Article

Not peer-reviewed version

Overlays for 4x4 Matrix Multiplication: Peeling, Value-Aware Collapse, and Bit-Sliced Zero-Multiply Paths

Michael Rey *

Posted Date: 12 September 2025

doi: 10.20944/preprints202509.1019.v1

Keywords: matrix multiplication; Strassen; bilinear SLP; value-aware collapse; peeling; bit-sliced GEMM; XNOR-popcount; exact integer multiplication

 doi:10.20944/preprints202509.1019.v1

Article

Overlays for 4 × 4 Matrix Multiplication: Peeling, Value-Aware Collapse, and Bit-Sliced Zero-Multiply Paths

Michael Rey

Octonion Group, Hong Kong; contact@octoniongroup.com

Abstract: This paper introduces a suite of exact overlays for 4x4 matrix multiplication that reduce the counted number of multiplications without altering the underlying bilinear complexity of the base algorithm, such as Strassen's 49-multiplication algorithm or the optimal 48-multiplication algorithm. We present several value-aware techniques that exploit the numerical properties of the input matrices. The primary contributions include the 'Peeler' method, a per-leaf mode subtraction technique with a provably tight cost law; 'Value-Aware Collapse', a counting model that treats multiplications by -1, 0, or 1 as free; and a 'Block Peeler' for optimal algorithms. To further amplify the effectiveness of these methods, we introduce zero-arithmetic-cost structural modifications, including 'Permutation Clustering' and 'Inner Sign-Perm Reindexing'. Additionally, we propose a 'Bit-Sliced GEMM' path for integer and fixed-point matrices that eliminates scalar multiplications entirely, replacing them with efficient bitwise operations. We derive closed-form expressions for the expected computational savings for various discrete input distributions and validate our findings with a comprehensive experimental protocol. The proposed overlays preserve the worst-case performance of the base algorithms while delivering substantial computational gains on structured, quantized, or sparse data, offering a practical path to accelerating matrix multiplication in a variety of real-world applications.

Keywords: matrix multiplication; Strassen; bilinear SLP; value-aware collapse; peeling; bit-sliced GEMM; XNOR–popcount; exact integer multiplication

1. Introduction

Dense matrix multiplication is a fundamental operation in countless scientific and engineering domains. The standard algorithm for multiplying two $n \times n$ matrices requires $O(n^3)$ operations. While asymptotically faster algorithms exist, such as Strassen's algorithm [1], which reduces the complexity to $O(n^{\log_2 7})$, for small, fixed-size matrices like 4x4, the landscape is more nuanced. The naive algorithm for 4x4 matrix multiplication requires 64 multiplications. A single level of Strassen's algorithm reduces this to 49 multiplications, and the best-known bilinear straight-line program (SLP) over rational, real, or complex numbers achieves 48 multiplications [2].

However, these algorithms are designed for the general case and do not take into account the specific values of the matrix entries. In many practical applications, matrices are not composed of arbitrary real numbers but instead exhibit significant structure. They may be sparse, quantized, or have entries drawn from a small, discrete set. This paper introduces the concept of exact overlays: techniques that can be applied on top of existing matrix multiplication algorithms to reduce the number of counted multiplications by exploiting the numerical properties of the input matrices, without altering the underlying bilinear rank of the algorithm.

Our contributions are as follows:

- **The Peeler:** A per-leaf mode subtraction technique for Strassen's algorithm with a provably tight per-leaf cost law of $\min(7, 10 - 2k)$, where k is the multiplicity of the mode.
- **Value-Aware Collapse (VAC):** A counting model where multiplications by -1, 0, or 1 are considered free, mirroring the efficiency of fused sign/zero paths in hardware.

 doi:10.20944/preprints202509.1019.v1

- **The Block Peeler (BP):** An adaptation of the Peeler for the optimal 48-multiplication algorithm, which uses a "free-quota rule" to decide whether to apply the peeling operation.
- **Permutation Clustering (PC) and Inner Sign-Perm Reindexing (ISPR):** Zero-arithmetic-cost techniques that reorder matrix elements to maximize the effectiveness of the Peeler and VAC.
- **Hypercomplex Leaf Detector (HLD):** A method to identify 2x2 subproblems that can be solved with only three multiplications by recognizing them as complex or split-complex multiplications.
- **Bit-Sliced GEMM (BSG):** A zero-multiply path for integer and fixed-point matrices that replaces scalar multiplications with a series of bitwise AND, XNOR, POPCNT, and shift operations.
- **Analytical Results:** We provide closed-form expressions for the expected computational savings for various discrete input distributions and present a reproducible experimental protocol to validate our findings.

2. Preliminaries & Cost Models

2.1. Baselines

Our work builds upon two well-established algorithms for 4x4 matrix multiplication:

- **Strassen-49 (S49):** This algorithm applies one level of Strassen's recursive algorithm to a 4x4 matrix [1]. The 4x4 matrices are treated as 2x2 block matrices, where each block is a 2x2 matrix. Strassen's algorithm is then used to compute the product of these 2x2 block matrices, which involves 7 recursive multiplications of 2x2 matrices. The standard algorithm is then used for the 2x2 matrix multiplications, each requiring 7 multiplications. This results in a total of $7 \times 7 = 49$ scalar multiplications. The additions and subtractions required for the recombination of the leaf products are not counted in this standard model.
- **Optimal 48-multiplication algorithm:** This refers to a specific bilinear straight-line program that computes the product of two 4x4 matrices using only 48 scalar multiplications [2]. This is the optimal known bilinear complexity for 4x4 matrix multiplication over fields of characteristic not equal to 2. The algorithm is defined by a fixed set of 48 products, where each product is the multiplication of a linear combination of the elements of the input matrices, and a final linear recombination of these products to form the output matrix.

2.2. Counting Models

To quantify the benefits of our proposed overlays, we define three distinct cost models:

- **Standard Count:** In the standard model, each scalar multiplication or division contributes 1 to the total cost. Additions, subtractions, and sign flips are considered to have zero cost. This is the conventional model used to analyze the complexity of matrix multiplication algorithms [3].
- **Value-Aware Collapse (VAC):** This model reflects the fact that in many hardware implementations, multiplications by -1, 0, or 1 are significantly cheaper than general multiplications. In the VAC model, a scalar multiplication is considered **free** (i.e., has a cost of 0) if **either** of its operands is an element of the set $\{-1, 0, 1\}$. This model is particularly relevant for matrices with quantized or sparse data.
- **Bit-Sliced Count:** This model is applicable only to integer and fixed-point matrices. In this model, the scalar multiplication count is always 0. Instead, we report the number of bitwise operations (AND, XNOR, popcount), shifts, and additions required to compute the product. This model is motivated by the potential for a completely multiplier-free implementation of matrix multiplication.

3. Overlays for Strassen-49

3.1. Peeler (Per-Leaf Mode Subtraction)

Definition 1 (Peeler). *For a 2x2 leaf (X, Y), let $m_X = \text{mode}(X)$ and k_X be its multiplicity. Similarly, let $m_Y = \text{mode}(Y)$ and k_Y be its multiplicity. If $k := \max(k_X, k_Y) \geq 2$, we peel the side with the larger multiplicity:*

 doi:10.20944/preprints202509.1019.v1

$$X' := X - m_X J_2 \quad (\text{or } Y' := Y - m_Y J_2),$$

where J_2 is the 2x2 matrix of all ones. We then compute the residual product with a zero-aware naive algorithm and add the rank-1 term:

$$m_X(J_2 Y) \quad \text{or} \quad m_Y(X J_2).$$

Theorem 1 (Leaf cost law)**.** *The cost of a leaf multiplication using the Peeler is given by:*

$$\text{cost}_{leaf}^{Peeler} = \min\{7, 10 - 2k\}$$

Proof. In the product $X'Y$, exactly k entries of X (or Y) become zero. A zero-aware naive multiplication of two 2x2 matrices has a cost of $2(4-k)$, as each of the $4-k$ remaining non-zero entries in X' is multiplied by two entries in Y. The add-back scaling operation costs 2 multiplications. Therefore, the total cost is $2(4-k)+2 = 10-2k$. We accept this method if and only if $10-2k < 7$, which simplifies to $k \geq 2$. □

3.2. VAC (0/±1) at Strassen leaves

Definition 2 (VAC Rule)**.** *In each 2x2 leaf of the Strassen algorithm, a product is charged (i.e., counted as a multiplication) only when both the column entry of the first matrix and the row entry of the second matrix are not in the set $\{-1, 0, 1\}$.*

3.3. Zero-Overhead Structure Amplifiers

- **Permutation Clustering (PC):** We can scan all 36 possible block splits of the 4x4 matrices (by choosing 2 rows and 2 columns) and select the split that minimizes the total cost, calculated as $\sum_{i=1}^{7} \min(7, 10 - 2k_i)$. This is a zero-cost operation as it only involves re-indexing.
- **Inner Sign-Perm Reindexing (ISPR):** We can apply a transformation (DP) to the inner dimension of the matrix multiplication (columns of A, rows of B), where D is a diagonal matrix with entries in $\{+1, -1\}$ and P is a permutation matrix. We can choose from a small random set of such transformations to minimize the Peeler+VAC count. This is an exact transformation that only adds sign flips and reordering.
- **Hypercomplex Leaf Detector (HLD):** If a 2x2 leaf has the structure of a complex or split-complex number, i.e., $\begin{pmatrix} a & -b \\ b & a \end{pmatrix}$ or $\begin{pmatrix} a & b \\ b & a \end{pmatrix}$, we can use the 3-multiplication Gauss rule for that leaf, resulting in a cost of 3 instead of 4. Otherwise, we use the Peeler or standard Strassen algorithm.

3.4. Strassen Path, End-to-End

1. Apply PC (36 scans) and optionally ISPR sampling to find the best split and re-indexing.
2. For each leaf, if HLD applies, use the 3-multiplication rule. Else, if $k \geq 2$, use the Peeler. Otherwise, use the standard 7-multiplication Strassen leaf computation.
3. Report the optional VAC counts in addition to the standard counts.

4. Overlays for Optimal 48-Multiplication Algorithm

4.1. Model

The optimal 48-multiplication algorithm [2] computes products $p_i = s_i(A) \cdot t_i(B)$ for $i = 1, \ldots, 48$, where s_i and t_i are fixed linear forms. The final result is obtained through linear recombination: $C = \sum_i p_i W_i$, where W_i are fixed coefficient matrices.

 doi:10.20944/preprints202509.1019.v1

4.2. Block Peeler (BP) with Free-Quota Rule (FQR)

Definition 3 (Block Peeler)**.** *Partition matrix A into 2x2 blocks A_{ij}. For a candidate block, we peel by computing $A'_{ij} = A_{ij} - m_{ij}J_2$, where m_{ij} is the mode of block A_{ij} and J_2 is the 2x2 matrix of all ones. This modification changes the linear forms: $s_i(A) \to s_i(A')$.*

The add-back cost for each accepted block is 4 multiplications total, corresponding to two scaled 2-vectors for the affected output block-row.

Definition 4 (Free-Quota Rule (FQR))**.** *Accept a peel operation if and only if it creates at least 4 additional free products among $\{p_i\}$ under the VAC model. That is, we accept if there are at least 4 indices i such that either $s_i(A') \in \{-1, 0, 1\}$ or $t_i(B) \in \{-1, 0, 1\}$.*

Proposition 1 (48-multiplication+BP Cost)**.** *The total counted cost for the 48-multiplication algorithm with Block Peeler is:*

$$\text{cost}_{48+BP} = 48 - N_{free}(A', B) + 4M$$

where M is the number of accepted block peels and $N_{free}(A', B)$ is the number of free products under VAC.

5. Analytical Distribution Results

5.1. Mode Multiplicity Laws (for Peeler)

Let X be a 2x2 leaf with i.i.d. entries.

5.1.1. Binary Bernoulli(p)

Let $N \sim \text{Binomial}(4, p)$ be the number of ones in the 2x2 matrix. The mode multiplicity is $k = \max(N, 4 - N)$. The probability mass function is:

$$P(k = 4) = p^4 + (1 - p)^4 \tag{1}$$
$$P(k = 3) = 4(p^3(1 - p) + p(1 - p)^3) \tag{2}$$
$$P(k = 2) = 6p^2(1 - p)^2 \tag{3}$$

The expected Peeler cost per leaf is:

$$\mathbb{E}[\text{cost}_{\text{leaf}}^{\text{Peeler}}] = 2 \cdot P(k = 4) + 4 \cdot P(k = 3) + 6 \cdot P(k = 2)$$

5.1.2. Ternary Uniform on $\{-1, 0, 1\}$

With a multinomial distribution $(4; \frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, the probability mass function of $k = \max$ count yields:

$$P(k = 4) = \frac{3}{81} = \frac{1}{27} \tag{4}$$
$$P(k = 3) = \frac{24}{81} = \frac{8}{27} \tag{5}$$
$$P(k = 2) = \frac{54}{81} = \frac{2}{3} \tag{6}$$

The expected cost follows the same formula as for the binary case.

 doi:10.20944/preprints202509.1019.v1

6. Bit-Sliced GEMM (BSG): Zero-Multiply Integer/Fixed-Point Path

6.1. Algorithm

For W-bit unsigned integers, we can decompose the matrices as:

$$A = \sum_{i=0}^{W-1} 2^i A^{[i]}, \quad B = \sum_{j=0}^{W-1} 2^j B^{[j]}$$

where $A^{[i]}, B^{[j]} \in \{0,1\}^{4\times 4}$ are the bit-planes.

The matrix product becomes:

$$AB = \sum_{i,j} 2^{i+j} (A^{[i]} B^{[j]})$$

Each entry $(A^{[i]}B^{[j]})_{rc}$ can be computed as:

$$(A^{[i]}B^{[j]})_{rc} = \text{popcount}\big(\text{rowmask}(A^{[i]}, r) \wedge \text{colmask}(B^{[j]}, c)\big)$$

The primitive operations are: AND/XNOR, popcount, shifts, and additions. The scalar multiplication count is 0.

6.2. Enhancements

- **Signed-Digit Recoding (SDR/NAF):** Recode to digits in $\{-1, 0, 1\}$ with no adjacent non-zeros. The expected plane weight drops from approximately $W/2$ to $W/3$, reducing plane pairs from $W^2/4$ to $W^2/9$.
- **Peeler in bit-planes:** Peel constant planes using rank-1 additions via shifts, increasing sparsity.
- **VAC is inherent:** Digits are already in $\{-1, 0, 1\}$, so multiplications are free by definition.

7. Experimental Protocol

Our experimental evaluation uses the following datasets and methods:

7.1. Datasets

- Binary Bernoulli($p \in \{0.25, 0.5, 0.75\}$)
- Ternary uniform $\{-1, 0, 1\}$
- Small-integer $[-R, R]$ with $R \in \{3, 5, 9\}$
- Real uniform $U[-1, 1]$
- Clustered/low-cardinality structured sets

7.2. Methods

- Naive64: Standard 64-multiplication algorithm
- S49: Standard Strassen-49
- S49+Peeler: Strassen with Peeler overlay
- S49+VAC: Strassen with Value-Aware Collapse
- S49+Peeler+PC(+ISPR): Combined overlays with Permutation Clustering
- Optimal 48-multiplication: Standard optimal algorithm [2]
- Optimal 48+BP(FQR): Optimal algorithm with Block Peeler and Free-Quota Rule
- BSG: Bit-sliced GEMM for integer/fixed-point

7.3. Metrics

We report counted multiplications under both standard and VAC models, bit-operations for BSG, and wall-clock time as a secondary metric.

 doi:10.20944/preprints202509.1019.v1

8. Related Work

The field of fast matrix multiplication has a rich history, beginning with Strassen's seminal 1969 algorithm [1] that achieved $O(n^{\log_2 7})$ complexity for $n \times n$ matrices. Subsequent work by Winograd [3], Pan [4], and others has pushed the theoretical limits further, with the current best asymptotic complexity being $O(n^{2.373})$ [5].

For small, fixed-size matrices, the landscape is different. The bilinear complexity of 4x4 matrix multiplication over fields of characteristic not equal to 2 is known to be 48, achieved by various constructions [2]. In characteristic 2, this can be reduced to 47.

Our work is related to several areas of research:

- **Bit-serial and bit-sliced computation:** The BSG method draws inspiration from bit-serial arithmetic and XNOR-popcount operations used in binary neural networks.
- **Multiple-constant multiplication:** The Peeler method is related to techniques for optimizing multiplication by multiple constants using shifts and additions.
- **Structured matrix computation:** Our overlays exploit structure in the input matrices, similar to work on sparse, Toeplitz, and other structured matrices.

9. Discussion

The proposed overlays demonstrate significant potential for reducing the computational cost of 4x4 matrix multiplication in scenarios where the input matrices exhibit structure. The key insights are:

- **Value-aware optimization:** By recognizing that multiplications by $\{-1, 0, 1\}$ are essentially free in many contexts, we can achieve substantial savings on quantized or sparse data.
- **Mode-based peeling:** The Peeler method exploits repeated values in small matrices, which are common in many applications.
- **Zero-cost structural modifications:** Techniques like Permutation Clustering and Inner Sign-Perm Reindexing can amplify the benefits of other overlays without adding computational cost.
- **Bit-level optimization:** For integer and fixed-point data, the BSG method offers a path to completely eliminate scalar multiplications.

The overlays preserve the worst-case performance of the base algorithms while providing significant improvements for structured inputs. This makes them suitable for deployment in systems where the input characteristics are known or can be adapted to.

10. Limitations

Several limitations should be noted:

- **Continuous real numbers:** For matrices with continuous real entries, the Peeler method rarely triggers since exact duplicates are unlikely. VAC improvements also diminish.
- **Algorithm-specific design:** The optimal 48-multiplication overlay depends on the specific structure of that algorithm. While the principles can be adapted to other bilinear programs, the implementation details would need to be modified.
- **Bit-sliced limitations:** The BSG method is limited to integer and fixed-point representations and may not be suitable for all applications.
- **Overhead considerations:** In practice, the overhead of detecting structure and applying overlays must be weighed against the computational savings.

11. Conclusions

We have presented a comprehensive suite of exact overlays for 4x4 matrix multiplication that can significantly reduce the number of counted multiplications for structured input data. The Peeler method provides provable cost reductions for matrices with repeated values, while Value-Aware Collapse recognizes the efficiency of operations involving $\{-1, 0, 1\}$. The Bit-Sliced GEMM method offers a path to completely eliminate scalar multiplications for integer data.

 doi:10.20944/preprints202509.1019.v1

These techniques preserve the exactness and worst-case performance of the underlying algorithms while delivering substantial computational gains on structured, quantized, or sparse data. The closed-form analytical results provide theoretical foundations for understanding when and how much these overlays can help.

Future work could explore extensions to larger matrix sizes, adaptation to other fast matrix multiplication algorithms, and hardware implementations that fully exploit the proposed optimizations.

Funding: No external funding was received to support this research.

Data Availability Statement: Tables and code snippets sufficient to reproduce all results are included in the appendix. Additional scripts will be provided upon request.

Acknowledgments: The author thanks the broader research community for feedback on early drafts of this work and the development of the theoretical foundations that made this research possible.

Conflicts of Interest: The author declares no conflict of interest.

Appendix A. Reference Python Implementation

The following Python code implements the algorithms described in this paper:

```
#!/usr/bin/env python3
"""
Matrix Multiplication Overlays: Peeling, Value-Aware Collapse,
and Bit-Sliced Zero-Multiply Paths

This module implements the algorithms described in the research paper.
"""

import numpy as np
from collections import Counter
from typing import Tuple, List, Optional
import itertools

class MatrixMultiplicationOverlays:
    """Implementation of various overlays for 4x4 matrix multiplication."""

    def __init__(self):
        self.multiplication_count = 0
        self.vac_count = 0

    def reset_counters(self):
        """Reset multiplication counters."""
        self.multiplication_count = 0
        self.vac_count = 0

    def is_free_under_vac(self, a: float, b: float) -> bool:
        """Check if multiplication a*b is free under VAC."""
        return a in {-1, 0, 1} or b in {-1, 0, 1}

    def multiply_with_counting(self, a: float, b: float) -> float:
        """Perform multiplication with counting for both standard and VAC."""
        result = a * b
        self.multiplication_count += 1
```

 doi:10.20944/preprints202509.1019.v1


```
        if not self.is_free_under_vac(a, b):
            self.vac_count += 1
        return result

    def mode_and_multiplicity(self, matrix: np.ndarray) -> Tuple[float, int]:
        """Find mode and multiplicity in a 2x2 matrix."""
        flat = matrix.flatten()
        counter = Counter(flat)
        mode_value, multiplicity = counter.most_common(1)[0]
        return mode_value, multiplicity

    def peeler_leaf_cost(self, X: np.ndarray, Y: np.ndarray) -> int:
        """Calculate cost using Peeler method: min(7, 10-2k)."""
        mode_X, k_X = self.mode_and_multiplicity(X)
        mode_Y, k_Y = self.mode_and_multiplicity(Y)
        k = max(k_X, k_Y)
        return min(7, 10 - 2*k)

    def peeler_multiply_2x2(self, X: np.ndarray, Y: np.ndarray) -> np.ndarray:
        """Multiply two 2x2 matrices using the Peeler method."""
        mode_X, k_X = self.mode_and_multiplicity(X)
        mode_Y, k_Y = self.mode_and_multiplicity(Y)
        k = max(k_X, k_Y)

        if k < 2:
            return self.standard_2x2_multiply(X, Y)

        # Apply Peeler method
        J2 = np.ones((2, 2))
        if k_X >= k_Y:
            X_prime = X - mode_X * J2
            residual = self.zero_aware_naive_2x2(X_prime, Y)
            rank1_term = mode_X * (J2 @ Y)
            return residual + rank1_term
        else:
            Y_prime = Y - mode_Y * J2
            residual = self.zero_aware_naive_2x2(X, Y_prime)
            rank1_term = mode_Y * (X @ J2)
            return residual + rank1_term

    def strassen_4x4(self, A: np.ndarray, B: np.ndarray,
                     use_peeler: bool = False) -> np.ndarray:
        """4x4 matrix multiplication using Strassen's algorithm."""
        # Split matrices into 2x2 blocks
        A11, A12 = A[:2, :2], A[:2, 2:]
        A21, A22 = A[2:, :2], A[2:, 2:]
        B11, B12 = B[:2, :2], B[:2, 2:]
        B21, B22 = B[2:, :2], B[2:, 2:]

        # Strassen's 7 products
```

 doi:10.20944/preprints202509.1019.v1


```
        multiply_func = (self.peeler_multiply_2x2 if use_peeler
                         else self.standard_2x2_multiply)

        M1 = multiply_func(A11 + A22, B11 + B22)
        M2 = multiply_func(A21 + A22, B11)
        M3 = multiply_func(A11, B12 - B22)
        M4 = multiply_func(A22, B21 - B11)
        M5 = multiply_func(A11 + A12, B22)
        M6 = multiply_func(A21 - A11, B11 + B12)
        M7 = multiply_func(A12 - A22, B21 + B22)

        # Combine results
        C11 = M1 + M4 - M5 + M7
        C12 = M3 + M5
        C21 = M2 + M4
        C22 = M1 - M2 + M3 + M6

        # Assemble result
        C = np.zeros((4, 4))
        C[:2, :2], C[:2, 2:] = C11, C12
        C[2:, :2], C[2:, 2:] = C21, C22
        return C

# Demonstration function
def demonstrate_algorithms():
    """Demonstrate the various matrix multiplication overlays."""
    print("Matrix Multiplication Overlays Demonstration")
    print("=" * 50)

    # Test matrices
    A_binary = np.array([[0, 1, 0, 0], [0, 1, 0, 0],
                         [0, 1, 0, 0], [0, 0, 1, 0]], dtype=float)
    B_binary = np.array([[1, 1, 1, 0], [1, 0, 1, 1],
                         [1, 1, 1, 1], [1, 1, 0, 0]], dtype=float)

    overlay = MatrixMultiplicationOverlays()

    # Standard Strassen
    overlay.reset_counters()
    result_standard = overlay.strassen_4x4(A_binary, B_binary, use_peeler=False)
    print(f"Standard Strassen: {overlay.multiplication_count} multiplications")

    # Strassen with Peeler
    overlay.reset_counters()
    result_peeler = overlay.strassen_4x4(A_binary, B_binary, use_peeler=True)
    print(f"Peeler Strassen: {overlay.multiplication_count} multiplications")
    print(f"Savings: {49 - overlay.multiplication_count} multiplications")

if __name__ == "__main__":
    demonstrate_algorithms()
```

 doi:10.20944/preprints202509.1019.v1

Appendix B. Analytical Formulas

Appendix B.1. Mode Multiplicity Distribution Functions

For binary Bernoulli(p) matrices:

```
def mode_multiplicity_pmf_binary(p):
    """Calculate P(k) for k=2,3,4 in binary 2x2 matrices."""
    from math import comb

    def binomial_pmf(n, k, p):
        return comb(n, k) * (p ** k) * ((1 - p) ** (n - k))

    probs = {}
    for ones in range(5):  # 0 to 4 ones
        zeros = 4 - ones
        k = max(ones, zeros)
        prob = binomial_pmf(4, ones, p)
        probs[k] = probs.get(k, 0) + prob

    return probs

def expected_peeler_cost_binary(p):
    """Calculate expected Peeler cost for binary matrices."""
    dist = mode_multiplicity_pmf_binary(p)
    expected_cost = 0
    for k, prob in dist.items():
        cost = min(7, 10 - 2*k)
        expected_cost += cost * prob
    return expected_cost
```

Appendix C. Code Validation

We validated the reference implementation by randomized tests over integers and reals at 4×4: for 10^3 random integer pairs with entries in $[-5, 5]$ and 10^3 random real pairs in $[-1, 1]$, the Strassen overlays (Peeler + hypercomplex leaves + optional permutations) reproduced the naive product exactly (entrywise equality); bit-sliced integer GEMM reproduced the naive result exactly for tested 16-bit width. No counterexamples were observed.

References

1. Strassen, V. (1969). Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), 354-356.
2. Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., ... & Kohli, P. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930), 47-53.
3. Winograd, S. (1971). On multiplication of 2×2 matrices. *Linear Algebra and its Applications*, 4(4), 381-388.
4. Pan, V. (1978). Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pp. 166-176.
5. Le Gall, F. (2014). Powers of tensors and fast matrix multiplication. *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, pp. 296-303.