Multiplication-Free Graph Spectral Embeddings: Bit-Sliced Repeated AX and $A^\top AX$ Products

Michael Rey *

Posted Date: 15 September 2025

doi: 10.20944/preprints202509.1160.v1

Keywords: graph embedding; Laplacian eigenmaps; spectral clustering; Boolean GEMM; bit slicing

 doi:10.20944/preprints202509.1160.v1

Article

Multiplication-Free Graph Spectral Embeddings: Bit-Sliced Repeated AX and $A^\top AX$ Products

Michael Rey

Octonion Group, Hong Kong; contact@octoniongroup.com

Abstract

Graph spectral embedding methods, such as spectral clustering and Laplacian Eigenmaps, are powerful techniques for uncovering the underlying structure of data by analyzing the eigenvectors of graph matrices. A core computational bottleneck in these methods is the repeated multiplication of a graph's adjacency or Laplacian matrix with a vector, as seen in iterative eigensolvers like the power method. This paper presents a multiplication-free approach to accelerate these critical operations. By representing the graph matrix with integer weights and employing a bit-slicing technique, the matrix-vector products are decomposed into a series of highly efficient Boolean matrix multiplications. This allows the computationally intensive steps to be executed with zero scalar multiplications, relying instead on fast bitwise operations. We demonstrate the method's validity by providing an executable Python implementation for the $A^\top AX$ product, a key kernel in many spectral methods.

Keywords: graph embedding; Laplacian Eigenmaps; spectral clustering; Boolean GEMM; bit slicing

1. Introduction

Spectral methods provide a powerful framework for analyzing the structure of graphs. Techniques like spectral clustering [2,3] and Laplacian Eigenmaps [1] have become standard tools in machine learning and data analysis for tasks such as community detection, dimensionality reduction, and manifold learning. These methods derive meaningful low-dimensional embeddings of the data by computing the eigenvectors of a graph's adjacency matrix or, more commonly, its Laplacian matrix. The eigenvectors corresponding to the smallest eigenvalues capture the smoothest possible mappings of the graph onto a lower-dimensional space, thus revealing the underlying cluster or manifold structure.

A key computational step in these spectral methods is the calculation of these eigenvectors, which is often performed using iterative algorithms like the power method or the Lanczos algorithm [5]. These algorithms require repeated multiplications of the graph matrix (e.g., the adjacency matrix A or the Laplacian L) with a vector. For large graphs, these matrix-vector products, particularly operations like AX and $A^\top AX$, dominate the computational cost. This paper introduces a bit-slicing approach that transforms these expensive matrix multiplications into a series of highly efficient Boolean operations, thereby offering a multiplication-free path to accelerating graph spectral embedding algorithms.

2. Method

The bit-slicing methodology is applied directly to the core matrix operations required by spectral embedding algorithms. The primary operations of interest are the matrix-vector products AX and the symmetric product $A^\top AX$, where A is the graph's adjacency matrix and X is a matrix of vectors.

Assuming the adjacency matrix A contains integer weights (with binary graphs being a special case where weights are 0 or 1), the product AX is computed using the `bitsliced_mm` function described in the Appendix. This function decomposes both matrices A and X into their constituent bitplanes. It then computes the product by performing Boolean matrix multiplications on these bitplanes and accumulating the results with appropriate bit shifts. The product $A^\top AX$ is simply computed as a nested application of the same function: first computing $Y = AX$, and then computing $A^\top Y$.

 doi:10.20944/preprints202509.1160.v1

For methods that use the graph Laplacian, $L = D - A$, where D is the diagonal degree matrix, we can compute the product LX without explicitly forming L. The product is calculated as $LX = DX - AX$. The AX term is computed as described above. The DX product is a simple element-wise multiplication of the vectors in X by the corresponding diagonal entries of D, which is computationally inexpensive. The final result is obtained by subtracting the two resulting matrices. This avoids the need to perform bit-sliced multiplication on the potentially denser Laplacian matrix, leveraging the sparsity of the adjacency matrix instead.

3. Computational Complexity

The computational advantage of the bit-slicing approach stems from the complete elimination of scalar multiplications within its core loop. The standard method for computing a matrix-product like AX is dominated by multiply-accumulate operations, with a complexity of $O(n^3)$ for general $n \times n$ matrices, or $O(n^2)$ for sparse matrices with an average of n non-zero entries per row [4].

In our proposed method, the computation is shifted to the bit-domain. The cost is determined by the number of bitwise operations (AND, POPCNT) and additions. For an $m \times k$ adjacency matrix A and a $k \times n$ vector matrix X, with data quantized to w bits, the complexity is proportional to $m \cdot n \cdot k \cdot w^2$. While the dependency on w^2 might seem unfavorable, the extreme efficiency of bitwise operations on modern hardware, especially when processing 64 or 128 bits in parallel using SIMD instructions, leads to substantial performance gains in practice. The arithmetic consists only of bit shifts and additions to accumulate the final result, making the core of the algorithm entirely multiplication-free.

4. Numerical Example

To demonstrate the bit-slicing process for a graph kernel, let's consider the computation of $A^\top AX$ with a simple binary adjacency matrix A and an integer vector X:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, X = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

First, we compute the product $Y = AX$ using the bit-sliced method. The matrix A is already binary. We decompose X into its bitplanes. The maximum value in X is 2, so we need 2 bits ($w = 2$):

$$X_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (LSB plane)}, X_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (MSB plane)}$$

The Boolean products for each bitplane are:

- Bit 0: $A \cdot X_0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$
- Bit 1: $A \cdot X_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

We accumulate these results with the appropriate bit shifts to get $Y = AX$:

$$Y = (2^0 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}) + (2^1 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Next, we compute $A^\top Y$. Since A is symmetric, $A^\top = A$. We decompose our intermediate result Y into bitplanes:

$$Y_0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ (LSB plane)}, Y_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ (MSB plane)}$$

The Boolean products are:

- Bit 0: $A^\top \cdot Y_0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- Bit 1: $A^\top \cdot Y_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

Finally, we accumulate the results to get the final vector:

$$A^\top AX = (2^0 \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}) + (2^1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

 doi:10.20944/preprints202509.1160.v1

This result matches the standard integer matrix multiplication, and the process was completed without any scalar multiplications, as validated by the executable code in the Appendix.

5. Discussion

The bit-sliced approach to graph spectral embedding offers a compelling trade-off between computational efficiency and numerical precision. The use of an integer-based, multiplication-free core provides several advantages. For unweighted graphs, where the adjacency matrix is naturally binary, the computations are exact, avoiding the potential pitfalls of floating-point inaccuracies. This is a significant benefit for algorithms where the precise structure of the graph is critical.

For weighted graphs, the method requires the edge weights to be quantized to a fixed-point integer representation. The choice of the number of bits for quantization presents a trade-off: a higher number of bits allows for a more precise representation of the weights but increases the computational load and memory footprint of the bit-slicing algorithm. However, for many applications, a relatively low-precision representation is sufficient to capture the essential structure of the graph, making this a viable and attractive option.

While the bit-slicing approach can be memory-intensive due to the expansion of the input matrices into bitplanes, there are opportunities for optimization. For sparse graphs, many of the bitplanes will also be sparse. This sparsity can be exploited by using compressed data structures for the bitplanes, which would reduce memory consumption and could also accelerate the Boolean matrix multiplication. Furthermore, the inherent parallelism of the bit-sliced operations makes the method highly suitable for implementation on modern hardware with wide SIMD units and dedicated bit-manipulation instructions.

In conclusion, the multiplication-free approach presented in this paper provides a promising avenue for accelerating spectral graph algorithms. By shifting the computational burden from expensive multiplications to highly efficient bitwise operations, it opens the door to the analysis of larger graphs and the development of more efficient machine learning and data analysis pipelines.

Disclosures

Funding: None. **Conflicts of Interest:** The author declares no conflict of interest. **Data/Code:** Fully executable code in Appendix. **AI Assistance:** Drafting assisted by AI; author verified all claims.

Appendix A Python (Executable)

```
import numpy as np

def popcount_sum_words(words):
    total = 0
    for w in words:
        v = int(w)
        if hasattr(v, "bit_count"):
            total += v.bit_count()
        else:
            total += bin(v).count("1")
    return np.uint32(total)

def pack_bits_rows(M, bit, wordbits=64):
    m, k = M.shape
    nwords = (k + wordbits - 1) // wordbits
    out = np.zeros((m, nwords), dtype=np.uint64)
    if m == 0 or k == 0:
        return out
    mask = (M >> bit) & 1
    for i in range(m):
        w = 0
        for j in range(k):
```

 doi:10.20944/preprints202509.1160.v1


```
            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):
    k, n = M.shape
    nwords = (k + wordbits - 1) // wordbits
    out = np.zeros((n, nwords), dtype=np.uint64)
    if k == 0 or n == 0:
        return out
    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):
    m, nwords = A_bits_rows.shape
    n, _ = B_bits_cols.shape
    C = np.zeros((m, n), dtype=np.uint32)
    if m == 0 or n == 0 or nwords == 0:
        return C
    for i in range(m):
        Ai = A_bits_rows[i]
        for j in range(n):
            anded = Ai & B_bits_cols[j]
            C[i, j] = popcount_sum_words(anded)
    return C

def bitsliced_mm(A, B, wordbits=64):
    if A.ndim != 2 or B.ndim != 2:
        raise ValueError("A and B must be 2D arrays")
    m, k = A.shape
    kb, n = B.shape
    if kb != k:
        raise ValueError(f"inner dims mismatch: {k} vs {kb}")
    if m == 0 or n == 0 or k == 0:
        return np.zeros((m, n), dtype=np.uint64)
    Au = A.astype(np.uint64, copy=False)
    Bu = B.astype(np.uint64, copy=False)
    wA = int(Au.max().bit_length()) if Au.size else 0
    wB = int(Bu.max().bit_length()) if Bu.size else 0
    w = max(wA, wB)
    if w == 0:
        return np.zeros((m, n), dtype=np.uint64)
    C = np.zeros((m, n), dtype=np.uint64)
    for b1 in range(w):
        Arows = pack_bits_rows(Au, b1, wordbits)
        for b2 in range(w):
            Bcols = pack_bits_cols(Bu, b2, wordbits)
            pop = boolean_gemm_popcnt_rows_cols(Arows, Bcols)
            C += (pop.astype(np.uint64) << (b1 + b2))
    return C


def AtAX(A, X):
    AX = bitsliced_mm(A, X)
    return bitsliced_mm(A.T, AX)
```

 doi:10.20944/preprints202509.1160.v1


```
if __name__ == "__main__":
    A = np.array([[0,1],[1,0]], dtype=np.uint64)
    X = np.array([[1],[2]], dtype=np.uint64)
    AtAX_bs = AtAX(A, X)
    AtAX_ref = (A.T @ (A @ X)).astype(np.uint64)
    print("A^T A X bit-sliced correct:", np.array_equal(AtAX_bs, AtAX_ref))
```

References

1. M. Belkin and P. Niyogi. Laplacian Eigenmaps. *Neural Computation*, 2003.
2. A. Ng, M. Jordan, Y. Weiss. On spectral clustering. *NIPS*, 2002.
3. U. von Luxburg. A tutorial on spectral clustering. *Stat. Comput.*, 2007.
4. L. N. Trefethen and D. Bau III. *Numerical Linear Algebra*, SIAM, 1997.
5. G. H. Golub and C. F. Van Loan. *Matrix Computations*, 4th ed., 2013.