to be distributed and these are not reduced by our alternative basis method (since it does not affect the number of multiplications performed). Replacing fast matrix multiplication with an alternative basis matrix multiplication algorithm thus saves floating point operations, which are performed locally by BFS-DFS algorithms [5, 45, 47], at the expense of a small, asymptotically negligible, increase in inter-processor communication. This tradeoff is expected to be particularly beneficial for certain machines and input parameters. However, when performing several subsequent DFS steps (which do not incur inter-processor communication), replacing fast matrix multiplication with alternative basis matrix multiplication saves both arithmetic and communication costs.

Numeric stability. Fast matrix multiplication and other numerical linear algebra algorithms that utilize matrix multiplication (e.g., LU decomposition, linear equation solving, least square problems) are known to be norm-wise numerically stable [4, 13, 24, 25, 34]. Therefore, replacing classical matrix multiplication (which is element-wise numerically stable) with fast matrix multiplication, improves the performance of many algorithms, without losing norm-wise stability [13]. Similarly, both the basis transformations and the bilinear part of alternative basis matrix multiplication preserve norm-wise stability, and hence can replace fast matrix multiplication in numerical linear algebra algorithms, while maintaining the same stability guarantees.

There are a number of techniques known to improve numerical stability of fast matrix multiplication algorithms, including diagonal scaling of input and output matrices [4, 27, 34]. Diagonal scaling methods are directly applicable to alternative basis matrix multiplication in the same way as fast matrix multiplication (as outlined in [4]). It may therefore be beneficial to perform diagonal scaling after basis transformation. Analysis of the exact effects of diagonal scaling on alternative basis matrix multiplication remains outside the scope of this article.

Future research. Very recently, Beniamini and Schwartz [9] presented an extension of the basis transformation method, where they used injective linear maps rather than basis transformations. Their decomposed matrix multiplication method improves the leading coefficients of several fast matrix multiplication algorithms beyond what is possible with basis transformations, but at the

cost of additional low order terms. Most notably, they obtained an optimal leading coefficient of 2 for several algorithms, including $\langle 3, 3, 3; 23\rangle$ matrix multiplication. For some fast matrix multiplication algorithms, however, their new method does not improve the leading coefficient further, and the current approach is provably optimal. In a follow-up work [8], we applied the approaches of this article and of [9] to existing fast matrix multiplication algorithms using novel sparsification heruristics, resulting in several improvements for fast matrix multiplication algorithms.

APPENDIXES

A ALTERNATIVE PROOF OF BASIS-INVARIANT LOWER BOUND ON ADDITIONS FOR 2×2 MATRIX MULTIPLICATION

In this section, we prove Theorem 1.2 which says that 12 additions are necessary to compute 2×2 matrix multiplication recursively with a base case of 2×2 and seven multiplications, irrespective of basis transformations. Theorem 1.2 completes the lower bound of Probert [52] which says that for standard basis, 15 additions are required, when multiplying matrices over $\mathbb{F}_2$.

Definition A.1. Denote by $P_{I\times J}$ the permutation matrix which swaps row-order for column-order of vectorization of an $I \times J$ matrix.

Lemma A.2. *[37] Let $\langle U, V, W\rangle$ be the encoding/decoding matrices of an $\langle m, k, n; t\rangle$-algorithm. Then $\langle WP_{n\times m}, U, VP_{n\times k}\rangle$ are encoding/decoding matrices of an $\langle n, m, k; t\rangle$-algorithm.*

We use the following results, shown by Hopcroft and Kerr [36].

Lemma A.3. *[36] If an algorithm for 2×2 matrix multiplication has k left (right) and side multiplicands from the set $S = \{A_{1,1}, (A_{1,2} + A_{2,1}), (A_{1,1} + A_{1,2} + A_{2,1})\}$, where additions are done modulo 2, then it requires at least $6 + k$ multiplications.*

Corollary A.4. *[36] Lemma A.3 also applies for the following definitions of S:*

(1) $(A_{1,1} + A_{2,1}), (A_{1,2} + A_{2,1} + A_{2,2}), (A_{1,1} + A_{1,2} + A_{2,2})$.
(2) $(A_{1,1} + A_{1,2}), (A_{1,2} + A_{2,1} + A_{2,2}), (A_{1,1} + A_{2,1} + A_{2,2})$.
(3) $(A_{1,1} + A_{1,2} + A_{2,1} + A_{2,2}), (A_{1,2} + A_{2,1}), (A_{1,1} + A_{2,2})$.
(4) $A_{2,1}, (A_{1,1} + A_{2,2}), (A_{1,1} + A_{2,1} + A_{2,2})$.
(5) $(A_{2,1} + A_{2,2}), (A_{1,1} + A_{1,2} + A_{2,2}), (A_{1,1} + A_{1,2} + A_{2,1})$.
(6) $A_{1,2}, (A_{1,1} + A_{2,2}), (A_{1,1} + A_{1,2} + A_{2,2})$.
(7) $(A_{1,2} + A_{2,2}), (A_{1,1} + A_{2,1} + A_{2,2}), (A_{1,1} + A_{1,2} + A_{2,1})$.
(8) $A_{2,2}, (A_{1,2} + A_{2,1}), (A_{1,2} + A_{2,1} + A_{2,2})$.

Corollary A.5. *Any 2×2 matrix multiplication algorithm where a left-hand (or right-hand) multiplicand appears at least twice (modulo 2) requires eight or more multiplications.*

Proof. Immediate from Lemma A.3 and Corollary A.4 since it covers all possible linear sums of matrix elements, modulo 2. □

Fact A.6. *A simple counting argument shows that any 7×4 binary matrix with less than 10 nonzero entries has a duplicate row (modulo 2) or an all-zero row.*

Lemma A.7. *Irrespective of basis transformations ϕ, ψ, υ, the encoding matrices U, V, of an $\langle 2, 2, 2; 7\rangle_{\phi,\psi,\upsilon}$-algorithm contain no duplicate rows.*

Proof. Let $\langle U, V, W\rangle$ be encoding/decoding matrices of an $\langle 2, 2, 2; 7\rangle_{\phi,\psi,\upsilon}$-algorithm. By Corollary 3.3, $\langle U\phi, V\psi, W\upsilon^{-T}\rangle$ are encoding/decoding matrices of a $\langle 2, 2, 2; 7\rangle$-algorithm. Assume, w.l.o.g., that U contains a duplicate row, which means that $U\phi$ contains a duplicate row as well. In

that case, $\langle U\psi, V\psi, Wv^{-T}\rangle$ is a $\langle 2,2,2;7\rangle$-algorithm in which one of the encoding matrices contains a duplicate row (modulo 2), in contradiction to Corollary A.5. □

Lemma A.8. *Irrespective of basis transformations ϕ, ψ, v, the encoding matrices U, V, of an $\langle 2,2,2;7\rangle_{\phi,\psi,v}$-algorithm have at least 10 non-zero entries.*

Proof. No row in U, V can be zeroed out since otherwise the algorithm would require less than seven multiplications. The result then follows from Fact A.6 and Lemma A.7. □

Lemma A.9. *Irrespective of basis transformations ϕ, ψ, v, the decoding matrix W of an $\langle 2,2,2;7\rangle_{\phi,\psi,v}$-algorithm has at least 10 non-zero entries.*

Proof. Let $\langle U, V, W\rangle$ be encoding/decoding matrices of an $\langle 2,2,2;7\rangle_{\phi,\psi,v}$-algorithm and suppose by contradiction that W has less than 10 non-zero entries. W is a 7×4 matrix. By Fact A.6, it either has a duplicate row or an all- zero row. Corollary 3.3 states that $\langle U\phi, V\psi, Wv^{-T}\rangle$ are encoding/decoding matrices of a $\langle 2,2,2;7\rangle$-algorithm. By Lemma A.2, because $\langle U\phi, V\psi, Wv^{-T}\rangle$ define a $\langle 2,2,2;7\rangle$-algorithm, so does $\langle Wv^{-T}P_{2\times 2}, U\phi, V\psi P_{2\times 2}\rangle$. Hence, $Wv^{-T}P_{2\times 2}$ is an encoding matrix of a $\langle 2,2,2;7\rangle$-algorithm which has duplicate or all-zero rows (modulo 2), contradicting Corollary A.5. □

Proof (of Theorem 1.2). Lemmas A.8 and A.9 then show that each encoding/decoding matrix must contain at least 10 non-zero entries. By Remark 2.6, the number of additions performed is

$$nnz(U) - rows(U) + nnz(V) - rows(V) + nnz(W) - cols(W).$$

Hence, 12 additions are necessary for an $\langle 2,2,2;7\rangle_{\phi,\psi,v}$-algorithm irrespective of basis transformations ϕ, ψ, v. □

B ALTERNATIVE BASIS MATRIX MULTIPLICATION ALGORITHMS

This section contains the basis transformations and encoding/decoding matrices of the alternative basis algorithms presented in Table 2. While the problem of finding optimal alternative bases remains NP-Hard (see Section 6), these cases were small enough to obtain improvements via simple search heuristics.

B.1 Alternative Basis $\langle 3,3,2;15\rangle_{\phi,\psi,v}$-Algorithm

In this section, we present the basis transformations and encoding/decoding matrices of our alternative basis $\langle 3,3,2;15\rangle_{\phi,\psi,v}$ variant of Smirnov's $\langle 3,3,2;15\rangle$-algorithm [56]. In this case, the encoding/decoding matrices of the alternative basis algorithm are

U									V						W					
0	0	−1	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	−1
0	0	−1	0	0	0	0	−1	0	0	0	0	−1	1	0	1	0	0	0	0	0
−1	1	0	0	0	0	0	1	0	0	0	−1	−1	0	0	1	0	0	0	−1	0
0	0	1	0	0	1	−1	0	0	0	−1	0	0	1	1	0	1	0	0	0	0
−1	1	0	0	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	0
1	−1	0	1	−1	0	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0
0	0	−1	0	0	−1	0	0	0	0	1	0	0	0	−1	0	−1	1	0	0	−1
0	0	1	1	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	−1
0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	1	−1	0	1	0	0
0	1	0	0	1	0	0	0	0	−1	0	−1	0	0	0	0	0	1	−1	0	0
0	0	1	0	0	0	1	0	1	0	0	0	0	1	1	0	0	0	0	1	0
0	0	1	0	0	1	0	−1	1	0	0	0	0	1	0	0	−1	0	1	0	0
1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	−1	0	1	0	1	0
0	0	0	0	0	0	−1	0	0	1	1	0	0	−1	−1	0	1	0	0	−1	0
0	0	−1	0	−1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

And the basis transformations are

ϕ	ψ	υ
$Id_{9\times9}$	$Id_{6\times6}$	$\begin{matrix} -1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 \\ -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 1 & -1 & 0 & 0 \end{matrix}$

B.2 Alternative Basis $\langle 3,3,3;23\rangle_{\phi,\psi,\upsilon}$-Algorithm

In this section, we present the basis transformations and encoding/decoding matrices of our alternative basis $\langle 3,3,3;23\rangle_{\phi,\psi,\upsilon}$ variant of Smirnov's $\langle 3,3,3;23\rangle$-Algorithm [56] rather than Laderman's original $\langle 3,3,3;23\rangle$-Algorithm. In this case, the encoding/decoding matrices of the alternative basis algorithm are

U									V									W								
−1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	−1	0	0	0	0
1	0	1	0	0	0	1	0	1	0	−1	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	−1	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	−1	0	0	1	0	1	0	0	0	0	1	0	0	−1	0	−1	0	0	1	0	0	0	0	0	0
0	−1	0	0	0	0	0	−1	0	0	0	0	0	1	−1	0	0	0	1	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	−1	0	0	1	0	0	0	0	0	0	−1	0	0	1	0	0	0	0	1	0	0	1	0	0	0
0	0	−1	0	0	1	0	0	−1	0	0	0	0	0	0	0	0	1	0	−1	1	0	0	0	1	0	0
−1	0	0	1	0	0	0	0	0	−1	1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	−1	−1	−1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	−1	0	−1	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	−1
1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0
0	0	−1	0	0	0	0	0	0	1	0	0	0	0	0	−1	0	0	0	0	0	1	0	1	0	0	0
0	0	−1	0	1	1	0	0	0	0	0	0	1	0	0	0	−1	0	0	0	0	0	0	1	0	0	0
0	0	1	1	0	0	0	0	1	0	−1	1	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	−1	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	−1	0	0	−1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	−1
0	0	0	0	0	0	1	−1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	−1
0	0	0	0	0	0	0	−1	0	0	0	1	1	0	1	−1	0	−1	0	0	0	0	0	0	0	0	1
0	0	−1	0	0	0	0	0	−1	0	−1	1	0	0	0	0	1	−1	1	0	0	0	0	0	1	0	0
0	0	0	−1	0	0	0	0	0	0	−1	0	0	0	0	0	1	0	0	0	0	0	−1	0	1	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	−1	0	0	0	0	1	0	0
0	0	−1	−1	0	0	0	0	0	−1	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0

And the basis transformations are

ϕ	ψ	υ
$Id_{9\times9}$	$Id_{9\times9}$	$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & -1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \end{matrix}$

B.3 Alternative Basis $\langle 4, 2, 3; 20\rangle_{\phi,\psi,\upsilon}$-Algorithm

In this section, we present the basis transformations and encoding/decoding matrices of our alternative basis $\langle 4, 2, 3; 20\rangle_{\phi,\psi,\upsilon}$ variant of Smirnov's $\langle 4, 2, 3; 20\rangle$-algorithm [56]. In this case, the encoding/decoding matrices of the alternative basis algorithm are

U								V						W									
0	0	0	0	−1	−1	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	−1	1	0
0	0	0	0	−1	0	−1	0	0	0	−1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	−1	0	0	−1	1	−1	0	−1	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	−1	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
−1	0	0	0	0	1	0	0	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	−1	1	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0	1	0	1	−1	0	−1	0	0	0	0	0	1	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	0	0
0	0	1	1	0	0	0	0	0	−1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0	−1	0	0
−1	0	0	1	0	0	0	0	0	−1	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	−1	1	0	1	0	0	−1	0	−1	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0	0	1	0	1	0	−1	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	0	−1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	−1	0	0	1
0	0	0	−1	0	0	0	−1	0	0	0	−1	−1	0	0	1	0	0	1	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
0	0	0	−1	0	0	0	0	0	−1	1	0	1	−1	0	1	0	0	0	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

And the basis transformations are

ϕ	ψ	υ											
$Id_{8\times 8}$	$Id_{6\times 6}$	0	0	0	0	0	0	0	0	1	0	0	0
		0	0	0	−1	0	0	0	0	0	0	1	0
		0	−1	−1	−1	1	1	0	1	0	0	0	0
		−1	0	0	0	0	0	0	1	0	0	0	0
		0	0	0	1	0	0	0	0	0	0	0	0
		0	0	0	−1	0	0	0	0	−1	0	1	1
		1	0	0	0	0	0	0	0	0	0	0	0
		0	0	0	1	−1	0	0	0	0	0	0	0
		0	0	−1	−1	1	1	0	0	0	0	0	0
		0	0	0	0	0	0	0	0	1	1	−1	−1
		1	0	0	0	0	0	−1	0	1	1	−1	−1
		0	0	0	−1	1	1	0	0	0	0	0	0

B.4 Alternative Basis $\langle 6, 3, 3; 40\rangle_{\phi,\psi,\upsilon}$-Algorithm

In this section, we present the basis transformations and encoding/decoding matrices of our alternative basis $\langle 6, 3, 3; 40\rangle_{\phi,\psi,\upsilon}$ variant of Smirnov's $\langle 6, 3, 3; 40\rangle$-algorithm [56]. In this case, the encoding/decoding matrices of the alternative basis algorithm are

U																		V									W																	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	−1	0	1	−1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	−1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	−1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	−1	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	−1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	−1	1	−1	0	0	0	0	0	−1	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	−1	0	−1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	−1	0	0	0	0	0	0	1	0	0	1	−1	0	0	0	0	0	1	0	0	0	1	0	0	−1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	−1	0	0	0	−1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0
1	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	1	−1	−1	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	−1	0	0	0	0	0	1	0	0	0	0	0	0	−1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	−1	0	−1	0	0	0	0	0
−1	0	1	1	0	0	0	0	−1	0	0	0	0	0	1	0	0	0	0	0	−1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	0	0	−1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	−1
−1	0	0	0	0	0	0	0	0	0	−1	0	1	0	0	0	0	0	1	−1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	−1	0	0	0	0
0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	−1	−1	0	0	0	0	0	−1	0	0	0	0	−1	0	0	0	0	0	0	0	−1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	−1	0	0	0	0	0	0	0	0	0	0	0	0	−1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	−1	0	0	0
1	1	0	0	0	−1	1	0	0	0	0	0	0	1	0	0	0	0	−1	0	−1	0	0	0	0	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	−1	0	0	0	0	0	0	1	1	−1	0	0	0	−1	−1	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	−1	0	0	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
−1	0	0	1	−1	0	0	0	−1	0	0	0	0	0	1	0	0	0	0	0	−1	0	0	0	0	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	−1	0	0	−1	0	1	0	0	0	1	0	0	1	0	−1	0	0	0	0	0	−1	0	0
0	0	0	0	0	1	−1	−1	0	0	0	−1	0	0	0	0	−1	0	−1	0	0	−1	0	1	0	−1	0	0	0	0	−1	0	1	0	0	−1	0	0	−1	0	0	0	0	−1	0
0	0	0	0	−1	0	0	0	0	1	0	0	0	0	0	0	0	−1	0	0	0	0	1	−1	0	1	0	0	0	0	0	1	0	0	0	−1	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	0	−1	0	0	0	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0	−1	0	0	0	0	0	−1	0	−1	0	−1	0	0	0	0
0	1	0	0	0	0	0	0	−1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0	1	0	−1	0	0	−1	1	0	0	0	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1	−1	0	0	0	0	0	0	0	−1	1	1	0	0	0	0	0	−1	1	0	0	0	0	−1	0	0	0	−1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	−1	0
0	0	−1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1	0	1	0	0	0	0	0	0	−1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	0	0	−1	−1	−1	0	0	0	0	0
0	0	0	0	−1	0	0	0	−1	0	−1	1	0	0	0	0	0	−1	0	−1	0	−1	1	0	1	0	0	0	−1	0	0	0	−1	0	0	1	1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	−1	0	0	0	0	0
0	0	0	−1	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	0	1	−1	0	0	0	0	−1	0	0	0	0	0	0	−1	0	0	0	0	1	0	1	0	0	0	0
0	0	0	0	−1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	−1	0	−1	0	0	−1	0	0	0	−1	0	1	0	0	1	0	0	0	0	0	1	0	0

And the basis transformations are

ϕ																		ψ									υ																	
$-\frac{1}{8}$	$-\frac{1}{8}$	0	−1	0	1	0	1	0	0	1	1	0	0	$-\frac{1}{8}$	1	0	0	1	0	1	0	1	1	−1	1	0	$\frac{1}{4}$	0	$\frac{1}{4}$	−2	0	−2	−2	2	0	0	2	2	$\frac{1}{4}$	$-\frac{1}{4}$	0	0	0	0
$\frac{1}{8}$	$\frac{1}{8}$	0	0	1	0	−1	0	1	−1	0	0	0	0	$\frac{1}{8}$	0	−1	−1	1	0	−1	0	1	−1	−1	1	0	0	0	0	−2	2	0	0	0	0	0	0	0	$-\frac{1}{4}$	$\frac{1}{4}$	0	2	−2	0
0	0	$-\frac{1}{8}$	0	1	0	−1	0	1	0	1	1	$-\frac{1}{8}$	$-\frac{1}{8}$	0	1	0	0	−1	0	−1	0	1	−1	1	1	0	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	0	0	2	2	0	0	0	0	0	0	0	−2	−2	0
$-\frac{1}{8}$	$\frac{1}{8}$	0	1	0	−1	0	1	0	0	−1	1	0	0	$-\frac{1}{8}$	1	0	0	−1	0	1	0	−1	−1	1	1	0	0	0	0	−2	0	2	0	0	0	0	0	0	$-\frac{1}{4}$	0	$\frac{1}{4}$	−2	0	2
$-\frac{1}{8}$	$\frac{1}{8}$	0	0	−1	0	−1	0	1	1	0	0	0	0	$-\frac{1}{8}$	0	−1	1	1	0	1	0	1	−1	1	1	0	0	0	0	0	0	0	−2	0	−2	2	0	2	$\frac{1}{4}$	0	$\frac{1}{4}$	0	0	0
0	0	$\frac{1}{8}$	1	0	1	0	1	0	−1	0	0	$-\frac{1}{8}$	$\frac{1}{8}$	0	0	1	1	1	0	1	0	−1	1	1	1	0	0	0	0	−2	2	0	2	0	−2	2	0	−2	0	$\frac{1}{4}$	$-\frac{1}{4}$	2	−2	0
0	0	$\frac{1}{8}$	0	−1	0	−1	0	−1	0	1	1	$-\frac{1}{8}$	$\frac{1}{8}$	0	−1	0	0	−1	0	−1	−1	0	−1	0	0	0	$-\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0	2	−2	0	0	0	0	0	0	0	2	−2	0
$\frac{1}{8}$	$-\frac{1}{8}$	0	1	0	1	0	1	0	0	−1	−1	0	0	$-\frac{1}{8}$	1	0	0	0	0	0	0	−1	1	0	−1	1	0	$-\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0	2	−2	0	0	0	0	0	0	0	−2	2
0	0	$-\frac{1}{8}$	0	1	0	1	0	−1	0	−1	1	$-\frac{1}{8}$	$\frac{1}{8}$	0	1	0	0	0	1	−1	0	1	−1	0	0	0	$-\frac{1}{4}$	0	$\frac{1}{4}$	0	0	0	−2	0	2	0	0	0	0	0	0	−2	0	2
0	0	$-\frac{1}{8}$	1	0	1	0	−1	0	1	0	0	$\frac{1}{8}$	$\frac{1}{8}$	0	0	1	−1										$-\frac{1}{4}$	0	$-\frac{1}{4}$	2	0	2	0	0	0	−2	0	−2	0	0	0	0	0	0
$-\frac{1}{8}$	$-\frac{1}{8}$	0	0	1	0	−1	0	−1	−1	0	0	0	0	$\frac{1}{8}$	0	−1	1										$\frac{1}{4}$	$\frac{1}{4}$	0	2	2	0	0	0	0	2	2	0	0	0	0	0	0	0
0	0	$-\frac{1}{8}$	0	1	0	−1	0	−1	0	1	−1	$\frac{1}{8}$	$\frac{1}{8}$	0	1	0	0										0	0	0	0	2	2	0	0	0	0	0	0	0	$\frac{1}{4}$	$\frac{1}{4}$	0	2	2
$-\frac{1}{8}$	$-\frac{1}{8}$	0	0	1	0	0	1	0	0	1	1	$-\frac{1}{8}$	$-\frac{1}{8}$	0	0	−1	1										0	0	0	2	2	0	−2	−2	0	2	2	0	0	0	0	2	2	0
0	0	$\frac{1}{8}$	1	0	1	1	0	−1	0	−1	−1	0	0	$-\frac{1}{8}$	0	1	1										0	$-\frac{1}{4}$	$-\frac{1}{4}$	0	0	0	0	0	0	0	−2	−2	0	$\frac{1}{4}$	$\frac{1}{4}$	0	2	2
$-\frac{1}{8}$	$-\frac{1}{8}$	0	−1	0	1	1	0	−1	1	0	0	$-\frac{1}{8}$	$\frac{1}{8}$	0	1	0	0										0	$\frac{1}{4}$	$-\frac{1}{4}$	0	0	0	0	0	0	0	−2	2	0	$-\frac{1}{4}$	$\frac{1}{4}$	0	2	−2
$-\frac{1}{8}$	$\frac{1}{8}$	0	−1	0	−1	−1	0	1	1	0	0	$\frac{1}{8}$	$-\frac{1}{8}$	0	−1	0	0										$-\frac{1}{4}$	0	$-\frac{1}{4}$	2	0	2	2	0	2	0	0	0	$-\frac{1}{4}$	0	$-\frac{1}{4}$	0	0	0
$-\frac{1}{8}$	$\frac{1}{8}$	0	−1	0	−1	1	0	1	−1	0	0	$-\frac{1}{8}$	$-\frac{1}{8}$	0	−1	0	0										$-\frac{1}{4}$	0	$\frac{1}{4}$	−2	0	2	−2	0	2	0	0	0	$-\frac{1}{4}$	0	$\frac{1}{4}$	0	0	0
$\frac{1}{8}$	$-\frac{1}{8}$	0	0	1	0	0	−1	0	0	1	−1	$\frac{1}{8}$	$\frac{1}{8}$	0	0	1	−1										0	0	0	2	−2	0	−2	2	0	−2	2	0	0	0	0	−2	2	0

REFERENCES

[1] Ilan Adler, Narendra Karmarkar, Mauricio G. C. Resende, and Geraldo Veiga. 1989. Data structures and programming techniques for the implementation of Karmarkar's algorithm. *ORSA Journal on Computing* 1, 2 (1989), 84–106.

[2] Noga Alon and Raphael Yuster. 2013. Matrix sparsification and nested dissection over arbitrary fields. *Journal of the ACM (JACM)* 60, 4 (2013), 25.

[3] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, C. Bischof, and Danny Sorensen. 1990. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society Press, 2–11.

[4] Grey Ballard, Austin R. Benson, Alex Druinsky, Benjamin Lipshitz, and Oded Schwartz. 2016. Improving the numerical stability of fast matrix multiplication. *SIAM Journal on Matrix Analysis and Applications* 37, 4 (2016), 1382–1418.

[5] 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 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 193–204.

[6] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications* 32, 3 (2011), 866–901.

[7] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2012. Graph expansion and communication costs of fast matrix multiplication. *Journal of the ACM (JACM)* 59, 6 (2012), 32.

[8] Gal Beniamini, Nathan Chen, Olga Holtz, Elaye Karstadt, and Oded Schwartz. 2019. Sparsifying the Operators of Fast Matrix Multiplication Algorithms. Manuscript.

[9] Gal Beniamini and Oded Schwartz. 2019. Faster matrix multiplication via sparse decomposition. In *Proceedings of the 31st ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 11–22.

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

[11] Gianfranco Bilardi and Lorenzo De Stefani. 2017. The I/O complexity of Strassen's matrix multiplication with recomputation. In *Workshop on Algorithms and Data Structures*. Springer, 181–192.

[12] Dario Bini, Milvio Capovani, Francesco Romani, and Grazia Lotti. 1979. O ($n^{2.7799}$) complexity for n× n approximate matrix multiplication. *Information Processing Letters* 8, 5 (1979), 234–235.

[13] Dario Bini and Grazia Lotti. 1980. Stability of fast algorithms for matrix multiplication. *Numerische Mathematik* 36, 1 (1980), 63–72.

[14] Marco Bodrato. 2010. A Strassen-like matrix multiplication suited for squaring and higher power computation. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*. ACM, 273–280.

[15] Richard P. Brent. 1970. *Algorithms for Matrix Multiplication*. Technical Report. Department of Computer Science, Stanford University of California.

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

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

[18] S. Frank Chang and S. Thomas McCormick. 1992. A hierarchical algorithm for making sparse matrices sparser. *Mathematical Programming* 56, 1 (1992), 1–30.

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

[20] James W. Cooley and John W. Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation* 19, 90 (1965), 297–301.

[21] Don Coppersmith and Shmuel Winograd. 1982. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing* 11, 3 (1982), 472–492.

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

[23] Paolo D'Alberto, Marco Bodrato, and Alexandru Nicolau. 2011. Exploiting parallelism in matrix-computation kernels for symmetric multiprocessor systems: Matrix-multiplication and matrix-addition algorithm optimizations by software pipelining and threads allocation. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 2.

[24] James Demmel, Ioana Dumitriu, and Olga Holtz. 2007. Fast linear algebra is stable. *Numerische Mathematik* 108, 1 (2007), 59–91.

[25] James Demmel, Ioana Dumitriu, Olga Holtz, and Robert Kleinberg. 2007. Fast matrix multiplication is stable. *Numerische Mathematik* 106, 2 (2007), 199–224.

[26] Frédéric Desprez and Frédéric Suter. 2004. Impact of mixed-parallelism on parallel implementations of the Strassen and Winograd matrix multiplication algorithms. *Concurrency and Computation: Practice and Experience* 16, 8 (2004), 771–797.

[27] Bogdan Dumitrescu. 1998. Improving and estimating the accuracy of Strassen's algorithm. *Numerische Mathematik* 79, 4 (1998), 485–499.
[28] Patrick C. Fischer. 1974. Further schemes for combining matrix algorithms. In *International Colloquium on Automata, Languages, and Programming*. Springer, 428–436.
[29] Patrick C. Fischer and Robert L. Probert. 1974. Efficient procedures for using matrix algorithms. In *International Colloquium on Automata, Languages, and Programming*. Springer, 413–427.
[30] Craig Gotsman and Sivan Toledo. 2008. On the computation of null spaces of sparse rectangular matrices. *SIAM Journal on Matrix Analysis and Applications* 30, 2 (2008), 445–463.
[31] Lee-Ad Gottlieb and Tyler Neylon. 2010. Matrix sparsification and the sparse null space problem. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Springer, 205–218.
[32] Brian Grayson and Robert Van De Geijn. 1996. A high performance parallel Strassen implementation. *Parallel Processing Letters* 6, 1 (1996), 3–12.
[33] Vince Grolmusz. 2008. Modular representations of polynomials: Hyperdense coding and fast matrix multiplication. *IEEE Transactions on Information Theory* 54, 8 (2008), 3687–3692.
[34] Nicholas J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms*. Vol. 80. SIAM.
[35] Alan J. Hoffman and S. T. McCormick. 1984. A fast algorithm that makes matrices optimally sparse. *Progress in Combinatorial Optimization* (1984), 185–196.
[36] John E. Hopcroft and Leslie R. Kerr. 1971. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM Journal on Applied Mathematics* 20, 1 (1971), 30–36.
[37] John E. Hopcroft and Jean Musinski. 1973. Duality applied to the complexity of matrix multiplications and other bilinear forms. In *Proceedings of the 5th annual ACM Symposium on Theory of Computing*. ACM, 73–87.
[38] Rodney W. Johnson and Aileen M. McLoughlin. 1986. Noncommutative bilinear algorithms for 3*3 matrix multiplication. *SIAM Journal on Computing* 15, 2 (1986), 595–603.
[39] Igor Kaporin. 1999. A practical algorithm for faster matrix multiplication. *Numerical Linear Algebra with Applications* 6, 8 (1999), 687–700.
[40] Igor Kaporin. 2004. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computer Science* 315, 2–3 (2004), 469–510.
[41] Elaye Karstadt and Oded Schwartz. 2017. Matrix multiplication, a little faster. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 101–110.
[42] Bharat Kumar, Chua-Huang Huang, P. Sadayappan, and Rodney W. Johnson. 1995. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming* 4, 4 (1995), 275–289.
[43] Julian Laderman, Victor Y. Pan, and Xuan-He Sha. 1992. On practical algorithms for accelerated matrix multiplication. *Linear Algebra and Its Applications* 162 (1992), 557–588.
[44] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*. ACM, 296–303.
[45] Benjamin Lipshitz, Grey Ballard, James Demmel, and Oded Schwartz. 2012. Communication-avoiding parallel Strassen: Implementation and performance. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 101.
[46] Qingshan Luo and John B. Drake. 1995. A scalable parallel Strassen's matrix multiplication algorithm for distributed-memory computers. In *Proceedings of the 1995 ACM Symposium on Applied Computing*. ACM, 221–226.
[47] William F. McColl and Alexandre Tiskin. 1999. Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24, 3–4 (1999), 287–297.
[48] S. Thomas McCormick. 1983. *A Combinatorial Approach to Some Sparse Matrix Problems*. Technical Report. DTIC Document.
[49] S. Thomas McCormick. 1990. Making sparse matrices sparser: Computational results. *Mathematical Programming* 49, 1–3 (1990), 91–111.
[50] Victor Y. Pan. 1978. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *Proceedings of the19th Annual Symposium on Foundations of Computer Science, 1978*. IEEE, 166–176.
[51] Victor Y. Pan. 1982. Trilinear aggregating with implicit canceling for a new acceleration of matrix multiplication. *Computers & Mathematics with Applications* 8, 1 (1982), 23–34.
[52] Robert L. Probert. 1976. On the additive complexity of matrix multiplication. *SIAM Journal on Computing* 5, 2 (1976), 187–203.
[53] Francesco Romani. 1982. Some properties of disjoint sums of tensors related to matrix multiplication. *SIAM Journal on Computing* 11, 2 (1982), 263–267.
[54] Arnold Schönhage. 1981. Partial and total matrix multiplication. *SIAM Journal on Computing*. 10, 3 (1981), 434–455.

[55] 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*. ACM, 35–45.
[56] Alexey V. Smirnov. 2013. The bilinear complexity and practical algorithms for matrix multiplication. *Computational Mathematics and Mathematical Physics* 53, 12 (2013), 1781–1795.
[57] Andrew J. Stothers. 2010. On the complexity of matrix multiplication. Ph.D Thesis.
[58] Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische Mathematik* 13, 4 (1969), 354–356.
[59] Volker Strassen. 1986. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science, 1986*. IEEE, 49–54.
[60] Virginia V. Williams. 2012. Multiplying matrices faster than Coppersmith-Winograd. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*. ACM, 887–898.
[61] Shmuel Winograd. 1971. On multiplication of 2×2 matrices. *Linear Algebra and Its Applications* 4, 4 (1971), 381–388.

Received August 2018; revised July 2019; accepted September 2019