Surya Matrix Multiplication: An Optimized 2-Multiplication Method for 2x2 Symmetric Circulant Matrices

Surya Teja Keerthi
keerthisuryateja2005@gmail.com

May 17, 2025

Abstract

Matrix multiplication is a cornerstone of computational mathematics. Standard algorithms for 2x2 matrices require 8 scalar multiplications, while Strassen's algorithm reduces this to 7. This paper introduces and details Surya Matrix Multiplication, a specialized method tailored for 2x2 symmetric matrices of the circulant form $\begin{bmatrix} a & b \\ b & a \end{bmatrix}$. We provide a direct algebraic derivation showing that the product can be achieved using only 2 scalar multiplications. This offers a significant computational advantage for this specific matrix structure. The method is compared against standard and Strassen's multiplication, including algorithmic descriptions and Python implementations, to highlight its distinct characteristics and efficiency gains. The resulting product matrix also retains the symmetric circulant form.

1 Introduction

The multiplication of matrices is a fundamental operation in linear algebra with wide-ranging applications [1]. For two general $N \times N$ matrices, the standard algorithm has a complexity of $O(N^3)$. For the base case of $N = 2$, this involves 8 scalar multiplications and 4 scalar additions. In 1969, Strassen presented a recursive algorithm that reduces the number of multiplications, achieving $O(N^{\log_2 7}) \approx O(N^{2.807})$ complexity, and requiring 7 multiplications for the 2×2 case [2].

While general-purpose algorithms are invaluable, specialized structures within matrices can often be exploited for further computational gains. This work focuses on a particular class of symmetric matrices: 2x2 matrices where the diagonal elements are equal and the off-diagonal elements are equal. Such matrices can be expressed as:

$$A = \begin{bmatrix} a & b \\ b & a \end{bmatrix}$$

These are also known as 2x2 symmetric circulant matrices. We present and analyze a method, termed Surya Matrix Multiplication, that computes the product of two such matrices using only 2 scalar multiplications. This paper provides a detailed derivation, compares it with standard and Strassen's methods, and includes implementation examples.

2 Methods of 2x2 Matrix Multiplication

Let $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$ and $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$. Their product is $C = A \times B = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$.

2.1 Standard Matrix Multiplication

The elements of C are computed as:

$$
\begin{aligned}
c_{11} &= a_{11}b_{11} + a_{12}b_{21} \\
c_{12} &= a_{11}b_{12} + a_{12}b_{22} \\
c_{21} &= a_{21}b_{11} + a_{22}b_{21} \\
c_{22} &= a_{21}b_{12} + a_{22}b_{22}
\end{aligned}
$$

This method requires 8 scalar multiplications and 4 scalar additions.

Algorithm 1 Standard 2x2 Matrix Multiplication

Require: Matrices $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$

Ensure: Product matrix C

1: $c_{11} \leftarrow a_{11} \cdot b_{11} + a_{12} \cdot b_{21}$
2: $c_{12} \leftarrow a_{11} \cdot b_{12} + a_{12} \cdot b_{22}$
3: $c_{21} \leftarrow a_{21} \cdot b_{11} + a_{22} \cdot b_{21}$
4: $c_{22} \leftarrow a_{21} \cdot b_{12} + a_{22} \cdot b_{22}$
5: **return** $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

2.2 Strassen's Algorithm (for 2x2 matrices)

Strassen's method reduces the number of multiplications by defining 7 intermediate products (M_1 to M_7):

$$
\begin{aligned}
M_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
M_2 &= (a_{21} + a_{22})b_{11} \\
M_3 &= a_{11}(b_{12} - b_{22}) \\
M_4 &= a_{22}(b_{21} - b_{11}) \\
M_5 &= (a_{11} + a_{12})b_{22} \\
M_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
M_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}
$$

The elements of C are then:

$$
\begin{aligned}
c_{11} &= M_1 + M_4 - M_5 + M_7 \\
c_{12} &= M_3 + M_5 \\
c_{21} &= M_2 + M_4 \\
c_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

This method requires 7 scalar multiplications and 18 scalar additions/subtractions.

Algorithm 2 Strassen's 2x2 Matrix Multiplication

Require: Matrices $A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$, $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$
Ensure: Product matrix C
1: $M_1 \leftarrow (a_{11} + a_{22}) \cdot (b_{11} + b_{22})$
2: $M_2 \leftarrow (a_{21} + a_{22}) \cdot b_{11}$
3: $M_3 \leftarrow a_{11} \cdot (b_{12} - b_{22})$
4: $M_4 \leftarrow a_{22} \cdot (b_{21} - b_{11})$
5: $M_5 \leftarrow (a_{11} + a_{12}) \cdot b_{22}$
6: $M_6 \leftarrow (a_{21} - a_{11}) \cdot (b_{11} + b_{12})$
7: $M_7 \leftarrow (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$
8: $c_{11} \leftarrow M_1 + M_4 - M_5 + M_7$
9: $c_{12} \leftarrow M_3 + M_5$
10: $c_{21} \leftarrow M_2 + M_4$
11: $c_{22} \leftarrow M_1 - M_2 + M_3 + M_6$
12: **return** $C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$

2.3 Surya Matrix Multiplication (for 2x2 Symmetric Circulant Matrices)

This method is specialized for matrices of the form $A = \begin{bmatrix} a & b \\ b & a \end{bmatrix}$ and $B = \begin{bmatrix} x & y \\ y & x \end{bmatrix}$. The standard product $C = A \times B$ is $C = \begin{bmatrix} ax + by & ay + bx \\ bx + ay & by + ax \end{bmatrix}$.

The derivation involves two intermediate products, P_1 and P_2:

1. Let $S_A = a + b$ and $D_A = a - b$.
2. Let $S_B = x + y$ and $D_B = x - y$.

The two key multiplications are:

$$P_1 = S_A \cdot S_B = (a + b)(x + y) \tag{1}$$
$$P_2 = D_A \cdot D_B = (a - b)(x - y) \tag{2}$$

The elements of $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ are:

$$C_{11} = C_{22} = \frac{P_1 + P_2}{2} = ax + by \tag{3}$$
$$C_{12} = C_{21} = \frac{P_1 - P_2}{2} = ay + bx \tag{4}$$

This method requires 2 scalar multiplications, 4 additions/subtractions for S_A, D_A, S_B, D_B, 2 additions/subtractions for C_{11}, C_{12} numerators, and 2 divisions by 2. Totaling 2 multiplications and 6 additions/subtractions (plus 2 cheap divisions).

Algorithm 3 Surya Matrix Multiplication for $A = \begin{bmatrix} a & b \\ b & a \end{bmatrix}, B = \begin{bmatrix} x & y \\ y & x \end{bmatrix}$

Require: Matrices $A = \begin{bmatrix} a & b \\ b & a \end{bmatrix}$ and $B = \begin{bmatrix} x & y \\ y & x \end{bmatrix}$.
Ensure: Product matrix $C = A \times B$.
1: $s_a \leftarrow a + b$
2: $d_a \leftarrow a - b$
3: $s_b \leftarrow x + y$
4: $d_b \leftarrow x - y$
5: $P_1 \leftarrow s_a \cdot s_b$ ▷ First multiplication
6: $P_2 \leftarrow d_a \cdot d_b$ ▷ Second multiplication
7: $C_{11} \leftarrow (P_1 + P_2)/2$
8: $C_{12} \leftarrow (P_1 - P_2)/2$
9: **return** $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{12} & C_{11} \end{bmatrix}$

3 Comparative Analysis

Table 1: Operation Count Comparison for 2x2 Matrix Multiplication

Method	**Multiplications**	**Additions/Subtractions**	**Applicability**
Standard	8	4	General 2x2
Strassen's	7	18	General 2x2
Surya	2	6 (+2 div by 2)	Symmetric Circulant 2x2

Differences in Approach and Applicability:

- **Standard Multiplication:** This is the direct definition-based approach. It is simple to understand and implement but is the least efficient in terms of multiplications for general matrices.
- **Strassen's Algorithm:** Strassen's method achieves a reduction in multiplications by cleverly rearranging the computation into 7 products and then combining them with a larger number of additions and subtractions. It is applicable to general 2x2 matrices and forms the basis for faster large matrix multiplication. The overhead of additional additions means it's typically beneficial when multiplications are significantly more expensive than additions, or for larger matrices where the recursive benefit kicks in.
- **Surya Matrix Multiplication:** This method is highly specialized. It achieves its remarkable reduction to 2 multiplications by exploiting the very specific structure of 2x2 symmetric circulant matrices ($a_{11} = a_{22}$, $a_{12} = a_{21}$). It does not apply to general 2x2 matrices or even general symmetric matrices (where $a_{11} \neq a_{22}$). The method essentially transforms the problem into an eigenvalue-like domain (sums and differences), performs the multiplication there, and transforms back.

The key difference is that Surya Matrix Multiplication leverages inherent symmetries and structural properties far beyond what general-purpose algorithms like Strassen's can assume. Strassen's algorithm works by finding an algebraic identity that reduces multiplications for any 2x2 matrix. Surya's method finds an even more potent identity valid only for a restricted class of matrices.

4 Implementation Examples in Python

Below are Python implementations for the three methods, demonstrating their application.

```python
def standard_multiply(A, B):
    """Standard 2x2 matrix multiplication."""
    # A = [[a11, a12], [a21, a22]]
    # B = [[b11, b12], [b21, b22]]
    a11, a12 = A[0]
    a21, a22 = A[1]
    b11, b12 = B[0]
    b21, b22 = B[1]

    c11 = a11 * b11 + a12 * b21
    c12 = a11 * b12 + a12 * b22
    c21 = a21 * b11 + a22 * b21
    c22 = a21 * b12 + a22 * b22
    return [[c11, c12], [c21, c22]]

def strassen_multiply(A, B):
    """Strassen's algorithm for 2x2 matrix multiplication."""
    a11, a12 = A[0]
    a21, a22 = A[1]
    b11, b12 = B[0]
    b21, b22 = B[1]

    m1 = (a11 + a22) * (b11 + b22)
    m2 = (a21 + a22) * b11
    m3 = a11 * (b12 - b22)
    m4 = a22 * (b21 - b11)
    m5 = (a11 + a12) * b22
    m6 = (a21 - a11) * (b11 + b12)
    m7 = (a12 - a22) * (b21 + b22)

    c11 = m1 + m4 - m5 + m7
    c12 = m3 + m5
    c21 = m2 + m4
    c22 = m1 - m2 + m3 + m6
    return [[c11, c12], [c21, c22]]

def surya_multiply(A_sc, B_sc):
    """Surya's method for 2x2 symmetric circulant matrices."""
    # A_sc = [a, b] representing [[a, b], [b, a]]
    # B_sc = [x, y] representing [[x, y], [y, x]]
    a, b = A_sc
    x, y = B_sc

    sa = a + b
    da = a - b
    sb = x + y
    db = x - y

    p1 = sa * sb  # Multiplication 1
    p2 = da * db  # Multiplication 2

    c11 = (p1 + p2) / 2
    c12 = (p1 - p2) / 2

    # Ensure integer results if inputs are integers and division is exact
    if isinstance(a, int) and isinstance(b, int) and \
       isinstance(x, int) and isinstance(y, int) and \
       (p1 + p2) % 2 == 0 and (p1 - p2) % 2 == 0:
        c11 = int(c11)
        c12 = int(c12)
```

```
    return [[c11, c12], [c12, c11]]

# Example Usage:
# Define matrices for Surya's method (symmetric circulant)
# A = [[3, 5], [5, 3]]  => a=3, b=5
# B = [[2, 4], [4, 2]]  => x=2, y=4

A_surya_form = [3, 5]
B_surya_form = [2, 4]

# Full matrix forms for standard and Strassen
A_full = [[3, 5], [5, 3]]
B_full = [[2, 4], [4, 2]]

print("Standard Multiplication Result:")
print(standard_multiply(A_full, B_full))

print("\nStrassen's Multiplication Result:")
print(strassen_multiply(A_full, B_full))

print("\nSurya's Multiplication Result:")
print(surya_multiply(A_surya_form, B_surya_form))

# Example with a general matrix (where Surya's method is not applicable)
A_general = [[1, 2], [3, 4]]
B_general = [[5, 6], [7, 8]]

print("\nStandard Multiplication (General Matrices):")
print(standard_multiply(A_general, B_general))
print("\nStrassen's Multiplication (General Matrices):")
print(strassen_multiply(A_general, B_general))
# Surya's method would not be directly applicable here in its current form
```

Listing 1: Python Implementations for 2x2 Matrix Multiplication

5 Example from Previous Sections

Let $A = \begin{bmatrix} 3 & 5 \\ 5 & 3 \end{bmatrix}$ and $B = \begin{bmatrix} 2 & 4 \\ 4 & 2 \end{bmatrix}$. Using Surya Matrix Multiplication ($a = 3, b = 5, x = 2, y = 4$): $P_1 = (3+5)(2+4) = 8 \times 6 = 48$ $P_2 = (3-5)(2-4) = (-2) \times (-2) = 4$ $C_{11} = (48+4)/2 = 26$ $C_{12} = (48-4)/2 = 22$. Result: $C = \begin{bmatrix} 26 & 22 \\ 22 & 26 \end{bmatrix}$. This matches the output from the Python code for all three methods when applied to these specific symmetric circulant matrices.

6 Discussion and A Note on Related Work

The matrices of the form $\begin{bmatrix} a & b \\ b & a \end{bmatrix}$ are 2x2 symmetric circulant matrices. It is known in the literature that circulant matrices can be diagonalized by the Discrete Fourier Transform (DFT) matrix, and their products can be computed by element-wise multiplication of their eigenvalues, followed by an inverse DFT [3]. For an $N \times N$ circulant matrix, this approach involves N multiplications of eigenvalues. For $N = 2$, this corresponds to 2 multiplications. The eigenvalues of $\begin{bmatrix} a & b \\ b & a \end{bmatrix}$ are $a + b$ and $a - b$. The products $(a + b)(x + y)$ and $(a - b)(x - y)$ are precisely P_1 and P_2, which are products of the corresponding eigenvalues. The reconstruction formulas for C_{11} and C_{12} are equivalent to an inverse DFT operation for $N = 2$.

The Surya Matrix Multiplication method, derived here through direct algebraic manipulation, provides a self-contained and explicit algorithm that achieves this 2-multiplication efficiency without direct reference to DFT theory. This directness can be advantageous for implementation and pedagogical clarity, especially when the broader theory of circulant matrices or DFTs is not immediately invoked. It highlights how fundamental algebraic insights can lead to highly optimized routines for specific structures.

Potential applications include areas where such symmetric, patterned 2x2 transformations arise, such as in simplified physical models, specific signal processing kernels, or graph theoretical problems involving 2-node systems with uniform coupling.

7 Conclusion

Surya Matrix Multiplication offers a computationally efficient method for multiplying 2x2 symmetric circulant matrices, significantly reducing the number of scalar multiplications from 8 (standard) or 7 (Strassen) to just 2. This paper has detailed the method, compared it algorithmically and operationally with standard and Strassen's multiplication, and provided Python implementations to illustrate its application. While this result aligns with known properties of circulant matrices, the direct derivation and explicit algorithm are valuable for optimizing computations involving this specific matrix form. This specialized approach underscores the benefits of exploiting matrix structure for computational gains, a principle that can be extended by integrating such specific optimizations into adaptive matrix multiplication libraries.

References

[1] Golub, G. H., & Van Loan, C. F. (2013). *Matrix Computations* (4th ed.). Johns Hopkins University Press.

[2] Strassen, V. (1969). Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13(4), 354–356.

[3] Davis, P. J. (1979). *Circulant Matrices*. John Wiley Sons. (Republished by AMS Chelsea Publishing, 2012).