Methods for Approximating Strassen-like Matrix Multiplication Schema

Gabriel M. Perry

***Abstract*—We explore the parameter space of multiplication schemes of the same form as Strassen's to better understand the landscape of local minima in this over-parameterized space. We do this by performing optimization techniques on the Brent equations. We also explore various algebraic manipulations to further understand the solution manifold.**

***Index Terms*—Brent equations, matrix multiplication, optimization, Strassen's algorithm**

I. Introduction

Matrix multiplication is one of the most commonly performed operations in computing, especially in graphics and big data. As such, much research is devoted to the topic; many procedures attempt to parallelize computations, while others take advantage of certain exceptional cases, such as rectangular [1] or sparse matrices [2]. One particular approach stands out among these as truly unique; in 1969, Volker Strassen showed that any matrix multiplication can be performed in $O(n^{2.8074})$ field operations [3], a surprising improvement in the $O(n^3)$ naïve method. Schemes for larger matrices have further reduced this exponent to $O(n^{2.371552})$ as of July 2023 [4]. In this paper, we explore the parameter space of multiplication schemes of the same form as Strassen's to better understand the landscape of possible solutions.

II. The Brent Equations

Matrix multiplication is defined as the following tensor product:

$$\begin{aligned} C_{a_1b_2} &= \sum_k A_{a_1k}B_{kb_2} \\ C_{c_1c_2} &= A_{a_1a_2}B_{b_1b_2}\ \delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1} \end{aligned} \tag{1}$$

We use Einstein sum notation in (1) to highlight the index relations where the summed indices are obvious to avoid cluttering the expression with sums. Strassen's scheme is expressed in the form:

$$\begin{aligned} C_{c_1c_2} &= \gamma_{c_1c_2p}(\alpha_{pa_1a_2}A_{a_1a_2})(\beta_{pb_1b_2}B_{b_1b_2}) \\ &= A_{a_1a_2}B_{b_1b_2}\ \gamma_{c_1c_2p}\alpha_{pa_1a_2}\beta_{pb_1b_2} \end{aligned} \tag{2}$$

where α are the coefficients on the terms of A used in each of the p products, β is the same for B, and γ are the coefficients of the products when constructing the final entries of C. By setting (1) equal to (2) and noting the linear independence of the entries of A and B (allowing us to set the coefficients of each term in the outer product of AB equal to themselves), we obtain:

$$\sum_p \gamma_{c_1c_2p}\alpha_{pa_1a_2}\beta_{pb_1b_2} = \delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1}$$

Which are the Brent equations for a matrix multiplication scheme using p products [5] [6].

III. Functional Analysis Perspectives

There are two principal perspectives of how these computations "change basis" into a more convenient space for matrix multiplication: the first involves covector measurements, and the dual view involves function evaluation and interpolation (closely related to the Fast Fourier Transform).

A. Dual Functionals

For a moment, we will flatten the matrices A, B, C into vectors and consider matrix multiplication to be a particular vector operation mapping the entries of these vectors to each other. This also flattens the α, β, γ and triple delta tensors into matrices:

$$C_c = \sum_p \gamma_{cp} \cdot \Big(\sum_a \alpha_{pa}A_a\Big)\Big(\sum_b \beta_{pb}B_b\Big)$$

We can interpret the row vectors $\alpha_{p\cdot}$ and $\beta_{p\cdot}$ as functionals or covectors measuring p features of the matrix-vectors A, B pertinent to the underlying operation. These features are then multiplied point-wise and interpreted as the values of the p products in the scheme. Finally, the p columns of $\gamma_{\cdot p}$ are components in the output vector space (the result matrix) corresponding to the features computed in the products. These features, once translated into the output matrix space, are combined linearly to recover the product matrix C.

B. Similarity to FFT

We will continue to use the flattened vectors A, B, C in this section as we further analyze the dual view. Let us choose a set of a basis functions $\alpha_a(p)$ in some variable p from some set Ω (likewise for $\beta_b(p)$). Now the matrices A, B can be mapped to vectors in this function space via a component-wise bijection:

$$A \to A(p) = \sum_a A_a\alpha_a(p)$$

$$B \to B(p) = \sum_b B_b\beta_b(p)$$

The corresponding author is a student at Brigham Young University, under the Computer Science department. He is affiliated with IDeA Labs and can be contacted at gmp99@byu.edu. This work was partially done for course credit and partially supported by DOE Grant #SC0021693.

where $A(p), B(p) : \Omega \to \mathbb{R}$. We multiply these functions $A(p)$ and $B(p)$ point-wise:

$$AB(p) = \sum_{a,b} A_a B_b \alpha_a(p)\beta_b(p)$$

Finally, we integrate (inner product) with some c basis functions $\gamma_c(p)$ to obtain the desired coefficients:

$$\begin{aligned} C_c &= \int_{p\in\Omega} \overline{\gamma_c(p)} \sum_{a,b} A_a B_b \alpha_a(p)\beta_b(p) \\ &= \sum_{a,b} A_a B_b \int_{p\in\Omega} \overline{\gamma_c(p)}\alpha_a(p)\beta_b(p) \qquad (3) \end{aligned}$$

If we select the properties of the basis functions strategically, we can recover the desired scheme:

$$\begin{aligned} \text{let } \int_{p\in\Omega} \overline{\gamma_{c_1c_2}(p)}\alpha_{a_1a_2}(p)\beta_{b_1b_2}(p) &= \delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1} \\ C_{c_1c_2} &= \sum_{a,b} A_{a_1a_2}B_{b_1b_2}\delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1} \end{aligned}$$

An example of such a choice of basis is (assuming $(n \times n) \times (n \times n)$ matrix multiplication):

$$\begin{aligned} \alpha_{a_1a_2}(p) &= e^{2\pi i p(a_1 n + a_2 n^2)/2n^3} \\ \beta_{b_1b_2}(p) &= e^{2\pi i p(b_2 - b_1 n^2)/2n^3} \\ \gamma_{c_1c_2}(p) &= e^{2\pi i p(c_1 n + c_2)/2n^3} \\ \Omega &= \mathbb{R}_{[0,2\pi]} \end{aligned}$$

This produces an orthogonal basis satisfying the desired properties (some normalizing constants provide an orthonormal basis, but we do not write them here). We see that

$$\alpha_{a_1a_2}(p)\beta_{b_1b_2}(p) = e^{2\pi i p(a_1 n + b_2 + (a_2 - b_1)n^2)/2n^3}$$

This corresponds with the basis for γ exactly when $a_2 = b_1$ (for mismatches, the frequency is higher, so the function becomes orthogonal to all γ). As this basis is orthonormal (with the normalizing constants), the integral in (3) recovers the desired entries $\sum_k A_{c_1k}B_{kc_2}$.

The trick empowering FFT is to perform the function of the inner product integral (identifying the coefficients on the basis functions) using an *efficient* discrete domain for p. The exponential basis chosen in our example recovers the coefficients (and thus the entries of C) using n^3 evaluation points p (a less efficient scheme using n^4 points is possible by using $a_1n^3 + a_2n^2 + b_1n + b_2$ in the exponent, with the $\gamma_{a_1b_2}$ terms being $\sum_k \alpha_{a_1k}\beta_{kb_2}$). We envision a strategic choice of basis functions (not necessarily orthogonal) that would allow us to recover n^2 coefficients via interpolation using only $O(n^2 \log n)$ evaluation points p (we also suspect that this bound might only be achieved as n grows unbounded). One approach we tried was to select the basis functions as follows:

$$\begin{aligned} \alpha_{a_1a_2}(p) &= p^{a_1 n} e^{2\pi i p a_2/2n} \\ \beta_{b_1b_2}(p) &= p^{b_2} e^{-2\pi i p b_1/2n} \end{aligned}$$

This produces an evaluation set:

$$AB(p) = \sum_{a_1,a_2,b_1,b_2} A_{a_1a_2}B_{b_1b_2}p^{a_1n+b_2}e^{2\pi i p(a_2-b_1)/2n}$$

Which, assuming real evaluation points p, allows us to separate the real and imaginary parts of the exponential term only:

$$\begin{aligned} &= \sum_{a_1,a_2,b_1,b_2} A_{a_1a_2}B_{b_1b_2}p^{a_1n+b_2}\cos(2\pi i p(a_2-b_1)/2n) \\ &+ i \sum_{a_1,a_2,b_1,b_2} A_{a_1a_2}B_{b_1b_2}p^{a_1n+b_2}\sin(2\pi i p(a_2-b_1)/2n) \end{aligned} \qquad (4)$$

We see that the real part is symmetric in $a_2 - b_1$, and when this difference is zero, we are left with the desired polynomial:

$$\sum_{a_1,k,b_2} A_{a_1k}B_{kb_2}p^{a_1n+b_2}$$

From which we can easily recover the coefficients of each basis polynomial:

$$C_{a_1b_2}p^{a_1n+b_2} = (\sum_k A_{a_1k}B_{kb_2})p^{a_1n+b_2}$$

Even though the polynomial terms are not orthogonal, there exists a dual basis γ uniquely interpolating these functions, which is easily derived using the associated metric tensor. The trick would be to find evaluation points p such that the real and imaginary parts of (4) destructively interfere on the $a_2 \neq b_1$ terms, but constructively interfere when $a_2 = b_1$, and to do this using only $O(\log n)$ evaluation points per basis polynomial $p^{a_1n+b_2}$. We could not get this to work and suspect that a different choice of basis functions is necessary.

IV. Iterative Methods

A. Gradient Decent

We implemented a gradient descent using the following formulae:

$$C = \sum_p \gamma_{c_1c_2p}\alpha_{pa_1a_2}\beta_{pb_1b_2} - \delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1} \qquad (5)$$

$$\begin{aligned} \frac{\partial\|C\|_F}{\partial\gamma_{c_1c_2p}} &= \sum_{a_1a_2b_1b_2} 2C \cdot \alpha_{pa_1a_2}\beta_{pb_1b_2} \\ \frac{\partial\|C\|_F}{\partial\alpha_{pa_1a_2}} &= \sum_{c_1c_2b_1b_2} 2C \cdot \gamma_{c_1c_2p}\beta_{pb_1b_2} \\ \frac{\partial\|C\|_F}{\partial\beta_{pb_1b_2}} &= \sum_{a_1a_2c_1c_2} 2C \cdot \gamma_{c_1c_2p}\alpha_{pa_1a_2} \end{aligned}$$

We also used automatic differentiation on (5) using PyTorch. We found that a high momentum term worked well for faster convergence without much cost for overshooting solutions.

B. Projection: The Moore-Penrose Pseudoinverse

Here, the element-wise 2-norm is equivalent to the Frobenius norm of the tensors. We used tensor inverses to derive the following formulae:

$$\gamma_{c_1c_2p} \leftarrow \delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1} \cdot (\alpha_{pa_1a_2}\beta_{pb_1b_2})^{\dagger}$$
$$\text{where}$$
$$\sum_{a_1a_2b_1b_2} (\alpha_{pa_1a_2}\beta_{pb_1b_2})(\alpha_{\bar{p}a_1a_2}\beta_{\bar{p}b_1b_2})^{\dagger} = \delta_{p\bar{p}}$$

Where the tensor inverse is equivalent to flattening the applicable axes to form a matrix, performing a pseudoinverse operation, and de-flattening the axes. We can update $\alpha_{pa_1a_2}$ and $\beta_{pb_1b_2}$ in a similar manner. We update only one tensor at a time (we do not perform all three atomically), performing all three tensor updates before starting another iteration. This has the effect of projecting each tensor onto the affine hyperplane, which minimizes C when holding the other two tensors fixed. We observe consistent exponential convergence rates using this method, and due to the projection theorem, this should be the optimal step size for optimizing the three tensors separately. It is, however, computationally expensive, requiring three pseudoinverses of matrices in $\mathbb{R}^{p\times n^4}$ for each iteration. It seems that this is worth the cost, converging faster on average.

C. Linear Programming Methods

It is also possible to use the appropriate formulations of the *element-wise* 1- or ∞-norms (the latter is equivalent to the tensor $\max$-norm, or for matrices, the induced $\|\cdot\|_{1\to\infty}$ norm). The 1-norm, however, as a measure of error tends to allow local minima where a nonzero entry of $\delta_{c_1a_1}\delta_{c_2b_2}\delta_{a_2b_1}$ is made zero (when using a product-deficient scheme, this norm tends to simply give up on approximating all entries). The $\max$-norm, however, worked very well. We implemented it using the CVXPY solver. We used a similar approach as the pseudoinverse solver, iteratively optimizing each of the three tensors in succession until convergence. The solver requires the optimization problem to have the form $\|A\vec{x}-\vec{b}\|_{\infty}$, so we column-wise flattened the optimization parameter tensor and the triple delta tensor into $\vec{\gamma}$ and $\vec{b}$ vectors, respectively, as well as extended the fixed parameters into an appropriate block-diagonal matrix A by repeating along a new c_1c_2 composite axis:

$$\gamma_{(c_1c_2p)} \leftarrow \arg\min_{\vec{\gamma}} \|A\vec{\gamma}-\vec{b}\|_{\infty}$$
$$\text{where}$$
$$A = (\alpha_{pa_1a_2}\beta_{pb_1b_2})_{(a_1a_2b_1b_2\bar{c}_1\bar{c}_2)(c_1c_2p)}$$
$$\vec{b} = (\delta_{\bar{c}_1a_1}\delta_{\bar{c}_2b_2}\delta_{a_2b_1})_{(a_1a_2b_1b_2\bar{c}_1\bar{c}_2)}$$

And similarly for the other two tensors. This allowed us to derive a scheme of size $2\times2\times2$ using 6 multiplications with a $\max$-norm error of exactly $1/8$.

We can likewise introduce a penalty for parameter sizes (the entries of α, β, γ); for this, we would like a sparse representation, so using the 1-norm for this loss would best approximate this. We find our new loss to be:

$$L = \|C\|_{\infty} + \lambda(\|\alpha\|_1 + \|\beta\|_1 + \|\gamma\|_1)$$

Where the norms are element-wise and λ is a weight on the parameter loss.

We implemented this loss function in PyTorch (not as a linear program) and found that the optimizer quickly bounded the error by one. Still, once this bound was achieved, it lost the ability to progress any further. This is likely because the program is blind to changes made on other entries when taking a step, thus hindering its progress. A true linear program might circumvent this, but it would probably run in no better than $O(n^4)$ (which is better than using the pseudo-inverse, which has $O(n^6)$ runtime).

V. Algebraic Manipulations for Existing Schema

A. Generating the Naïve Schemes

In general, any naïve multiplication scheme for the multiplication $A_{nd}B_{dm} = C_{nm}$ can be generated using the following tensors:

$$\alpha_{pa_1a_2} = I_{p_1a_1}I_{p_2a_2}\vec{1}_{p_3}$$
$$\beta_{pb_1b_2} = \vec{1}_{p_1}I_{p_2b_1}I_{p_3b_2}$$
$$\gamma_{c_1c_2p} = I_{c_1p_1}\vec{1}_{p_2}I_{c_2p_3}$$

where the axes' sizes are $a_1 = c_1 = p_1 = n, a_2 = b_1 = p_2 = d, b_2 = c_2 = p_3 = m$, $\vec{1}$ is the vector of ones, and the axes $p_1p_2p_3$ flatten into axis p of size ndm (these are all outer products). Multiplying out all three tensors and summing across corresponding p_is does indeed produce $\delta_{c_1a_1}\delta_{a_2b_1}\delta_{c_2b_2}$, making such an assignment a valid scheme of size ndm.

B. Permuting Products

We can permute the p products in any consistent way we wish using a permutation σ:

$$\alpha'_{pnd} = \alpha_{\sigma(p)nd}$$
$$\beta'_{pdm} = \beta_{\sigma(p)dm}$$
$$\gamma'_{nmp} = \gamma_{nm\sigma(p)}$$

C. Basis Changes on Matrices A, B, C

One can easily change the basis of the matrices A, B, C using some invertible L, M, R:

$$(L_{nN}C_{NM}R_{Mm}) = (L_{nN}A_{ND}M_{Dd})(M^{\dagger}_{dD}B_{DM}R_{Mm})$$

We can use these bases to derive a modified scheme:

$$\alpha_{pND} = M_{Dd}\alpha_{pnd}L_{nN}$$
$$\beta_{pDM} = R_{Mm}\beta_{pdm}M^{\dagger}_{dD}$$
$$\gamma_{NMp} = L^{\dagger}_{Nn}\gamma_{nmp}R^{\dagger}_{mM}$$

Where the capitalized indices indicate the new scheme/basis, which could be of a smaller size than the old (i.e., L_{nN} need only be left invertible, while M_{Dd} and R_{Mm} need only be right invertible; hence the pseudoinverse $\dagger$). When using a well-conditioned basis change, the scheme's Frobenius error is invariant (due to the unitary invariance of the Frobenius norm), while the 1- and $\max$-norm errors are affected. This change of basis does not touch the product axis p, so the number of products used remains the same, even if the scheme is for smaller N, D, M. We tried using a different basis for each product along the p axis, but the result did not preserve the scheme's performance.

D. Transposing Schemes

Another form of basis change is transposition of the original equation $C = AB \to C^\top = B^\top A^\top$. This leads to a simple permutation of the entries of the tensors:

$$\begin{aligned}\alpha'_{pnd} &= \beta_{pdn}\\ \beta'_{pdm} &= \alpha_{pmd}\\ \gamma'_{nmp} &= \gamma_{mnp}\end{aligned}$$

E. Composing Existing Schema

We can also compose schemes. In illustration, these schemes are most useful when used recursively to extend their computational complexity to larger applications. The recursive call structure produces a scheme for the larger matrix; we can also derive this scheme directly using outer products:

$$\begin{aligned}\alpha_{(pP)(nN)(dD)} &= \alpha_{pnd}\alpha_{PND}\\ \beta_{(pP)(dD)(mM)} &= \beta_{pdm}\beta_{PDM}\\ \gamma_{(nN)(mM)(pP)} &= \gamma_{nmp}\gamma_{NMP}\end{aligned}$$

Where the capital indices are two schemes, and the parenthesized indices are flattened into a single axis. The order of the indices in the parentheses determines the scheme's order of use in the call structure. Transposing the schemes (for example, flattening the indices (Nn) in α and γ instead of using the order above) will predictably permute the scheme, but the values will be identical. More general extensions are possible, where a different scheme is used for each recursive sub-multiplication.

When square schemes ($n = d = m$) are composed in this way, the complexity of the composed scheme is $O(n^c)$, where

$$c = \frac{c_1}{1 + \log_{p_1} p_2} + \frac{c_2}{1 + \log_{p_2} p_1}$$

where $O(n^{c_i})$ is the complexity of each scheme and p_i is the number of products used in that scheme. If $p_1 = p_2$, this is a simple mean $(c_1 + c_2)/2$. Otherwise, the mean is biased toward the c_i corresponding with the larger p_i, with the extreme case (one p_i is effectively infinitely greater than the other) approaching the value of the then-dominant c_i.

To compute the error of the composed scheme, Let $\Sigma_i = \alpha\beta\gamma$ for some scheme i, and $\Delta_i = \delta_{c_1 a_1}\delta_{a_2 b_1}\delta_{c_2 b_2}$ for the same scheme. The Frobenius (element-wise 2-) norm of the error $\|\Sigma_i - \Delta_i\|_F^2 = \|\Sigma_i\|_F^2 + \|\Delta_i\|_F^2 - 2\langle\Sigma_i, \Delta_i\rangle_F$. The error of a composed scheme constructed from Σ_i and Σ_j is:

$$\begin{aligned}\|\Sigma_i\Sigma_j - \Delta_i\Delta_j\|_F^2 =& \|\Sigma_i\|_F^2\|\Sigma_j\|_F^2 + \|\Delta_i\|_F^2\|\Delta_j\|_F^2\\ &- 2\langle\Sigma_i, \Delta_i\rangle_F\langle\Sigma_j, \Delta_j\rangle_F\end{aligned}$$

A proof of these equations is straight forward using the definition of the Frobenius inner product; we can also consider each tensor element-wise as a vector, where the Frobenius norm becomes the 2-norm, and the composed scheme error is the Frobenius norm of the difference between the outer products of these Σ and Δ "vectors." Note that $\|\Delta\|_F = ndm$. We observed that as one iteratively composes schemes, the average error decreased monotonically for any p, including $p = n^2$. This is not surprising, however, as the number of entries (n^6 for Σ) grows faster than the total error:

$$\frac{\|\Sigma^k - \Delta^k\|_F}{n^{6k}} = \left\|\left(\frac{\Sigma}{n^6}\right)^k - \left(\frac{\Delta}{n^6}\right)^k\right\|_F$$

F. Real Schemes From Complex Schemes

The derivation of a real scheme (tensors are real-valued) from a complex scheme is trivial and only increases the number of products p by a factor of 4. We use the flattened notation from III-A for convenience. These tensors are stacked along the p-axis.

$$\begin{gathered}\alpha = \alpha_R + i\alpha_I\\ \beta = \beta_R + i\beta_I\\ \gamma = \gamma_R + i\gamma_I\\ \Downarrow\\ \alpha' = \begin{bmatrix}\alpha_R\\ \alpha_I\\ \alpha_R\\ \alpha_I\end{bmatrix},\ \beta' = \begin{bmatrix}\beta_R\\ \beta_I\\ \beta_I\\ \beta_R\end{bmatrix}\\ \gamma' = \begin{bmatrix}\gamma_R & -\gamma_R & -\gamma_I & -\gamma_I\end{bmatrix}\end{gathered}$$

G. Degenerate Schemes and the Manifold of Solutions

Throughout this section, we use the convention that the matrix multiplication is of size $(n \times d) \times (d \times m)$, or (n, d, m, p), similar to V-C and V-E. The degrees of freedom for each scheme was derived either algebraically or by sampling the neighborhood and projecting back onto the solution manifold, analyzing the dimensionality of the projected neighborhood. The degrees of freedom in the parameters of a scheme are exactly equal to the intrinsic dimension of the solution manifold near that scheme. We believe that the algebraic solutions suggest there is no speedup possible, but some pseudoinverse/null space trickery may be able to be done.

Note that the total number of parameters in the tensors is $p(nd + dm + nm)$, so the solution manifold is a nonlinear subset of this parameter space.

1) Vector-Vector (inner): For dot products, $n = m = 1$, and there are exactly $p(d + 1)$ degrees of freedom in the set of valid schemes. Given any arbitrary β, γ (or swap α, β in a transposed multiplication), the derivation of a unique α satisfying the scheme is given by the following pseudoinverse:

$$\alpha_{p(1d)} = \left(\beta_{pd\bar{1}}\gamma_{1\bar{1}p}\right)^\dagger_{(1d)p}$$

2) Vector-Vector (outer): For outer products, $d = 1$, and there are exactly $p(n + m)$ degrees of freedom in the set of valid schemes. Given any arbitrary α, β, the derivation of a unique γ satisfying the scheme is given:

$$\gamma_{(nm)p} = \left(\alpha_{pn\bar{1}}\beta_{p\bar{1}m}\right)_{p(nm)}^{\dagger}$$

3) Matrix-Vector: For matrix-vector (and by transpose vector-matrix) products, $m = 1$, and there are exactly $p(n+d)$ degrees of freedom in the set of valid schemes. Given any arbitrary β, γ (or swap α, β in a transposed multiplication), the derivation of a unique α satisfying the scheme is given:

$$\alpha_{p(nd)} = \left(\beta_{pd\bar{1}}\gamma_{n\bar{1}p}\right)_{(nd)p}^{\dagger}$$

4) Naïve Schemes: The intrinsic dimension of the solution manifold around in the neighborhood of naïve schemes (from V-A) seems to be of dimension $p(n + d + m - 1)$ based on adding noise, projecting, and analyzing the resulting cloud of solutions. This is consistent with the degenerate cases listed above.

5) Other Schemes: We analyzed the intrinsic dimensions of efficient schemes. Strassen and Winograd provide schemes for size $(2, 2, 2, 7)$, and the intrinsic dimension around these schemes was 23 (note this is $2 \cdot 7 + 9$). Laderman's $(3, 3, 3, 23)$ scheme has local degrees of freedom of dimension 76 ($= 3 \cdot 23 + 7$). Composing Strassen's scheme with itself provides a $(4, 4, 4, 49)$ scheme with dimension 251 ($= 4 \cdot 49 + 2 \cdot 23 + 9$). To compare, a random $(2, 2, 2, 8)$ valid scheme had local dimension 32 (40 for naïve as computed in the previous paragraph).

Acknowledgment

The authors thank Chris Archibald, David Grimsman, and Sean Warnick for the opportunity to pursue this research. This paper began as a passion project during Dr. Grimsman's algorithm design and analysis course, was adopted into IDeA Labs at BYU, and received attention as a machine learning project in Dr. Archibald's Intro to AI course.

The Python code of the described methods in this paper is in the GitHub repository found at [7].

References

[1] M. Bläser, "On the complexity of the multiplication of matrices of small formats," *Journal of Complexity*, vol. 19, no. 1, pp. 43–60, 2003.

[2] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Transactions On Algorithms (TALG)*, vol. 1, no. 1, pp. 2–13, 2005.

[3] V. Strassen, "Gaussian elimination is not optimal," *Numerische mathematik*, vol. 13, no. 4, pp. 354–356, 1969.

[4] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou, "New bounds for matrix multiplication: from alpha to omega," in *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 3792–3835, SIAM, 2024.

[5] R. P. Brent, "Algorithms for matrix multiplication," tech. rep., Stanford University Stanford, 1970.

[6] N. T. Courtois, D. Hulme, and T. Mourouzis, "Multiplicative complexity and solving generalized brent equations with sat solvers," *COMPUTATION TOOLS*, vol. 2012, pp. 22–27, 2012.

[7] G. M. Perry, "Brent scheme." https://github.com/perryGabriel/brent-scheme, 2026. Accessed: 2026-01-06.