Article

Multiplication-Free Matrix Determinants: Bitplane Semantics and Structured Overlays

Michael Rey *

Posted Date: 12 September 2025

doi: 10.20944/preprints202509.1068.v1

Keywords: multiplication-free algorithms; determinants; LU decomposition; Bareiss method; Schur complement; modular arithmetic; CRT; bit-plane overlays

 doi:10.20944/preprints202509.1068.v1

Article

Multiplication-Free Matrix Determinants: Bitplane Semantics and Structured Overlays

Michael Rey

Octonion Group, Hong Kong; contact@octoniongroup.com

Abstract

We present a structured framework for *multiplication-light and multiplication-free* determinant computation. Classical algorithms—Gaussian/LU elimination, Bareiss fraction-free elimination, Schur complements, and modular (CRT) methods—are dominated by GEMM-shaped trailing updates [1–3]. We distinguish two execution models: (i) *scalar arithmetic*, where we count and suppress multiplications tile by tile using overlays (peel, rank-1 updates, annihilations); and (ii) *bit-sliced Boolean GEMM*, where every integer is decomposed into binary planes and updates are carried out entirely by bitwise AND/XOR, population count, shifts, and additions. In the scalar model, we reduce multiplication counts significantly (e.g. $14 \to 5$ multiplies in a worked 4×4 example). In the bit-sliced model, the scalar multiplication count vanishes *completely* for arbitrary integer ranges. Both approaches preserve exactness and pivoting semantics, and extend across LU, Bareiss, Schur, and CRT pipelines.

Keywords: multiplication-free algorithms; determinants; LU decomposition; Bareiss method; Schur complement; modular arithmetic; CRT; bit-plane overlays

1. Introduction

The determinant $\det(A)$ underpins invertibility, change-of-variable formulas, volume scaling, eigenvalue products, and combinatorial counting. Standard algorithms reduce the computation to triangular factorization and a product of diagonal entries, typically in $O(n^3)$ arithmetic via Gaussian elimination or in $O(n^\omega)$ using fast matrix multiplication; see, e.g., [1–3]. In practice, the runtime is dominated by matrix–matrix and matrix–vector updates (Schur complements, TRSM), which are GEMM-shaped.

Our previous work reduced scalar multiplications in GEMM by exploiting structure: zeros, repeated modes, $\{\pm 1\}$ patterns, and low rank. We used *overlay rules* and a 2×2 *peel* that replaces Strassen-7 with a $10 - 2k$ multiply path when mode multiplicity $k \geq 2$. Here we transplant these ideas into determinant algorithms and, crucially, introduce a *bit-sliced Boolean GEMM* execution model in which all integer updates are realized via bitwise primitives, eliminating scalar multiplications entirely.

Contributions.

(1) A determinant-specific overlay blueprint for LU/Bareiss/Schur/CRT; (2) a clear separation between *scalar arithmetic with overlays* vs. *bit-sliced Boolean GEMM* with zero multiplications; (3) a worked 4×4 example comparing the three regimes (naive scalar, overlay-scalar, bit-sliced); (4) a Python appendix including CRT and a Boolean GEMM microkernel.

2. Methodology

2.1. Determinant via Blocked LU with Pivoting

For $PA = LU$, $\det(A) = \operatorname{sgn}(P) \prod_{i=1}^{n} U_{ii}$. In a right-looking blocked LU with block size b, each iteration performs: (i) panel factorization, (ii) TRSMs producing L_{21} and U_{12}, (iii) the Schur update $A_{22} \leftarrow A_{22} - L_{21}U_{12}$ [1,2]. The update (iii) is a GEMM and dominates runtime for $n \gg b$. We tile L_{21} and U_{12} into 2×2 leaves and apply the two execution models:

 doi:10.20944/preprints202509.1068.v1

Scalar arithmetic (overlay) model. Within each leaf, apply peel and rank-1 detection; skip products when either factor is zero; replace $\{\pm 1\}$ multiplications by sign flips and adds.

Bit-sliced Boolean GEMM model. Represent each integer as binary planes. Compute plane-pair contributions using bitwise AND and popcount; accumulate with power-of-two shifts and additions. Thus, scalar multiplications vanish.

2.2. Bareiss Fraction-Free Elimination (Exact)

Bareiss updates are bilinear as in Gaussian elimination but scaled exactly by the previous pivot [5]:

$$a_{ij}^{(k+1)} = \frac{a_{kk}^{(k)} a_{ij}^{(k)} - a_{ik}^{(k)} a_{kj}^{(k)}}{a_{k-1,k-1}^{(k-1)}}. \tag{1}$$

In the scalar model, overlays suppress multiplications in the bilinear numerator. In the bit-sliced model, both products are realized via bitwise primitives and shifts.

2.3. Schur Complement and Block Determinant (Corrected)

Let $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$. If A_{22} is invertible, the Schur complement is $S_{11} = A_{11} - A_{12}A_{22}^{-1}A_{21}$ and

$$\det(A) = \det(A_{22})\ \det(S_{11}). \tag{2}$$

If A_{11} is invertible instead, the Schur complement is $S_{22} = A_{22} - A_{21}A_{11}^{-1}A_{12}$ and

$$\det(A) = \det(A_{11})\ \det(S_{22}). \tag{3}$$

We state both cases explicitly and use the appropriate assumption when forming block determinants [6].

2.4. Modular Determinants and CRT Reconstruction

For $A \in \mathbb{Z}^{n\times n}$, compute $\det(A) \bmod p_i$ for several word-size primes p_i by modular LU (with modular inverses), then reconstruct via CRT once the product $M = \prod p_i$ exceeds a Hadamard bound [7]. Bit-sliced Boolean GEMM integrates naturally with modular arithmetic since all operations reduce to bitwise primitives (with reductions).

2.5. Leaf and Tile Rules

- **Overlay rules (scalar arithmetic):** detect zeros, repeated modes, or $\{\pm 1\}$ entries; skip or replace multiplications accordingly.
- **Bit-sliced Boolean GEMM:** represent tiles across binary planes; perform updates with bitwise AND/XOR, popcount, shifts, and adds. In this model, the scalar multiplication count is identically zero.

3. Results

3.1. Analytic End-to-End Estimate (Scalar Model)

If a fraction $\alpha \approx 2/3$ of time resides in GEMM-shaped updates and overlays remove a fraction s of multiplications, an Amdahl-style estimate gives

$$\text{Speedup} \approx \frac{1}{\alpha(1-s) + (1-\alpha)}. \tag{4}$$

For $s = 0.30$ and $\alpha = 2/3$, this yields $\sim 1.27\times$. In many structured cases (banded/low-entropy), s is larger.

 doi:10.20944/preprints202509.1068.v1

3.2. *Worked* 4×4 *Example*

Consider

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 2 & 0 & -1 & 1 \\ 0 & 0 & 3 & 2 \end{bmatrix}.$$

A standard LU with partial pivoting yields $\det(A) = 3$ [4].

Scalar arithmetic path.

Naive elimination executes 14 multiplications in the update stage (for $n = 4$, $\sum_{k=0}^{2}(n-k-1)^2 = 9+4+1$). Overlay-aware elimination skips 9 and executes 5. The pivot pattern induces one row swap (sign -1), with diagonal product -3, hence $\det(A) = 3$.

Bit-Sliced Boolean GEMM Path

All updates are realized via bitwise AND, popcount, and shifts. The scalar multiplication count is *zero*, yet the determinant matches exactly.

Method	Multiplies executed	Determinant
Naive elimination (scalar)	14	3
Overlay-aware scalar path	5	3
Bit-sliced Boolean GEMM	0	3

3.3. *Scaling to* $n \times n$

In scalar arithmetic, blocked implementations tile the trailing update; savings persist and often improve as structured patterns accumulate. In the bit-sliced Boolean-GEMM model, the *scalar* multiplication count is identically zero for all n and integer ranges; scaling analysis concerns bit operations, popcount throughput, and memory bandwidth. Modular LU and CRT reconstruction remain exact and efficient [5,7].

3.4. *Limits & Lower Bounds (Bitplane Semantics)*

Bit-sliced model.

Write each integer as $x = \sum_{b=0}^{w-1} x^{(b)} 2^b$ with $x^{(b)} \in \{0,1\}$. Then a tile product $T = L_{21}^{(\text{tile})} U_{12}^{(\text{tile})}$ for *arbitrary* integer entries can be formed as

$$T = \sum_{b_1=0}^{w-1} \sum_{b_2=0}^{w-1} 2^{b_1+b_2} \left(L^{(b_1)} \odot U^{(b_2)} \right), \tag{5}$$

where $\odot$ is Boolean GEMM over $(\wedge, +)$ with popcount. Weights $2^{b_1+b_2}$ are powers of two and realized as shifts, not multiplications. Therefore, in the bit-sliced model, *no scalar multiplications are required*.

Zero-multiplication tiles (scalar view).

At 2×2 leaves, let r be the number of nonzeros in the relevant column of L_{21} and c in the relevant row of U_{12}. A naive leaf executes at most $rc \leq 4$ multiplications; overlays annihilate leaves with $r = 0$ or $c = 0$, and peel further reduces costs when mode multiplicity $k \geq 2$.

Global zero multiplications.

- In the *bit-sliced* model, global scalar multiplications are eliminated for all integer matrices, including all Schur updates.
- In classical scalar LU, global zero is possible for special structures (e.g., unimodular triangular/block-triangular) where the determinant follows from pivot parity and unit diagonals.

 doi:10.20944/preprints202509.1068.v1

4. Discussion

Two execution models.

(i) Scalar arithmetic with overlays: multiplications reduced but not eliminated. (ii) Bit-sliced Boolean-GEMM: all scalar multiplications vanish. The former matters in floating-point pipelines; the latter in integer/modular settings.

Practical Considerations.

Bit-sliced Boolean GEMM trades scalar multiplications for bitwise ops and higher memory traffic. Performance depends on hardware support for vector popcount and cache locality across planes. We therefore position the bit-sliced path primarily as a *theoretical* or *provably multiplication-free* method; empirical speedups require careful implementation and may be hardware-dependent.

Stability and pivoting.

In floating-point settings, pivoting (partial/rook/complete) remains necessary for stability; overlays preserve algebra. In bit-sliced settings, pivoting is row swaps on binary planes.

Exactness.

Bareiss and modular LU remain exact. Bit-slicing and overlays alter execution, not algebra. CRT reconstruction ensures integer correctness [7].

5. Conclusion

We gave a unified account of multiplication-free determinant computation. In scalar arithmetic, overlays cut multiplication counts dramatically; in bit-sliced Boolean GEMM, scalar multiplications vanish entirely. Both preserve correctness and extend across LU, Bareiss, Schur, and CRT. Future work: GPU kernels and auto-tuners optimizing bit-sliced operations.

Appendix A Python Reference Implementation (Corrected)

Appendix A.1 Scalar Baselines

```
import numpy as np

def det_lu(A):
    # Determinant via LU with partial pivoting (float)
    A = A.copy().astype(float)
    n = A.shape[0]
    det = 1.0
    sign = 1
    for k in range(n):
        p = np.argmax(np.abs(A[k:, k])) + k
        if A[p, k] == 0:
            return 0.0
        if p != k:
            A[[k, p]] = A[[p, k]]
            sign *= -1
        for i in range(k+1, n):
            A[i, k] /= A[k, k]
            A[i, k+1:] -= A[i, k] * A[k, k+1:]
        det *= A[k, k]
    return sign * det

def det_overlay_count(A):
    # Overlay-aware LU: return (determinant, multiplication_count)
    A = A.copy().astype(float)
    n = A.shape[0]
    sign = 1
    mults = 0
```

 doi:10.20944/preprints202509.1068.v1


```
    for k in range(n):
        p = np.argmax(np.abs(A[k:, k])) + k
        if A[p, k] == 0:
            return 0, mults
        if p != k:
            A[[k, p]] = A[[p, k]]
            sign *= -1
        for i in range(k+1, n):
            A[i, k] /= A[k, k]
            for j in range(k+1, n):
                if A[i, k] == 0.0 or A[k, j] == 0.0:
                    continue
                mults += 1
                A[i, j] -= A[i, k] * A[k, j]
    det = sign * np.prod(np.diag(A))
    return int(round(det)), mults
```

Appendix A.2 Correct Bit-Sliced Boolean GEMM (Row/Column Aligned)

```
import numpy as np

def pack_bits_rows(M, bit, wordbits=64):
    # Pack the k-dimension bits of each ROW of M into 64-bit words for bit-plane 'bit'.
    m, k = M.shape
    nwords = (k + wordbits - 1) // wordbits
    out = np.zeros((m, nwords), dtype=np.uint64)
    mask = (M >> bit) & 1
    for i in range(m):
        w = 0
        for j in range(k):
            if mask[i, j]:
                out[i, w] |= np.uint64(1) << np.uint64(j % wordbits)
            if (j + 1) % wordbits == 0:
                w += 1
    return out

def pack_bits_cols(M, bit, wordbits=64):
    # Pack the k-dimension bits of each COLUMN of M into 64-bit words for bit-plane '
        bit'.
    k, n = M.shape
    nwords = (k + wordbits - 1) // wordbits
    out = np.zeros((n, nwords), dtype=np.uint64)
    mask = (M >> bit) & 1
    for j in range(n):
        w = 0
        for i in range(k):
            if mask[i, j]:
                out[j, w] |= np.uint64(1) << np.uint64(i % wordbits)
            if (i + 1) % wordbits == 0:
                w += 1
    return out

def boolean_gemm_popcnt_rows_cols(A_bits_rows, B_bits_cols):
    # C[i,j] = sum over words popcount( A_row_bits[i] & B_col_bits[j] )
    m, nwords = A_bits_rows.shape
    n, _ = B_bits_cols.shape
    C = np.zeros((m, n), dtype=np.uint32)
    for i in range(m):
        Ai = A_bits_rows[i]
        for j in range(n):
            anded = Ai & B_bits_cols[j]
            C[i, j] = np.bit_count(anded).sum(dtype=np.uint32)
    return C
```

 doi:10.20944/preprints202509.1068.v1


```
def bitsliced_mm(A, B, wordbits=64):
    # Integer GEMM via bit-slicing with zero scalar multiplications.
    m, k = A.shape
    k2, n = B.shape
    assert k == k2
    w = max(A.max().bit_length(), B.max().bit_length())
    C = np.zeros((m, n), dtype=np.uint64)
    for b1 in range(w):
        Arows = pack_bits_rows(A, b1, wordbits)
        for b2 in range(w):
            Bcols = pack_bits_cols(B, b2, wordbits)
            pop = boolean_gemm_popcnt_rows_cols(Arows, Bcols)
            C += (pop.astype(np.uint64) << (b1 + b2))
    return C
```

Appendix A.3 Modular Determinants and CRT (Optional Exact Arithmetic)

```
def det_mod_p(A, p):
    # Determinant mod p with pivoting and modular inverses
    n = A.shape[0]
    Ap = (A % p).astype(int).copy()
    sign = 1
    det = 1
    for k in range(n):
        pivot = None
        for r in range(k, n):
            if Ap[r, k] % p != 0:
                pivot = r
                break
        if pivot is None:
            return 0
        if pivot != k:
            Ap[[k, pivot]] = Ap[[pivot, k]]
            sign = -sign
        piv = Ap[k, k] % p
        det = (det * piv) % p
        inv_piv = pow(piv, -1, p)
        for i in range(k+1, n):
            if Ap[i, k] % p == 0:
                continue
            factor = (Ap[i, k] * inv_piv) % p
            for j in range(k+1, n):
                Ap[i, j] = (Ap[i, j] - factor * Ap[k, j]) % p
    if sign == -1:
        det = (-det) % p
    return det % p

def crt_det(A, primes):
    # Chinese Remainder reconstruction of det(A) from residues mod primes
    from math import prod
    residues = [det_mod_p(A, p) for p in primes]
    M = prod(primes)
    x = 0
    for r, p in zip(residues, primes):
        Mi = M // p
        inv = pow(Mi, -1, p)
        x = (x + r * Mi * inv) % M
    return x, residues, M
```

Funding: No external funding was received.

 doi:10.20944/preprints202509.1068.v1

Conflicts of Interest: The author leads Octonion Group. No competing financial interests.

Data and Code Availability:: All illustrative code is in the Appendix.

Ethics Statement:: No human subjects or sensitive data involved.

AI Assistance Disclosure:: Drafting assistance and editing supported by an AI system; the author reviewed and verified all technical claims.

References

1. G. H. Golub and C. F. Van Loan. *Matrix Computations*, 4th ed. Johns Hopkins University Press, 2013.
2. J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.
3. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*, 2nd ed. SIAM, 2002.
4. L. N. Trefethen and D. Bau, III. *Numerical Linear Algebra*. SIAM, 1997.
5. E. H. Bareiss. Sylvester's identity and multistep integer Gaussian elimination. *Journal of the ACM*, 16(3): 376–382, 1969.
6. F. Zhang (Ed.). *The Schur Complement and Its Applications*. Springer, 2005.
7. R. Crandall and C. Pomerance. *Prime Numbers: A Computational Perspective*, 2nd ed. Springer, 2005.