Matrix Multiplication, a Little Faster

Regular Submission

Elaye Karstadt
The Hebrew University of Jerusalem
elayeek@cs.huji.ac.il

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

ABSTRACT

Strassen’s algorithm (1969) was the first sub-cubic matrix multiplication algorithm. Winograd (1971) improved its complexity by a constant factor. Many asymptotic improvements followed. Unfortunately, most of them have done so at the cost of very large, often gigantic, hidden constants. Consequently, Strassen-Winograd’s $O(n\log_2 7)$ algorithm often outperforms other matrix multiplication algorithms for all feasible matrix dimensions. The leading coefficient of Strassen-Winograd’s algorithm was believed to be optimal for matrix multiplication algorithms with $2 \times 2$ base case, due to a lower bound of Probert (1976).

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 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 prove a generalization of Probert’s lower bound that holds under change of basis, showing that for matrix multiplication algorithms with a $2 \times 2$ base case, the leading coefficient of our algorithm cannot be further reduced, hence optimal. We apply our technique to other Strassen-like algorithms, improving their arithmetic and communication costs by significant constant factors.

CCS CONCEPTS

• Mathematics of computing → Computations on matrices;
• Computing methodologies → Linear algebra algorithms;

KEYWORDS

Fast Matrix Multiplication, Bilinear Algorithms

1 INTRODUCTION

Strassen’s algorithm [37] was the first sub-cubic matrix multiplication algorithm, with complexity $O(n\log_2 7)$. Winograd [40] reduced the leading coefficient from 7 to 6 by decreasing the number of additions and subtractions from 18 to 15. In practice, Strassen-Winograd’s algorithm often performs better than some asymptotically faster algorithms [3] due to these smaller hidden constants. The leading coefficient of Strassen-Winograd’s algorithm was believed to be optimal, due to a lower bound on the number of additions \(^1\) for matrix multiplication algorithms with $2 \times 2$ base case, obtained by Probert [31].

We obtain a method for improving the practical performance of Strassen and Strassen-like fast matrix multiplication algorithms by improving the hidden constants inside the $O$-notation. To this end, we extend Bodrato’s (2010) method for matrix squaring, and transform matrices to an alternative basis.

1.1 Strassen-like Algorithms

Strassen-like algorithms are a class of divide-and-conquer algorithms which utilize a base $(n_0, m_0, k_0; t)$-algorithm: multiplying an $n_0 \times m_0$ matrix by an $m_0 \times k_0$ matrix using $t$ scalar multiplications, where $n_0$, $m_0$, $k_0$, and $t$ 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 multiplication by scalar in the base algorithm are interpreted as block-wise additions. Multiplications in the base algorithm are interpreted as block-wise multiplication via recursion. We refer to a Strassen-like algorithm by its base case. Hence, an $(n, m, k; t)$-algorithm may refer to either the algorithm’s base case or the corresponding block recursive algorithm, as obvious from context.

1.2 Known Strassen-like algorithms

Since Strassen’s original discovery, many fast matrix multiplication algorithms followed and improved the asymptotic complexity [6, 12, 13, 26, 29, 32, 33, 36, 38, 39]. Some of these improvements have come at the cost of very large, often gigantic, hidden constants. Le Gall [26] estimated that even if matrix multiplication could be done in $O\left(n^2\right)$ arithmetic operations, it is unlikely to be applicable as the base case sizes would have to be astronomical.

Recursive fast matrix multiplication algorithms with reasonable base case size for both square and rectangular matrices have been discovered [3, 20, 24, 25, 29, 30, 35]. Thus, they have manageable hidden constants, some of which are asymptotically faster than Strassen’s algorithm. While many fast matrix multiplication algorithms fail to compete with Strassen’s in practice due to their

\(^1\)From here on, when referring to addition and subtraction count we say additions.
hidden constants. However, some have achieved competitive performance (e.g., Kapronin’s [21] implementation of Laderman et al.’s algorithm [24]).

Recently, Smirnov presented several fast matrix multiplication algorithms derived by computer aided optimization tools [35], including an (6, 3, 3; 40)-algorithm with asymptotic complexity of \(O(n^{\log_5 40})\), faster than Strassen’s algorithm. Ballard and Benson [3] later presented several additional fast Strassen-like algorithms, found using computer aided optimization tools as well. They implemented several Strassen-like algorithms, including Smirnov’s (6, 3, 3; 40)-algorithm, on shared-memory architecture in order to demonstrate that Strassen and Strassen-like algorithms can outperform classical matrix multiplication in practice (such as, Intel’s MKL), on modestly sized problems (at least up to \(n=13000\)), in a shared-memory environment. Their experiments also showed Strassen’s algorithm outperforming Smirnov’s algorithm in some of the cases.

1.3 Previous work

Bodrato [7] introduced the intermediate representation method, for repeated squaring and for chain matrix multiplication computations. This enables decreasing the number of additions between consecutive multiplications. Thus, he obtained an algorithm with a \(2 \times 2\) base case, which uses 7 multiplications, and has a leading coefficient of 5 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. While this is not the first time that linear transformations are applied to matrix multiplication, the main focus of previous research on the subject was on improving asymptotic performance rather than reducing the number of additions [10, 17].

Very recently, Cenk and Hasan [8] showed a clever way to apply Strassen-Winograd’s algorithm directly to \(n \times n\) matrices by forsaking the uniform divide-and-conquer pattern of Strassen-like algorithms. 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 performs linear post-computations to calculate the output. Their method enables reuse of sums, resulting in a matrix multiplication algorithm with arithmetic complexity of \(5n^{\log_5 7} + 0.5 \cdot n^{\log_2 6} + 2n^{\log_2 5} - 6.5n^2\). However, this comes at the cost of increased communication costs and memory footprint.

1.4 Our contribution

We present the Alternative Basis Matrix Multiplication method, and show how to apply it to existing Strassen-like algorithms (see Sections 3). While basis transformation is, in general, as expensive as matrix multiplications, some can be performed very fast (e.g., Hadamard in \(O(n^2 \log n)\) using FFT [11]). Fortunately, so is the case for our basis transformation (see Section 3.1). Thus, it is a worthwhile trade-off of reducing the leading coefficient in exchange of an asymptotically insignificant overhead (see Section 3.2). We provide analysis as to how these constants are affected and the impact on both arithmetic and IO-complexity.

We discuss the problem of finding alternative bases to improve Strassen-like algorithms (see Section 5), and present several improved variants of existing algorithms, most notable of which are the alternative basis variant of Strassen’s \((2, 2, 2; 7)\)-algorithm which reduces the number of additions from 15 to 12 (see Section 3.3), and the variant of Smirnov’s \((6, 3, 3; 40)\)-algorithm with leading coefficient reduced by about 83.2\%.

**Theorem 1.1 (Probert’s lower bound).** [31] 15 additions are necessary for any \((2, 2, 2, 7)\)-algorithm.

Our result seemingly contradicts Probert’s lower bound. However, his bound implicitly assumes that the input and output are represented in the standard basis, thus there is no contradiction. We extend Probert’s lower bound to account for alternative bases (see Section 4):

**Theorem 1.2 (Basis invariant lower bound).** 12 additions are necessary for any matrix multiplication algorithm that uses a recursive-bilinear algorithm with a \(2 \times 2\) base case with 7 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.

2 PERLIMINARIES

2.1 The communication bottleneck

Fast matrix multiplication algorithms have lower IO-complexity than the classical algorithm. That is, they communicate asymptotically less data within the memory hierarchy and between processors. The IO-complexity is measured as a function of the number of processors \(P\), the local memory size \(M\), and the matrix dimension \(n\). Namely, the communication costs of a parallel \((n_0, n_0, n_0; t)\)-algorithm are \(\Theta((n_0^2 \log_t t)^{1/2} M/P)\) [1, 2, 4, 34]. Thus, parallel versions of Strassen’s algorithm which minimize communication cost outperform the well tuned classical matrix multiplication in practice, both in shared-memory [3, 14, 23] and distributed-memory architectures [1, 16, 27].

Our \((2, 2, 2, 7)\)-algorithm not only reduces the arithmetic complexity by 16.66\%, but also the IO-complexity by 20\%, compared to Strassen-Winograd’s algorithm. Hence, performance gain should be in range of 16-20\% on a shared-memory machine.

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 \(U \in R^{k \times n}, V \in R^{l \times m}, W \in R^{l \times k}\) 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 element-wise vector product (Hadamard product).

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

---

1Files containing the encoding/decoding matrices and the corresponding basis transformations can be found at https://github.com/elayeek/matmultfaster.
### Table 1: (2, 2, 2; 7)-algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Additions</th>
<th>Arithmetic Computations</th>
<th>I/O-Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strassen [37]</td>
<td>18</td>
<td>$7n \log_2 n - 6n^2$</td>
<td>$6 \cdot \left(\frac{\sqrt{3} \cdot n}{\sqrt{M}}\right) \cdot M - 18n^2 + 3M$</td>
</tr>
<tr>
<td>Strassen-Winograd [40]</td>
<td>15</td>
<td>$6n \log_2 n - 5n^2$</td>
<td>$5 \cdot \left(\frac{\sqrt{3} \cdot n}{\sqrt{M}}\right) \cdot M - 15n^2 + 3M$</td>
</tr>
<tr>
<td>Ours</td>
<td>12</td>
<td>$5n \log_2 n - 4n^2 + 3n^2 \log_2 n$</td>
<td>$4 \cdot \left(\frac{\sqrt{3} \cdot n}{\sqrt{M}}\right) \cdot M - 12n^2 + 3n^2 \cdot \log_2 \left(\frac{\sqrt{2} \cdot n}{\sqrt{M}}\right) + 5M$</td>
</tr>
</tbody>
</table>

### Table 2: Alternative Basis Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Linear Operations</th>
<th>Improved Linear Operations</th>
<th>Arithmetic Leading Coefficient</th>
<th>Improved Leading Coefficient</th>
<th>Computations Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 2, 2, 7) [40]</td>
<td>15</td>
<td>12</td>
<td>6</td>
<td>5</td>
<td>16.6%</td>
</tr>
<tr>
<td>(3, 2, 3, 15) [3]</td>
<td>64</td>
<td>52</td>
<td>15.06</td>
<td>7.94</td>
<td>47.3%</td>
</tr>
<tr>
<td>(2, 3, 4, 20) [3]</td>
<td>78</td>
<td>58</td>
<td>9.96</td>
<td>7.46</td>
<td>25.6%</td>
</tr>
<tr>
<td>(3, 3, 3, 23) [3]</td>
<td>87</td>
<td>75</td>
<td>8.91</td>
<td>6.57</td>
<td>26.3%</td>
</tr>
<tr>
<td>(6, 3, 3, 40) [35]</td>
<td>1246</td>
<td>202</td>
<td>55.63</td>
<td>9.39</td>
<td>83.2%</td>
</tr>
</tbody>
</table>

**Definition 2.3.** Let $R$ be a ring, and let $A \in R^{n \times m}$ a matrix. We denote the vectorization of $A$ by $\bar{A}$. We use the notation $U_{c,(i,j)}$ when referring to the element in the $t’$th row on the column corresponding with the index $(i,j)$ in the vectorization of $A$. For ease of notation, we sometimes write $A_{j,i}$ rather than $\bar{A}_{(i,j)}$.

**Fact 2.4.** (Triple product condition. [22]) Let $R$ be a ring, and let $U \in R^{t \times n \times m}$, $V \in R^{t \times m \times k}$, $W \in R^{t \times n \times k}$. $(U, V, W)$ are encoding/decoding matrices of an $(n, m, k, t)$-algorithm if and only if:

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

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

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

**Notation 2.5.** Denote the number of nonzero entries in a matrix by $\text{nnz}(A)$, and the number of rows/columns by $\text{rows}(A), \text{cols}(A)$.

**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 encoding is:

- $\text{Additions}_U = \text{nnz}(U) - \text{rows}(U)$
- $\text{Additions}_V = \text{nnz}(V) - \text{rows}(V)$

The number of additions performed by the decoding is:

- $\text{Additions}_W = \text{nnz}(W) - \text{cols}(W)$

The number of scalar multiplication performed by each of the encoding/decoding is equal to the total number of matrix entries which are not 1, -1, and 0.

---

3 ALTERNATIVE BASIS MATRIX MULTIPLICATION

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

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

Given a recursive-bilinear, $(n, m, k, t)$-algorithm, $ALG$, alternative basis matrix multiplication works as follows:

**Algorithm 1** Alternative Basis Matrix Multiplication Algorithm

**Input:** $A \in R^{n \times m}, B \in R^{m \times k}$

**Output:** $n \times k$ matrix $C = A \cdot B$

1. function $\text{ABS}(A, B)$
2. $\hat{A} = \phi(A)$  
3. $\hat{B} = \psi(B)$  
4. $\breve{C} = ALG(\hat{A}, \hat{B})$  
5. $C = \upsilon^{-1}(\breve{C})$  
6. return $C$

**Lemma 3.2.** Let $R$ be a ring, let $(U, V, W)$ be the encoding/decoding matrices of an $(n, m, k; t)$-algorithm, and let $\phi, \psi, \upsilon$ be automorphisms of $R^{n \times m}$, $R^{m \times k}$, $R^{n \times k}$ (respectively). $(U \phi^{-1}, V \psi^{-1}, W \upsilon^T)$ are encoding/decoding matrices of an $(n, m, k; t)$-algorithm.
Proof. \(\langle U, V, W \rangle\) are encoding/decoding matrices of an \((n, m, k, t)\)-algorithm. Hence, for any \(A \in R^{n \times m}\), \(B \in R^{m \times k}\)

\[
W^T \left( \left( (U \cdot \tilde{A}) \otimes (V \cdot \tilde{B}) \right) \right) = \tilde{A} \cdot \tilde{B}
\]

Hence,

\[
\psi \left( \tilde{A} \cdot \tilde{B} \right) = \psi \left( W^T \left( (U \cdot \tilde{A}) \otimes (V \cdot \tilde{B}) \right) \right) = \left( W_{\psi} \right)^T \left( (U \psi^{-1} \cdot \psi (\tilde{A}) \otimes V \psi^{-1} \cdot \psi (\tilde{B}) \right)
\]

□

**Corollary 3.3.** Let \(R\) be a ring, and let \(\phi, \psi, v\) be automorphisms of \(R \times m, R \times k, R \times k\) (respectively). \(\langle U, V, W \rangle\) are encoding/decoding matrices of an \((n, m, k, t)\)-algorithm if and only if \(U \psi, V \psi, W \psi^T\) are encoding/decoding matrices of an \((n, m, k, t)\)-algorithm.

### 3.1 Fast basis transformation

**Definition 3.4.** Let \(R\) be a ring and let \(\psi_1 : R^{n_0 \times m_0} \rightarrow R^{n_0 \times m_0}\) be a linear map. We recursively define a linear map \(\psi_{k+1} : R^{n_{k+1} \times m_{k+1}} \rightarrow R^{n_k \times m_k}\) (where \(n = n_0, m = m_0^k\) for some \(\ell_1, \ell_2 \leq k + 1\)) by \(\psi_{k+1}(A)_{i,j} = \psi_k \left( \psi_1(A)_{i,j} \right)\), where \(A_{i,j}\) are \(R^{n_0 \times m_0}\) sub-matrices.

Note that \(\psi_{k+1}\) is a linear map. For convenience, we omit the subscript of \(\psi\) when obvious from context.

**Claim 3.5.** Let \(R\) be a ring, let \(\psi_1 : R^{n_0 \times m_0} \rightarrow R^{n_0 \times m_0}\) be a linear map, and let \(A \in R^{n_{k+1} \times m_{k+1}}\) (where \(n = n_0^k, m = m_0^{k+1}\)). Define \(\hat{A}\) by \(\hat{A}_{i,j} = \psi_k \left( A_{i,j} \right)\). Then \(\psi_1(\hat{A}) = \psi_{k+1}(A)\).

**Proof.** \(\psi_1\) is a linear map. Hence, for any \(i \in [n_0], j \in [m_0]\), \((\psi_1(A))_{i,j}\) is a linear sum of elements of \(A\). Therefore, there exist scalars \(\{x_{r,t}^{(i,j)}\}_{r \in [n_0], t \in [m_0]}\) such that

\[
(\psi_{k+1}(A))_{i,j} = \psi_k \left( \sum_{r,t} x_{r,t}^{(i,j)} A_{r,t} \right)
\]

By linearity of \(\psi_k\),

\[
\sum_{r,t} x_{r,t}^{(i,j)} \psi_k( A_{r,t} ) = (\psi_1(\hat{A}))_{i,j}
\]

□

**Claim 3.6.** Let \(R\) be a ring, let \(\psi_1 : R^{n_0 \times m_0} \rightarrow R^{n_0 \times m_0}\) be an invertible linear map, and let \(\psi_{k+1}\) as defined above. \(\psi_{k+1}\) is invertible and its inverse is \((\psi_{k+1}(A))^{-1} = \tilde{\psi}_k^{-1} \left( \psi_1^{-1}(A) \right)_{i,j}\).

**Proof.** Define \(\tilde{A}\) by \(\tilde{A}_{i,j} = \psi_k (A_{i,j})\) and define \(\tilde{\psi}_k^{-1}\) by \((\tilde{\psi}_k^{-1}(A))_{i,j} = \psi_k^{-1} (\psi_1^{-1}(A))_{i,j}\). Then:

\[
(\tilde{\psi}_k^{-1}(\tilde{\psi}_{k+1}(A)))_{i,j} = \psi_k^{-1} (\psi_1^{-1}(\tilde{\psi}_{k+1}(A)))_{i,j}
\]

By Claim 3.5

\[
= \psi_k^{-1} (\psi_1^{-1}(\psi_1(\hat{A})))_{i,j} = \psi_k^{-1} (\hat{A})_{i,j}
\]

By definition of \(\hat{A}\)

\[
\psi_k^{-1} (\psi_1(\hat{A}))_{i,j} = A_{i,j}
\]

□

We next analyze the arithmetic complexity and IO-complexity of fast basis transformations. For convenience and readability, we presented here the square case only. The analysis for rectangular matrices is similar.

**Claim 3.7.** Let \(R\) be a ring, let \(\psi_1 : R^{n_0 \times m_0} \rightarrow R^{n_0 \times m_0}\) 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

\[
F_{\psi}(n) = \frac{q}{n_0^2} n^2 \log n_0, n
\]

where \(q\) is the number of linear operations performed by \(\psi_1\).

**Proof.** Let \(F_{\psi}(n)\) be the number of additions required by \(\psi\). Each step of the recursion consists of computing \(n_0^2\) sub-problems and performs \(q\) additions of sub-matrices. Therefore, \(F_{\psi}(n) = n_0^2 F_{\psi} \left( \frac{n}{n_0} \right) + q \left( \frac{n}{n_0} \right)^2 \) and \(F_{\psi}(1) = 0\). Thus,

\[
F_{\psi}(n) = \frac{n_0^2}{n_0} F_{\psi} \left( \frac{n}{n_0} \right) + q \left( \frac{n}{n_0} \right)^2
\]

\[
= \frac{n}{n_0} \log n_0(n-1) \left( \frac{n}{n_0} \right)^k \left( \frac{n}{n_0} \right)^2
\]

\[
= \frac{n}{n_0} \log n_0(n-1) \sum_{k=0}^{n_0^2} \left( \frac{n}{n_0} \right)^k = \frac{n}{n_0^2} \log n_0(n)
\]

□

**Claim 3.8.** Let \(R\) be a ring and let \(\psi_1 : R^{n \times n} \rightarrow R^{n \times n}\) 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

\[
IO_{\psi}(n, M) \leq \frac{3q}{n_0^2} n^2 \log n_0 \left( \frac{n}{n_0} \right)^2 + 2M
\]

where \(q\) is the number of linear operations performed by \(\psi_1\).

**Proof.** Each step of the recursion consists of computing \(n_0^2\) 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 at most 3 data transfers (one of each input and one for writing the output). Hence, a basis transformation which performs \(q\) linear operations at each recursive steps has the recurrence:

\[
IO_{\psi}(n, M) \leq \frac{3q}{2M} IO_{\psi} \left( \frac{n}{n_0}, M \right) + 3q \left( \frac{n}{n_0} \right)^2 \quad 2n^2 > M
\]

otherwise
Therefore
\[
IO_{\psi} (n, M) \leq n_0^2 IO_{\psi} \left( \frac{n}{n_0}, M \right) + 3q \left( \frac{n}{n_0} \right)^2
\]
\[
= \log_{n_0} \left( \frac{n}{\sqrt{n}} \right)^{-1} \sum_{k=0}^{n_0/2} \left( \frac{n_0}{n} \right)^k \left( \frac{n}{n_0} \right)^2 + 2M
\]
\[
= \frac{3q}{n_0^2} \log_{n_0} \left( \sqrt{2} \cdot \frac{n}{2} \right)^{-1} \sum_{k=0}^{n_0/2} \left( \frac{n_0}{n_0} \right)^k + 2M
\]
\[
= \frac{3q}{n_0^2} \cdot \log_{n_0} \left( \sqrt{2} \cdot \frac{n}{\sqrt{M}} \right) + 2M
\]
\[\square\]

### 3.2 Computing matrix multiplication in alternative basis

**Claim 3.9.** Let \(\phi_1, \psi_1, v_1\) be automorphisms of \(R^{n \times m_0}, R^{m_0 \times k_0}, R^{n \times k_0}\) (respectively), and let ALG be an \((n_0, m_0, k_0; t)\) \(\phi_1, \psi_1, v_1\) algorithm. For any \(A \in R^{nm}, B \in R^{mk}\) :
\[
ALG (\phi_t (A), \psi_t (B)) = v_t (A \cdot B)
\]
where \(n = n_0^t, m = m_0^t, k = k_0^t\).

**Proof.** Denote \(\tilde{C} = ALG (\phi_{t+1} (A), \psi_{t+1} (B))\) and the encoding/decoding matrices of ALG by \((U, V, W)\). We prove by induction on \(t\) that \(\tilde{C} = v_t (A \cdot B)\). For \(r \in [t]\), denote
\[
S_r = \sum_{i \in [n_0], j \in [m_0]} U_{r, (i, j)} (\phi_1 (A))_{i,j}
\]
\[
T_r = \sum_{i \in [m_0], j \in [k_0]} V_{r, (i, j)} (\psi_1 (B))_{i,j}
\]

The base case, \(\ell = 1\), holds by Lemma 3.2 since ALG is an \((n_0, m_0, k_0; t)\) \(\phi_1, \psi_1, v_1\) algorithm. Note that this means that for any \(i \in [n_0], j \in [k_0]\)
\[
(u_1 (AB))_{i,j} = \left( W^T \left( (U \cdot \phi_1 (A)) \otimes (V \cdot \psi_1 (B)) \right) \right)_{i,j}
\]
\[
= \sum_{r \in [t]} W_{r, (i, j)} (S_r \cdot T_r)
\]

Next, we assume the claim holds for \(\ell \in \mathbb{N}\) and show for \(\ell + 1\). Given input \(\hat{A} = \phi_{t+1} (A)\), \(\hat{B} = \psi_{t+1} (B)\), ALG performs \(t\) multiplications \(P_1, \ldots, P_t\). For each multiplication \(P_r\), its left hand side multiplicand is of the form
\[
L_r = \sum_{i \in [n_0], j \in [m_0]} U_{r, (i, j)} \hat{A}_{i,j}
\]
By Definition 3.4, \((\phi_{t+1} (A))_{i,j} = \phi_t (\phi_1 (A))_{i,j}\). Hence,
\[
= \sum_{i \in [n_0], j \in [m_0]} U_{r, (i, j)} (\phi_t (\phi_1 (A))_{i,j})
\]
From linearity of \(\phi_t\)
\[
= \phi_t \left( \sum_{i \in [n_0], j \in [m_0]} U_{r, (i,j)} (\phi_1 (A))_{i,j} \right)
\]
\[
= \phi_t (S_r)
\]
And similarly, the right hand multiplication \(R_r\) is of the form
\[
R_r = \psi_t (T_r)
\]
Note that for any \(r \in [t]\), \(S_r, T_r\) are \(n_0^t \times m_0^t\) and \(m_0^t \times k_0^t\) matrices, respectively. Hence, by the induction hypothesis,
\[
P_r = ALG (\phi_t (S_r), \psi_t (T_r)) = v_t (S_r \cdot T_r)
\]
Each entry in the output \(\tilde{C}\) is of the form:
\[
\tilde{C}_{i,j} = \sum_{r \in [t]} W_{r, (i,j)} P_r
\]
By linearity of \(v_t\)
\[
= v_t \left( \sum_{r \in [t]} W_{r, (i,j)} (S_r \cdot T_r) \right)
\]
And, as noted in the base case:
\[
(v_1 (A \cdot B))_{i,j} = \left( \sum_{r \in [t]} W_{r, (i,j)} (S_r \cdot T_r) \right)_{(i,j)}
\]
Hence,
\[
\tilde{C}_{i,j} = v_t (v_1 (A \cdot B))_{i,j}
\]
Therefore, by Definition 3.4, \(\tilde{C} = v_{t+1} (A \cdot B)\) \(\square\)

**Notation 3.10.** When discussing an \((n_0, n_0, n_0; t)\) \(\phi, \psi, v\) algorithm, we denote \(\omega_0 = \log_{n_0} t\).

**Claim 3.11.** Let ALG be an \((n_0, n_0, n_0; t)\) \(\phi, \psi, v\) algorithm which performs \(q\) linear operations at its base case. The arithmetic complexity of ALG is
\[
F_{ALG} (n) = \left( 1 + \frac{q}{t - n_0^k} \right) n^{\omega_0} - \left( \frac{q}{t - n_0^k} \right) n^2
\]

**Proof.** Each step of the recursion consists of computing \(t\) subproblems and performs \(q\) linear operations (additions/multiplication by scalar) of sub-matrices. Therefore \(F_{ALG} (n) = t F_{\psi} \left( \frac{n^2}{n_0^t} \right) + q \left( \frac{n_0^t}{n_0} \right)^2\), and \(F_{ALG} (1) = 1\). Thus,
\[
F_{ALG} (n) = \sum_{k=0}^{\log_{n_0} n - 1} t^k \cdot q \cdot \left( \frac{n}{n_0^{k+1}} \right)^2 + t \log_{n_0} n \cdot F_{ALG} (1)
\]
\[
= q \left( \frac{n_0}{n} \right)^2 \sum_{k=0}^{\log_{n_0} n - 1} \left( \frac{t}{n_0} \right)^k + n^{\omega_0}
\]
\[
= q \left( \frac{n_0}{n} \right)^2 \left( \frac{t}{n_0^{\log_{n_0} n}} - 1 \right) + n^{\omega_0} q \left( \frac{n_0^{\omega_0} - n^2}{t - n_0^k} \right) + n^{\omega_0}
\]
\[\square\]
Claim 3.12. Let ALG be an \( (n_0, n_0, n_0; t) \phi, \psi, \upsilon \)-algorithm which performs \( q \) linear operations at its base case. The IO-complexity of ALG is
\[
IO_{ALG} (n, M) \leq \left( \frac{q}{t - n_0^2} \right) \left( M \sqrt{3 \cdot \frac{n}{\sqrt{M}}} \right)^{\frac{n_0}{t}} + 3n^2 + 3M
\]

Proof. Each step of the recursion consists of computing \( n_0^2 \) 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 \( 3n^2 \leq M \). Each addition requires at most 3 data transfers (one of each input and one for writing the output). Hence, a basis transformation which performs \( q \) linear operations at each recursive steps has the recurrence:
\[
IO_{ALG} (n, M) \leq \left( \frac{q}{t - n_0^2} \right) \left( M \sqrt{3 \cdot \frac{n}{\sqrt{M}}} \right)^{\frac{n_0}{t}} + 3n^2 + 3M
\]

Therefore
\[
\log_{n_0} \left( \frac{n}{n_0} \right)^{n_0} - 1 = \sum_{k=0}^{n_0-1} t^k \cdot 3q \left( \frac{n}{n_0+1} \right)^2 + 3M
\]
\[
= 3q \cdot n_0^2 \sum_{k=0}^{n_0-1} \left( \frac{t}{n_0^2} \right)^k + 3M
\]
\[
= 3q \cdot n_0^2 \left( \frac{t}{n_0^2} \log_{n_0} \left( \frac{n}{n_0} \right) - 1 \right) + 3M
\]
\[
= q \left( \frac{M \sqrt{3 \cdot \frac{n}{\sqrt{M}}} \frac{n_0}{t}}{t - n_0^2} \right)^{\frac{n_0}{t}} - 3n^2 + 3M
\]

Corollary 3.13. If ABS (Algorithm 1) performs \( q \) linear operations at the base case, then its arithmetic complexity is
\[
F_{ABS} (n) = \left( 1 + \frac{q}{t - n_0^2} \right) n^{\frac{n_0}{t}} - \left( \frac{q}{t - n_0^2} \right) n^2 + O \left( n^2 \log n \right)
\]

Proof. The number of flops performed by the algorithm is the sum of: (1) the number of flops performed by the basis transformations (denoted \( \phi, \psi, \upsilon \)) and (2) the number of flops performed by the recursive bilinear algorithms ALG.
\[
F_{ABS} (n) = F_{ALG} (n) + F_{\phi} (n) + F_{\psi} (n) + F_{\upsilon} (n)
\]
The result immediately follows from Claim 7 and Claim 3.11.

3.3 Optimal \( (2, 2, 2; 7) \)-algorithm

We now present a basis transformation \( \psi_{\text{opt}} : R^4 \rightarrow R^4 \) and an \( (2, 2, 2; 7) \phi \)-algorithm which performs only 12 linear operations.

Notation 3.15. Let \( \psi_{\text{opt}} \) refer to the following transformation:
\[
\psi_{\text{opt}} = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 \\
0 & 0 & -1 & 1 \\
0 & 1 & 0 & 1 \\
\end{pmatrix}
\]

For convenience, when applying \( \psi \) to matrices, we omit the vectorization and refer to it as \( \psi : R^{2 \times 2} \rightarrow R^{2 \times 2} \)
\[
\psi_{\text{opt}} (A) = \psi_1 \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} = \begin{pmatrix} A_{1,2} & A_{1,1} & A_{1,2} & A_{2,2} \\ A_{2,1} & A_{2,2} & A_{1,1} & A_{2,2} \end{pmatrix}
\]

Where \( A_{i,j} \) can be ring elements or sub-matrices. \( \psi_{\text{opt}} \) is defined analogously. Both \( \psi_{\text{opt}} \) and \( \psi_{\text{opt}}^{-1} \) extend recursively as in Definition 3.4.

Figure 1: \( \langle U, V, W \rangle \) are the encoding/decoding matrices of our \( (2, 2, 2; 7) \psi_{\text{opt}} \)-algorithm which performs 12 linear operations.

Claim 3.16. \( \langle U_{\text{opt}}, V_{\text{opt}}, W_{\text{opt}} \rangle \) are encoding/decoding matrices of an \( (2, 2, 2; 7) \psi_{\text{opt}} \)-algorithm.

Proof. Observe that
\[
\langle U_{\text{opt}}' \cdot \psi_{\text{opt}} \cdot V_{\text{opt}}' \cdot W_{\text{opt}}' \cdot \psi_{\text{opt}}^T \rangle =
\]
\[
\begin{pmatrix}
0 & 1 & 0 & 1 \\
0 & 0 & -1 & 1 \\
0 & 1 & -1 & 1 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 1 & -1 & 0 \\
0 & 1 & -1 & 0 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
\[
\begin{pmatrix}
0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
\[
\begin{pmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}
\]
It is easy to verify that \((U_{\text{opt}} \cdot \psi_{\text{opt}}, V_{\text{opt}} \cdot \psi_{\text{opt}}, W_{\text{opt}} \cdot \psi_{\text{opt}}^T)\) satisfy the triple product condition in Fact 2.4. Hence, they are encoding/decoding algorithm of an \((2, 2, 2, 7)\)-algorithm. By Corollary 3.3, the claim follows.

\[ F_{\psi_{\text{opt}}} (n) = n^2 \log_2 n \]  

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

**Proof.** Both \(\psi_{\text{opt}}, \psi_{\text{opt}}^{-1}\) perform \(q = 4\) linear operations at each recursive step and has base case size of \(n_0 = 2\). The lemma follows immediately from Claim 3.7.

\[ I O_{\psi_{\text{opt}}} (n, M) \leq 2n^2 \log_2 \left( \sqrt{\frac{n}{\sqrt{M}}} \right) + 2M \]

**Proof.** Both \(\psi_{\text{opt}}, \psi_{\text{opt}}^{-1}\) perform \(q = 4\) linear operations at each recursive step. The lemma follows immediately from Claim 3.8 with base case \(n_0 = 2\).

**Corollary 3.19.** Our \((2, 2, 2, 7)\) \(\psi_{\text{opt}}\)-algorithm’s arithmetic complexity is \(F_{\psi_{\text{opt}}} (n) = 5n^2 \log_2 7 - 4n^2\).

**Proof.** Our \((2, 2, 2, 7)\) \(\psi_{\text{opt}}\)-algorithm has a \(2 \times 2\) base case and performs \(7\) multiplications. Applying Fact 2.6 to its encoding/decoding matrices \((U_{\text{opt}}, V_{\text{opt}}, W_{\text{opt}})\), we see that it performs 12 linear operations. The result follows immediately from Claim 3.11.

**Corollary 3.20.** Our \((2, 2, 2, 7)\) \(\psi_{\text{opt}}\)-algorithm’s IO-complexity is

\[ I O_{\psi_{\text{opt}}} (n, M) \leq 12 \cdot \left( \sqrt{3} \cdot M \left( \frac{n}{\sqrt{M}} \right) \log_2 7 - 3n^2 \right) + 3M \]

**Proof.** Our algorithm has a \(2 \times 2\) base case and performs \(7\) 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 3.12.

**Corollary 3.21.** The arithmetic complexity of \(\text{ABS} (\text{Algorithm 1})\) with our \((2, 2, 2, 7)\) \(\psi_{\text{opt}}\)-algorithm is

\[ F_{\text{ABS}} (n) = 5n^2 \log_2 7 - 4n^2 + 3n^2 \log_2 n \]

**Proof.** The proof is similar to that of Corollary 3.13.

**Corollary 3.22.** The IO-complexity of \(\text{ABS} (\text{Algorithm 1})\) with our \((2, 2, 2, 7)\) \(\psi_{\text{opt}}\)-algorithm is

\[ I O_{\text{ALG}} (n, M) \leq 4 \cdot \left( \sqrt{3} \cdot n \left( \frac{n}{\sqrt{M}} \right) \log_2 7 - 12n^2 \right) + 3n^2 \cdot \log_2 \left( \sqrt{2} \cdot \frac{n}{\sqrt{M}} \right) + 5M \]

**Proof.** The proof is similar to that of Corollary 3.14.

**Theorem 3.23.** Our \((2, 2, 2, 7)\) \(\psi_{\text{opt}}\)-algorithm’s sequential and parallel IO-complexity is bound by \(\Omega \left( \left( \frac{n}{\sqrt{M}} \right)^{\log_2 7} \right)\) where \(P\) is the number of processors, \(1\) in the sequential case, and \(c_0 = \log_2 7\).

**Proof.** We refer to the undirected bipartite graph defined by the decoding matrix of our \(\psi\)-algorithm as its decoding graph (i.e., the edge \((i, j)\) exists if \(W_{i,j} \neq 0\)). In [2], 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, the decoding graph is connected then these bounds apply with \(c_0 = \log_2 t\). The decoding graph of our \(\psi\)-algorithm is connected. Hence, the claim is true.

**4 BASIS-IN Variant LOWER BOUND ON ADDITIONS FOR 2 \(\times\) 2 MATRIX MULTIPLICATION**

In this section we prove Theorem 1.2 which says that 12 additions are necessary to compute \(2 \times 2\) matrix multiplication recursively with base case of \(2 \times 2\) and 7 multiplications, irrespective of basis. Theorem 1.2 completes Probert’s lower bound which says that for standard basis, 15 additions are required.

**Definition 4.1.** Denote the permutation matrix which swaps row-order for column-order of vectorization of an \(I \times J\) matrix by \(P_{I \times J}\).

**Lemma 4.2.** [19] Let \((U, V, W)\) be the encoding/decoding matrices of an \((m, k, n, t)\)-algorithm. Then \((WP_{m \times n}, U, VP_{m \times k})\) are encoding/decoding matrices of an \((m, m, k, t)\)-algorithm.

We use the following results, shown by Hopcroft and Kerr [20]:

**Lemma 4.3.** [20] If an algorithm for \(2 \times 2\) matrix multiplication has \(k\) left (right) hand side multiplicands from the set \(S = \{A_{1,1}, A_{1,2} + A_{2,1}, A_{1,1} + A_{2,1} + A_{2,2}\}\), where additions are done modulo 2, then it requires at least \(6 + k\) multiplications.

**Corollary 4.4.** [20] Lemma 4.3 also applies for the following definitions of \(S:\)

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

**Corollary 4.5.** Any \(2 \times 2\) matrix multiplication algorithm where a left hand (right hand) multiplicant appears at least twice (modulo 2) requires \(8\) or more multiplications.

**Proof.** Immediate from Lemma 4.3 and Corollary 4.4 since it covers all possible linear sums of matrix elements, modulo 2.

**Fact 4.6.** A simple counting argument shows that any \(7 \times 4\) binary matrix with less than 10 non-zero entries has a duplicate row (modulo 2) or an all zero row.

**Lemma 4.7.** Irrespective of basis transformations \(\phi, \psi, \nu\), the encoding matrices \(U, V, W\), of an \((2, 2, 2, 7)\) \(\phi, \psi, \nu\)-algorithm contain no duplicate rows.
To apply our alternative basis method to other Strassen-like matrix algorithm performs the least amount of linear operations is closely. Therefore, the problem of finding a basis in which a Strassen-like algorithm consists of three independent MS problems. Unfortunately, MS is not only NP-Hard to solve, but also NP-Hard to approximate within a factor of $2^{\log^* n} n$ [15] (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., [9]), or algorithms under very limiting assumptions (e.g., [18]). Nevertheless, for existing Strassen-like 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 proved useful. Our resulting alternative basis Strassen-like algorithms are summarized in Table 2. Note, particularly, our alternative basis version of Smirnov’s (6, 3, 3, 40)-algorithm, which is asymptotically faster than Strassen’s, where we have reduced the number of linear operations in the binarized-recursive algorithm from 1246 to 202, thus reducing the leading coefficient by 83.2%.

6 IMPLEMENTATION AND PRACTICAL CONSIDERATIONS

6.1 Recursion cutoff point

Implementations of fast matrix multiplication algorithms often take several recursive steps then call the classical algorithm from a vendor-tuned library. This gives better performance in practice due to two main reasons: (1) the asymptotic improvement of Strassen-like algorithms makes them faster than classical algorithms by margins which increase with matrix size, and (2) vendor-tuned libraries have extensive built-in optimization, which makes them perform better than existing implementations of fast matrix multiplication algorithms on small matrices.

We next present theoretical analysis for finding the optimal number of recursive steps without tuning.

CLAIM 6.1. Let $\text{ALG}$ be an $(n, m, n; t)$-algorithm with $q$ linear operations at the base case. The arithmetic complexity of running $\text{ALG}$ for $t$ steps, then switching to classical matrix multiplication is:

$$F_{\text{ALG}} (n, t) = \frac{q}{t - n_0^t} \left( (\frac{n}{n_0^t})^t - 1 \right) n^2 + t^\ell \left( 2 \frac{n}{n_0^t}^3 - \frac{n}{n_0^t}^2 \right)$$

Proof. Each step of the recursion consists of computing $t$ subproblems and performs $q$ linear operations. Therefore, $F_{\text{ALG}} (n, t) = t \cdot F_{\text{ALG}} \left( \frac{n}{n_0}^t \right) + q \left( \frac{n}{n_0}^t \right)^2$ and $F_{\text{ALG}} (n, 0) = 2n^3 - n^2$. Thus

$$F_{\text{ALG}} (n, t) = \sum_{k=0}^{t-1} t^k \cdot q \left( \frac{n}{n_0} \right)^{k+1} + t^\ell \cdot F_{\text{ALG}} \left( \frac{n}{n_0}^t \right)$$

$$= \frac{q}{t - n_0^t} \left( (\frac{n}{n_0^t})^t - 1 \right) \cdot n^2 + t^\ell \left( 2 \frac{n}{n_0^t}^3 - \frac{n}{n_0^t}^2 \right)$$

5 OPTIMAL ALTERNATIVE BASES

To apply our alternative basis method to other Strassen-like matrix multiplication algorithms, we find bases which 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, we want our encoding/decoding matrices to be as sparse as possible, and ideally to have only entries of the form -1, 0, and 1. From Lemma 3.2 and Corollary 3.3 we see that any $(n, m, k; t)$-algorithm and dimension compatible basis transformations $\phi$, $\psi$, $\upsilon$ can be composed into an $(n, m, k; t)$-algorithm. Therefore, the problem of finding a basis in which a Strassen-like algorithm performs the least amount of linear operations is closely tied to the Matrix Sparsification problem:

Problem 5.1. Matrix Sparsiﬁcation Problem (MS): Let $U$ be an $m \times n$ matrix of full rank, find an invertible matrix $A$ such that

$$A = \arg \min_{A \in \text{GL}_n} (\text{nnz} (UA))$$

that is, finding basis transformations for a Strassen-like algorithm consists of three independent MS problems. Unfortunately, MS is not only NP-Hard [28] to solve, but also NP-Hard to approximate within a factor of $2^{\log^* n} n$ [15] (Over $\mathbb{Q}$, assuming NP does not admit quasi-polynomial time deterministic algorithms).
When running an alternative basis algorithm for a limited number of recursive steps, the basis transformation needs to be computed only for the same number of recursive steps. If the basis transformation is computed for more steps than the alternative basis multiplication, the classical algorithm will compute incorrect results as it does not account for the input being represented in an alternative basis. This introduces a small saving in the runtime of basis transformation.

**Claim 6.2.** Let \( R \) be a ring and let \( \psi_k : R^{n_0 \times n_0} \to R^{n_0 \times n_0} \) be an invertible linear map, let \( A \in R^{n \times n} \) where \( n = n_0^2 \), and let \( \ell \leq k \). The arithmetic complexity of computing \( \psi_{\ell}(A) \) is

\[
F_{\psi}(n, \ell) = \frac{q}{n_0^2} \cdot \ell
\]

**Proof.** Let \( F_{\psi_{\ell}}(n) \) be the number of additions required by \( \psi_{\ell} \). Each recursive consists of computing \( n_0^2 \) sub-problems and performing \( q \) linear operations. Therefore, \( F_{\psi}(n, \ell) = n_0^2 \cdot F_{\psi}\left(\frac{n}{n_0}, \ell - 1\right) + q \cdot \left(\frac{n}{n_0}\right)^2 \) and \( F_{\psi}(n, 0) = 0 \).

\[
F_{\psi_{\ell}}(n) = \sum_{k=0}^{\ell-1} \left(n_0^2\right)^k \cdot q \left(\frac{n}{n_0} \right)^{2k} = \frac{q}{n_0^2} \sum_{k=0}^{\ell-1} \left(n_0^2\right)^k = \frac{q}{n_0^2} \cdot \ell
\]

\( \square \)

### 6.2 Performance experiments

We next present performance results for our \((2, 2, 7)\) algorithm. All 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 [40] 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, matrix additions are always fully parallelized. All results are the median over 6 experiments.

In Figure 2 we see that our algorithm outperforms Strassen-Winograd’s, with the margin of improvement increasing with each recursive step and nearing the theoretical improvement.

### 7 DISCUSSION

Our method obtained novel variants of existing Strassen-like algorithms, reducing the number of linear operations required. Our algorithm also outperforms Strassen-Winograd’s algorithm for any matrix dimension \( n \geq 32 \). Furthermore, we’ve obtained an alternative basis algorithm of Smirnov’s \((6, 3, 3; 40)\)-algorithm, reducing the number of additions by 83.8\%. While the problem of finding bases which optimally sparsifies an algorithm’s encoding/decoding matrices is NP-Hard (see Section 5), it is still solvable for many fast matrix multiplication algorithms with small base cases. Hence, finding basis transformations could be done in practice using search heuristics, leading to further improvements.

We leave large scale implementations for future research but note that both kernels of our alternative basis algorithms (basis transformation and recursive-bilinear algorithms) are known to be highly parallelizable recursive divide-and-conquer algorithms, and admit various communication minimizing parallelization techniques (e.g., \([1, 5]\)).

### ACKNOWLEDGMENTS

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. Research is also supported by the Einstein Foundation and the Minerva Foundation. This work was supported by the PetaCloud industry-academia consortium. This research was supported by a grant from the United States-Israel Bi-national Science Foundation (BSF), Jerusalem, Israel. This work was supported by the Huawei 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.
REFERENCES


[29] V Ya Pan. 1978. Strassen’s algorithm is not optimal trilinear technique of ag-


[34] Jacob Scott, Olga Holtz, and Oded Schwartz. 2015. Matrix multiplication I/O-


[37] Volker Strassen. 1969. Gaussian elimination is not optimal. Numerische mathe-

