Accelerating Graph Neural Networks Using a Novel Computation-Friendly Matrix Compression Format

João N. F. Alves*§†‡, Samir Moustafa*§, Siegfried Benkner§,
Alexandre P. Francisco †‡, Wilfried N. Gansterer§, Luís M. S. Russo†‡
* University of Vienna, Doctoral School Computer Science, Vienna, Austria
§ University of Vienna, Faculty of Computer Science, Vienna, Austria
† Universidade de Lisboa, Instituto Superior Técnico, Lisbon, Portugal
‡ INESC-ID Lisboa, Lisbon, Portugal
{joao.ferreira.alves, samir.moustafa, siegfried.benkner, wilfried.gansterer}@univie.ac.at,
{luis.russo, alexandre.francisco}@tecnico.ulisboa.pt

***Abstract*—This paper proposes the Compressed Binary Matrix (CBM) format, a novel, computation-friendly compression scheme for binary matrices. CBM not only reduces the memory footprint of the matrix but also enables faster matrix multiplication between binary and dense, real-valued matrices. The CBM format can be applied to accelerate various graph-related tasks, where the (binary) adjacency matrix of the graph is repeatedly multiplied by another matrix, such as during inference and training of various types of Graph Neural Networks (GNNs). The format is evaluated on a shared-memory architecture in both serial and parallel settings. Experimental results show that CBM can reduce the memory footprint of real-world graphs up to 11×, and that the parallel matrix multiplication using CBM is more than 5× faster than state-of-the-art sparse-dense matrix multiplication kernels. Furthermore, when applied to the inference stage of Graph Convolutional Networks (GCNs), the CBM format achieves speedups close to 2.5× compared to inference using other parallel matrix multiplication kernels.**

***Index Terms*—Binary Matrix Product, Graph Compression, Graph Neural Networks**

I. Introduction

Let $\mathbf{A} \in \{0,1\}^{n \times n}$ be an $n \times n$ binary matrix. Matrix products involving $\mathbf{A}$ are a fundamental building block and a major execution hot-spot in many graph algorithms, when $\mathbf{A}$ represents the graph's adjacency matrix. An important example that we use to motivate this work, is the inference and training of various Graph Neural Network (GNN) architectures where $\mathbf{A}$ is repeatedly multiplied with the current nodes' embedding.

Graphs commonly encountered in GNN applications, such as social or collaboration networks, tend to be large and very sparse. Therefore, popular machine learning frameworks represent matrix $\mathbf{A}$ in standard sparse matrix formats, such as Coordinate List (COO) or Compressed Sparse Row (CSR), to accelerate GNN inference and training. These formats only represent the non-zero elements in $\mathbf{A}$, allowing us to store the matrix in space proportional to its number of non-zero elements, i.e., $\mathcal{O}(\mathbf{nnz}(\mathbf{A}))$ space. More importantly, standard sparse formats enable efficient sparse-dense matrix multiplication kernels that only require $\mathcal{O}(p \cdot \mathbf{nnz}(\mathbf{A}))$ scalar operations, where p represents the number of columns of the dense operand matrix.

While standard sparse matrix formats reduce memory and computational cost compared to dense matrix representations, previous work show that there are more efficient computation-friendly compression schemes for binary matrices [1]–[3]. In other words, it is possible to compress a binary matrix $\mathbf{A}$ beyond what sparsity alone allows, and still perform the computation of the matrix product in a number of scalar operations proportional to the size of the compressed representation, i.e., less than $\mathbf{nnz}(\mathbf{A})$. In this work we propose the Compressed Binary Matrix (CBM) format, a new computation-friendly compression scheme for binary matrices. The key advantage of CBM is that it only represents the elements that differ (or deltas) from a row of $\mathbf{A}$ with respect to another similar row of the same matrix, which for many use cases tends to be substantially smaller than $\mathbf{nnz}(\mathbf{A})$. This property makes CBM particularly well-suited for compressing and accelerating matrix products involving the adjacency matrix of social and collaboration networks, where neighboring nodes often share very similar neighborhoods. This makes the CBM format particularly interesting for various graph algorithms, including the inference and training of several GNN architectures.

A. Our Contributions

In this paper, we propose the Compressed Binary Matrix (CBM) format, an efficient computation-friendly compression scheme for binary matrices $\mathbf{A} \in \{0,1\}^{n \times n}$. We also show that our format can be easily extended to represent matrices that can be factorized as $\mathbf{AD}$ or $\mathbf{DAD}$, where $\mathbf{D} \in \mathbb{R}^{n \times n}$ is a diagonal matrix. Then, we show how to carry out fast matrix multiplication between the various types of matrices that can be represented in CBM format and a dense real-valued matrix, in sequential and in parallel shared-memory environments. Additionally, we prove that the number of scalar

We would like to express our gratitude to Tomás Almeida, who helped us refactoring our benchmarks and GCN pipeline; and to Martin Sandrieser, who helped us setting the experimental environment. This work was supported by the Innovation Study CBM4scale through the Inno4scale project, funded by the European High-Performance Computing Joint Undertaking (JU) under Grant Agreement No. 101118139. The JU receives support from the European Union's Horizon Europe Programme. This work was also partly supported by national funds through FCT – Fundação para a Ciência e Tecnologia, under project http://doi.org/10.54499/LA/P/0078/2020.

operations required for matrix multiplication using the CBM format is never greater than that required when using standard sparse matrix formats.

We implemented the CBM format and corresponding matrix multiplication kernels in C++ and OpenMP in such a way that they can be used together with PyTorch [4], a state-of-the-art Deep Learning framework. Experimental evaluation using real-world datasets demonstrates the effectiveness of our approach. Our format reduced the memory footprint of the original adjacency matrix by up to $11\times$, resulting in a speedup above $5\times$ compared to state-of-the-art sparse dense kernels in parallel shared-memory CPU architectures. Finally, matrix multiplication using the CBM format, together with PyTorch, reduced the inference time of a two-layer GCN by $2.48\times$. Our implementation is available at https://github.com/cbm4scale.

As far as we are aware, the CBM format is the first computation-friendly compression scheme that considers the parallel matrix product between a binary and dense real-valued matrix. Furthermore, our format overcomes the limitations of previous works (see Section VII) by exploiting compression opportunities along complete rows of $\mathbf{A}$, being able to be constructed in polynomial time, not requiring additional memory during matrix multiplication, and ensuring that in the worst-case scenario matrix multiplication with CBM does not require more scalar operations than standard sparse formats.

II. Motivation: Graph Neural Networks (GNNs)

Currently, GNNs are the preferred tool to learn from graph-structured data and thus are considered key for future artificial intelligence tasks including node-classification, graph-classification, and link-prediction. The training and inference time of many GNN architectures is dominated by long sequences of matrix products [5]–[7]. This is particularly evident in GNNs that resort to message passing layers, where in each layer the nodes of the graph aggregate the feature vectors (embeddings) of neighboring nodes and adjust their own embedding based on the information collected. In some variants of GNNs, such as the widely used Graph Convolutional Networks (GCNs) [5], Graph Isomorphism Networks (GINs) [7], and GraphSAGE [8] the message produced in each layer is essentially the product of the adjacency matrix of the graph and its node's embedding. Our work is motivated by the potential impact of the CBM format on the inference and training of these GNNs.

Without loss of generality, we demonstrate the efficacy of our method by using it in the inference of a two-layer GCN, where each node aggregates information from its neighbors, which in their turn also aggregate information from their own neighbors. Since the aggregation is done using summation, this can be reduced to a sparse-dense matrix multiplication between the graph's adjacency and the embedding of the current layer. The inference of a two-layer GCN requires two sparse-dense matrix multiplications, two dense-dense matrix multiplications and an element-wise activation function:

$$\hat{\mathbf{A}}\ \sigma(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^0)\mathbf{W}^1, \tag{1}$$

where $\hat{\mathbf{A}}$ represents the normalized Laplacian adjacency matrix of the graph, such that $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}(\mathbf{A}+\mathbf{I})\mathbf{D}^{-\frac{1}{2}}$, $\mathbf{D}$ is the degree diagonal matrix of the graph, σ denotes a ReLU activation function, $\mathbf{X}$ is the matrix of node features, and $\mathbf{W}^0$ and $\mathbf{W}^1$ are learnable dense matrices for the first and second layers [9]. During training, there is a sequence of sparse-dense matrix multiplications between $\hat{\mathbf{A}}$ and the gradients as well [10]. Note that $(\mathbf{A}+\mathbf{I})$ simply adds self-loops to the nodes of the graph. Therefore, if the graph is unweighted, $(\mathbf{A}+\mathbf{I})$ must be a binary matrix. In real scenarios, $\hat{\mathbf{A}}$ tends to be much larger than both $\mathbf{X}$ and $\mathbf{W}$, meaning it is reasonable to assume that matrix products involving $\hat{\mathbf{A}}$ represent most of the computational burden associated with inference [11]. This suggests that, by representing $\hat{\mathbf{A}}$ in the CBM format, we will be able to substantially accelerate matrix products involving $\hat{\mathbf{A}}$, consequently reducing the inference time of GCN models.

III. Compressed Binary Matrix (CBM) Format

Let $\mathbf{A} \in \{0,1\}^{n\times n}$ be a binary matrix, and let $\mathbf{A}_{i,:}$ denote the i-th row vector of $\mathbf{A}$ for $i = 1,\ldots,n$. Additionally, assume that $\mathbf{A}_{i,:}$ is represented as an adjacency list containing the column indices of its non-zero elements. Then, it is clear that any $\mathbf{A}_{x,:}$ can be expressed in terms of a another row vector $\mathbf{A}_{y,:}$ and two **sets of deltas** ($\Delta^+_{x,y}$ and $\Delta^-_{x,y}$). These sets indicate which column indices of $\mathbf{A}_{y,:}$ must be set to 1 or 0, respectively, to obtain $\mathbf{A}_{x,:}$:

$$\mathbf{A}_{x,:} = (\mathbf{A}_{y,:} \cup \Delta^+_{x,y}) \setminus \Delta^-_{x,y}. \tag{2}$$

Equation 2 suggests that representing $\mathbf{A}_{x,:}$ requires space proportional to the number of deltas, when $\mathbf{A}_{y,:}$ is already in memory. Assuming $\mathbf{A}_{x,:}$ and $\mathbf{A}_{y,:}$ are similar, then it is likely that the number of deltas required to represent $\mathbf{A}_{x,:}$ will be smaller than $\mathbf{nnz}(\mathbf{A}_{x,:})$. If this is the case, it would be more efficient to represent $\mathbf{A}_{x,:}$ with respect to $\mathbf{A}_{y,:}$ than by resorting to standard sparse formats.

The Compressed Binary Matrix (CBM) format leverages the intuition provided in Equation 2 to reduce the memory footprint of a binary matrix, such as $\mathbf{A}$, beyond $\mathcal{O}(\mathbf{nnz}(\mathbf{A}))$. To achieve this objective, the compression algorithm of the CBM format first identifies a suitable **compression tree** for matrix $\mathbf{A}$. This is, for each row $\mathbf{A}_{x,:}$ identify another row $\mathbf{A}_{y,:}$ that characterizes the former, such that the number of deltas required to represent $\mathbf{A}_{x,:}$ is both **(1)** minimized with respect to $\mathbf{A}_{y,:}$, and **(2)** does not exceed $\mathbf{nnz}(\mathbf{A}_{x,:})$.

Minimizing the number of deltas: To address point **(1)**, the CBM format must first measure the number of deltas required to convert each row $\mathbf{A}_{y,:}$ into all other rows $\mathbf{A}_{x,:}$ of $\mathbf{A}$, i.e., measure the Hamming distance for each pair of matrix rows. This step provides a global view of the dissimilarity between the rows of the matrix $\mathbf{A}$, and it can be modeled as a fully-connected and undirected distance graph G. This graph has n nodes, where each node represents a unique row of the matrix, and the weight of each edge (y,x) corresponds to the number of deltas required to represent $\mathbf{A}_{x,:}$ with respect to $\mathbf{A}_{y,:}$. To reduce the number of deltas required to compress

the rows of $\mathbf{A}$ we can find a Minimal Spanning Tree (MST) of G, which by definition spans G with the minimum sum of edge weights possible. Naturally, any MST of the distance graph, rooted in node x, defines a compression tree with as many deltas as the weight of this tree plus the number of non-zero elements of $\mathbf{A}_{x,:}$. Therefore, any MST rooted in the node corresponding to the row with the fewest non-zero elements, defines a compression tree that satisfies point **(1)**.

Worst-case guarantees: Note that the compression tree obtained by finding an MST of G does not satisfy point **(2)**, as the weight of the lightest incoming edge of x possibly exceeds $\mathbf{nnz}(\mathbf{A}_{x,:})$. In such cases, representing $\mathbf{A}_{x,:}$ with an adjacency list is clearly more memory efficient. To avoid this issue, we extended the distance graph G with a virtual node $\mathbf{0}$ which is connected to all other nodes of the graph, as illustrated in Figure 1b. This virtual node corresponds to a null-vector $\vec{\emptyset} \in \{0\}^n$, which ensures that the weight of each edge connecting nodes $\mathbf{0}$ and x is equal to the number of non-zero elements in $\mathbf{A}_{x,:}$. The inclusion of this virtual node in the distance graph G ensures that the issue described above cannot occur, since the lightest incoming edge of any node x is now at most as heavy as $\mathbf{nnz}(\mathbf{A}_{x,:})$. Therefore, any compression tree characterized by an MST of G, rooted on node $\mathbf{0}$, is guaranteed to satisfy points **(1)** and **(2)**. These points ensure that CBM format is competitive with standard sparse formats from a theoretical standpoint, as the following property holds:

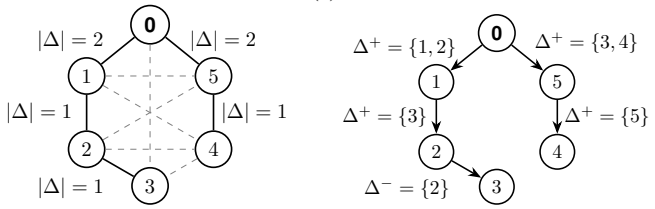
Property 1. *The number of deltas required to represent a binary matrix $\mathbf{A} \in \{0,1\}^{n\times n}$ in Compressed Binary Matrix (CBM) format does not exceed* $\mathbf{nnz}(\mathbf{A})$.

To finalize the construction of the CBM format for matrix $\mathbf{A}$, as shown in Figure 1c, we just have to traverse our compression tree in topological order, while computing the sets of positive and negative deltas required to convert $\mathbf{A}_{y,:}$ into $\mathbf{A}_{x,:}$ for each edge visited.

The CBM format can be easily extended to represent matrices that can be factorized into $\mathbf{AD}$, where $\mathbf{D} \in \mathbb{R}^{n\times n}$ is a diagonal matrix. Note that $\mathbf{AD}$ is a column-scaled matrix, meaning that all non-zero elements of the j-th column of $\mathbf{A}$ will be scaled by entry $\mathbf{D}_{j,j}$. Therefore, to represent $\mathbf{AD}$ in CBM format we just need to store the diagonal of $\mathbf{D}$ alongside with the compression tree of $\mathbf{A}$ and the corresponding sets of positive and negative deltas.

$$\mathbf{A} = \begin{array}{r} 1: \\ 2: \\ 3: \\ 4: \\ 5: \end{array} \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(a)



(b) (c)

Fig. 1: **Construction of the CBM format for (a) matrix A: (b)** shows the Minimum Spanning Tree (MST) of the extended distance graph G for matrix $\mathbf{A}$, where $\mathbf{0}$ denotes the virtual node and each node is identified by its corresponding row in $\mathbf{A}$; **(c)** presents the resulting compression tree along with the associated sets of deltas.

A. Time and Space Analysis

Lemma 1. *Any binary matrix $\mathbf{A} \in \{0,1\}^{n\times n}$ can be represented with Compressed Binary Matrix (CBM) format in $\mathcal{O}(n \cdot \mathbf{nnz}(\mathbf{A}) + n^2 \cdot \log n)$ time.*

Proof. To construct the extended distance graph G of matrix $\mathbf{A}$, we need to compute $n(n+1)/2$ Hamming distances between all row pairs $(\mathbf{A}_{y,:}, \mathbf{A}_{x,:})$. Each distance can be determined by the set intersection of $\mathbf{A}_{y,:}$ and $\mathbf{A}_{x,:}$, which requires $O(\mathbf{nnz}(\mathbf{A}_{x,:})+\mathbf{nnz}(\mathbf{A}_{y,:}))$ operations. Consequently, the time required to measure all Hamming distances is bounded by:

$$\sum_{x=0}^{n}\sum_{y=0}^{n}\left(\mathbf{nnz}(\mathbf{A}_{x,:}) + \mathbf{nnz}(\mathbf{A}_{y,:})\right) = (n+1)\ \mathbf{nnz}(\mathbf{A}).$$

Finding a MST of G using algorithms like Prim or Kruskal requires $\mathcal{O}(E \log V)$ operations, where E is the number of edges and V is the number of nodes. For G, this is $\mathcal{O}(n^2 \cdot \log n)$. Thus, the time required to represent matrix $\mathbf{A}$ in CBM format is bounded by:

$$\mathcal{O}(n \cdot \mathbf{nnz}(\mathbf{A}) + n^2 \cdot \log n). \tag{3}$$

Note that the sequence of intersections between $\mathbf{A}_{x,:}$ and $\mathbf{A}_{y,:}$ required to compute the sets of positive and negative deltas can be computed alongside with the Hamming distance of each node pair. Therefore, the construction of the $2n$ sets of deltas is already accounted for in Equation 3. □

Lemma 2. *Representing matrix $\mathbf{A} \in \{0,1\}^{n\times n}$ in Compressed Binary Matrix (CBM) format requires at most $\mathcal{O}(n + \sum_{x=0}^{n}(|\Delta^{+}_{x,r_x}| + |\Delta^{-}_{x,r_x}|))$ space, where r_x is the index of the row selected to compress row $\mathbf{A}_{x,:}$.*

Proof. Assuming $\mathbf{A}$ is represented in CBM format, then this matrix is composed by a compression tree and corresponding sets of deltas. The compression tree is obtained by computing the MST of a fully-connected graph with n nodes. This implies that the size of the compression tree is proportional to the number of edges, and thus bounded by $\mathcal{O}(n)$. Similarly, the space occupied by all sets of deltas is also proportional to the combined number of elements across all lists, which corresponds to $\mathcal{O}(\sum_{x=0}^{n}(|\Delta^{+}_{x,r_x}| + |\Delta^{-}_{x,r_x}|))$. □

IV. Fast Matrix Products with CBM Format

Let $\vec{v} \in \mathbb{R}^n$ be a real-valued vector, and $\vec{\Delta}^+_{x,y}$ and $\vec{\Delta}^-_{x,y}$ represent the indicator vectors[1] for the sets of positive and negative deltas, respectively. Based on equation Equation 2, we can use the dot-product $\mathbf{A}_{y,:} \cdot \vec{v}$ to compute $\mathbf{A}_{x,:} \cdot \vec{v}$ as:

$$\mathbf{A}_{x,:} \cdot \vec{v} = \mathbf{A}_{y,:} \cdot \vec{v} + (\vec{\Delta}^+_{x,y} - \vec{\Delta}^-_{x,y}) \cdot \vec{v}. \tag{4}$$

Equation 4 shows that dot-product $\mathbf{A}_{x,:} \cdot \vec{v}$ can be computed in $2 \cdot (|\Delta^+_{x,y}| + |\Delta^-_{x,y}|)$ scalar operations once $\mathbf{A}_{y,:} \cdot \vec{v}$ is known. Again, if $\mathbf{A}_{x,:}$ and $\mathbf{A}_{y,:}$ are similar, this value is likely to be smaller than $2 \cdot \mathbf{nnz}(\mathbf{A}_{x,:}) - 1$, the number of scalar operations required to compute the same dot product when $\mathbf{A}_{x,:}$ is represented in a standard sparse format. This intuition suggest that the product between a binary matrix $\mathbf{A}$ and a dense real-value vector $\vec{v}$ can be more efficiently computed than by resorting to sparse formats. To do so, our kernel must compute the dot-product between each row vector of $\mathbf{A}$ and $\vec{v}$ in an order where: **(1)** each dot-product $\mathbf{A}_{x,:} \cdot \vec{v}$ is calculated with respect to the dot-product $\mathbf{A}_{y,:} \cdot \vec{v}$ that results in the minimum overall number of scalar operations, and **(2)** the result of $\mathbf{A}_{y,:} \cdot \vec{v}$ must be known before computing $\mathbf{A}_{x,:} \cdot \vec{v}$.

By definition, the compression tree of the CBM format already imposes an order that satisfies both points when $\mathbf{A}_{x,:}$ is compressed with respect to $\mathbf{A}_{r_x,:}$. The design of fast matrix-vector multiplication kernels with CBM becomes possible by traversing the compression tree of $\mathbf{A}$ in topological order, and for each edge (r_x, x) visited during the traversal compute $u_x \leftarrow \mathbf{A}_{x,:} \cdot \vec{v}$ as

$$u_x \leftarrow u_{r_x} + (\vec{\Delta}^+_{x,r_x} - \vec{\Delta}^-_{x,r_x}) \cdot \vec{v}, \tag{5}$$

where $\vec{u} \in \mathbb{R}^n$ is the result vector and entry $u_{r_x} = \mathbf{A}_{r_x,:} \cdot \vec{v}$.

As observed in Section III, the number of deltas required to represent any row vector $\mathbf{A}_{x,:}$ with CBM format is known to not exceed $\mathbf{nnz}(\mathbf{A}_{x,:})$. If the number of deltas needed to represent $\mathbf{A}_{x,:}$ is strictly smaller than $\mathbf{nnz}(\mathbf{A}_{x,:})$, then the cost of computing the dot-product $\mathbf{A}_{x,:} \cdot \vec{v}$ cannot be greater than $\mathbf{nnz}(\mathbf{A}_{x,:})$ scalar operations. However, when the number of deltas required to represent $\mathbf{A}_{x,:}$ equals $\mathbf{nnz}(\mathbf{A}_{x,:})$, our dot-product exceeds the cost of standard sparse formats by 1. To avoid the second scenario, the CBM format can be engineered to ignore this type of compression opportunities, and instead compress $\mathbf{A}_{x,:}$ with respect to the virtual node $\mathbf{0}$ which is equivalent to represent $\mathbf{A}_{x,:}$ with its adjacency list. Since the number of scalar operations needed to compute $\mathbf{A}_{x,:} \cdot \vec{v}$ with CBM format is always smaller than, or equal to, the number of non-zero elements in $\mathbf{A}_{x,:}$ for $x = 1, \ldots, n$, the following property becomes evident:

Property 2. *The number of scalar operations required to compute matrix-vector products based on the Compressed Binary Matrix (CBM) format is never greater than those required to compute matrix-vector products based on classic sparse formats.*

[1]Vector $\vec{u} \in \{0,1\}^n$ is an indicator of set S if, and only if, $u_k = 1$ when $k \in S$, and $u_k = 0$ when otherwise.

Algorithm 1: Matrix-Matrix product using CBM

Input: $\mathbf{A} \in \{0,1\}^{n \times n}$, $\mathbf{B} \in \mathbb{R}^{n \times p}$, $\mathbf{C} \in \mathbb{R}^{n \times p}$, $\vec{d} \in \mathbb{R}^n$

for $(r_x, x) \in \mathbf{TopologicalOrder}(\mathbf{A})$ **do**
  for $k = 1, \ldots, p$ **do**
    if $r_x = \mathbf{0}$ **then**
      for $j \in \Delta^+_{r_x,x}$ **do**
        $\mathbf{C}_{x,k} \leftarrow \mathbf{C}_{x,k} + (\mathbf{d}_x \cdot \mathbf{d}_j \cdot \mathbf{B}_{j,k})$
    else
      for $j \in \Delta^+_{r_x,x}$ **do**
        $\mathbf{C}_{x,k} \leftarrow \mathbf{d}_x \left(\frac{\mathbf{C}_{r_x,k}}{\mathbf{d}_{r_x}} + \mathbf{d}_j \cdot \mathbf{B}_{j,k} \right)$
      for $j \in \Delta^-_{r_x,x}$ **do**
        $\mathbf{C}_{x,k} \leftarrow \mathbf{d}_x \left(\frac{\mathbf{C}_{r_x,k}}{\mathbf{d}_{r_x}} - \mathbf{d}_j \cdot \mathbf{B}_{j,k} \right)$
return $\mathbf{C}$

Additionally, note that the matrix-vector product using the CBM format does not require the allocation of additional memory. In Equation 5, the result of each dot-product $\mathbf{A}_{x,:} \cdot \vec{v}$, and consequently $\mathbf{A}_{r_x,:} \cdot \vec{v}$, is only stored in $\vec{u}$. Thus, the following property is observed:

Property 3. *The amount of memory required to compute matrix-vector products based on the Compressed Binary Matrix (CBM) format is proportional to the size of its operands and remains constant during execution of this algorithm.*

Finally, note that matrix-vector product with CBM can be easily extended to $\vec{u} \leftarrow \mathbf{AD} \cdot \vec{v}$, where $\mathbf{D} \in \mathbb{R}^n$ is a diagonal matrix. The key observation is that the j-th column of $\mathbf{AD}$ is obtained by scaling the j-th column $\mathbf{A}$ by the diagonal entry $\mathbf{D}_{j,j}$. If the diagonal entries of $\mathbf{D}$ are stored in a vector $\vec{d}$, where $d_j = \mathbf{D}_{j,j}$ for $j = 1, \ldots, n$, the dot-product between the x-th row vector of $\mathbf{AD}$ and $\vec{v}$ is given by:

$$u_x \leftarrow u_{r_x} + (\vec{d} \odot (\vec{\Delta}^+_{x,r_x} - \vec{\Delta}^-_{x,r_x})) \cdot \vec{v},$$

where $\odot$ represents the element-wise multiplication. Similarly, matrix-vector product with our format can be extended to compute $\vec{u} \leftarrow \mathbf{DAD} \cdot \vec{v}$, which will be crucial to accelerate the inference stage of different GCNs. Here, the key observation is that the x-th row of vector $\mathbf{DAD} \cdot \vec{v}$ is obtained by scaling the x-th row of $\mathbf{AD} \cdot \vec{v}$ by $\mathbf{D}_{x,x}$ or d_x for $x = 1, \ldots, n$. Thus, the dot-product between the x-th row vector of $\mathbf{DAD}$ and $\vec{v}$ can be written as:

$$u_x \leftarrow d_x \left(\frac{u_{r_x}}{d_{r_x}} + (\vec{d} \odot (\vec{\Delta}^+_{x,r_x} - \vec{\Delta}^-_{x,r_x})) \cdot \vec{v} \right). \tag{6}$$

Note that the entry u_{r_x} must first be divided by d_{r_x}, since it was previously scaled by d_{r_x} during the computation of the r_x-th row of $\mathbf{DAD} \cdot \vec{v}$.

A. Matrix-Matrix Products with CBM

Building on the different forms of matrix-vector products described above, we designed fast matrix-matrix multiplication kernels based on the CBM format that also meet Properties 2 and 3. Our matrix-matrix multiplication kernel is implemented by performing matrix-vector products between a matrix in CBM format and each column of the right-hand-side operand matrix. Algorithm 1 describes our approach for computing $\mathbf{C} \leftarrow \mathbf{DADB}$ with CBM format, where both $\mathbf{B} \in \mathbb{R}^n$ and $\mathbf{C} \in \mathbb{R}^n$ are dense, real-valued matrices. To adapt this strategy for products of the form $\mathbf{C} \leftarrow \mathbf{ADB}$, simply omit the operations involving d_x and d_{r_x}. For matrix products of the form $\mathbf{C} \leftarrow \mathbf{AB}$ omit operations involving d_j in addition to d_x and d_{r_x}.

V. Implementation Details and Optimizations

A. Leveraging High-Performance Numerical Libraries

Up to this point, the CBM format was conceptualized as a compression tree, where each edge (r_x, x) is associated with two sets of deltas (Δ^+_{x,r_x} and Δ^-_{x,r_x}). As is, matrix multiplication with the CBM format presents sub-optimal performance, because this operation is structured as a sequence of dot-products between sparse indicator vectors ($\vec{\Delta}^+_{x,r_x}$ and $\vec{\Delta}^-_{x,r_x}$) and the column vectors of the right-hand side operand matrix. To accelerate our multiplication kernels, we need to cast this sequence of dot-product as a single sparse matrix dense matrix product, for which efficient and high-performance implementations already exist.

Let $\mathbf{A}' \in \{-1, 0, 1\}^{n \times n}$ be the matrix of deltas of $\mathbf{A}$:

$$\mathbf{A}' = \begin{bmatrix} \vec{\Delta}^+_{1,r_1} - \vec{\Delta}^-_{1,r_1} \\ \vdots \\ \vec{\Delta}^+_{n,r_n} - \vec{\Delta}^-_{n,r_n} \end{bmatrix}.$$

Assume we would like to multiply matrices $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{B}_{:,j}$ is the j-th column vector of $\mathbf{B}$ for $j = 1, \cdots, n$. By definition, all dot-products $(\vec{\Delta}^+_{x,r_x} - \vec{\Delta}^-_{x,r_x}) \cdot \mathbf{B}_{:,j}$ required to calculate $\mathbf{AB}$ using the CBM format can be now obtained in a single matrix multiplication:

$$\mathbf{A'B} = \begin{bmatrix} (\vec{\Delta}^+_{1,r_1} - \vec{\Delta}^-_{1,r_1}) \cdot \mathbf{B}_{1,:} & \cdots & (\vec{\Delta}^+_{1,r_1} - \vec{\Delta}^-_{1,r_1}) \cdot \mathbf{B}_{n,:} \\ \vdots & \ddots & \vdots \\ (\vec{\Delta}^+_{n,r_n} - \vec{\Delta}^-_{n,r_n}) \cdot \mathbf{B}_{1,:} & \cdots & (\vec{\Delta}^+_{n,r_n} - \vec{\Delta}^-_{n,r_n}) \cdot \mathbf{B}_{n,:} \end{bmatrix}.$$

To obtain matrix $\mathbf{AB}$ from $\mathbf{A'B}$ we must traverse the compression tree in topological order, and for each edge (r_x, x) visited compute:

$$(\mathbf{A'B})_{x,:} \leftarrow (\mathbf{A'B})_{r_x,:} + (\mathbf{A'B})_{x,:}.$$

Once all edges of the compression tree are updated the value of $\mathbf{A'B}$ is guaranteed to be equal to $\mathbf{AB}$. Given that Property 1 ensures that the matrix of deltas $\mathbf{A}'$ is at least as sparse as $\mathbf{A}$, we opted to represent $\mathbf{A}'$ in CSR format to exploit its sparsity, and we leverage Intel MKL's sparse-dense matrix multiplication kernels to accelerate the product $\mathbf{A'B}$. It is also important to highlight that the update stage of matrix multiplication using the CBM format is a sequence of vector additions. Hence, we accelerated this stage by using the `axpy` implementation also offered by Intel MKL. Finally, note that the sets of the deltas Δ^+_{x,r_x} and Δ^-_{x,r_x} previously associated with each edge of the compression tree are no longer needed. The information of these sets is now embedded in $\mathbf{A}'$.

Extending our implementation of the CBM format to matrices of the form $\mathbf{AD}$ and $\mathbf{DAD}$, where $\mathbf{D}$ is a diagonal matrix, was straightforward. To represent $\mathbf{AD}$, and multiply it with matrix $\mathbf{B}$, using our format, we only need to define an appropriate matrix of deltas:

$$(\mathbf{AD})' = \begin{bmatrix} \vec{d} \odot (\vec{\Delta}^+_{1,r_1} - \vec{\Delta}^-_{1,r_1}) \\ \vdots \\ \vec{d} \odot (\vec{\Delta}^+_{n,r_n} - \vec{\Delta}^-_{n,r_n}) \end{bmatrix},$$

where vector $\vec{d}$ contains the diagonal entries of the matrix $\mathbf{D}$. Once the matrix of deltas $(\mathbf{AD})'$ is initialized, we multiply it with $\mathbf{B}$ and update the resulting matrix as described before, to accelerate the product $\mathbf{ADB}$. Note that when multiplying matrices of the form $\mathbf{AD}$ using the CBM format, the diagonal matrix $\mathbf{D}$, or its vector representation, does not need to be stored in memory, as the necessary information is also embedded in the matrix of deltas $(\mathbf{AD})'$. Furthermore, both products $(\mathbf{A})'\mathbf{B}$ and $(\mathbf{AD})'\mathbf{B}$ present the same sparsity pattern and require the same number of scalar operations, suggesting that the performance of both operations should be similar.

To compute the product of a matrix of the form $\mathbf{DAD}$ with $\mathbf{B}$ using the CBM format, in addition to constructing $(\mathbf{AD})'$ and multiplying it by $\mathbf{B}$, we must also modify the update stage to scale the rows of the resulting matrix by $\vec{d}$. For this type of matrix, the update stage must traverse the compression tree, and for each edge (r_x, x), update the x-th row of the resulting matrix according to Equation 6:

$$((\mathbf{AD})'\mathbf{B})_{x,:} \leftarrow d_x \left(\frac{((\mathbf{AD})'\mathbf{B})_{r_x,:}}{d_{r_x}} + ((\mathbf{AD})'\mathbf{B})_{x,:} \right).$$

The update stage for $\mathbf{DADX}$ introduces two additional floating-point operations per row element. To minimize this overhead, we fused row scaling with row update, ensuring that the rows of the matrix $(\mathbf{AD})'\mathbf{B}$ are not loaded more times than necessary compared to the update stage of the CBM format for $\mathbf{A'B}$ and $(\mathbf{AD})'\mathbf{B}$. This type of matrix multiplication also introduces some memory overhead, as the vector $\vec{d}$ must remain in memory during the update stage. For simplicity, we assumed that the left and right diagonal matrices are the same. Nevertheless, the CBM format can be easily extended to support matrices of the form $\mathbf{D}_1\mathbf{AD}_2$, where $\mathbf{D}_1$ and $\mathbf{D}_2$ are distinct real-valued diagonal matrices.

B. SIMD and Multi-Threading

In order to exploit the full performance potential of modern processors and to ensure that our matrix multiplication strategy is competitive with the state-of-the-art in sparse-dense matrix multiplication, our solution must leverage both SIMD and multi-threading parallelism. Intel MKL already leverages both

types of parallelism to accelerate its sparse-dense matrix multiplication and `axpy` kernels. While, multiplying either $\mathbf{A}'$, or $(\mathbf{AD})'$, with the right-hand side matrix $\mathbf{B}$ is already efficiently parallelized by Intel MKL alone, the same is not observed for the update stage of the CBM format. The update stage of our format requires the execution of many `axpy` operations to convert the matrix obtained in the multiplication stage into $\mathbf{AB}$, $\mathbf{ADB}$, or $\mathbf{DADB}$. Applying multi-threading within the `axpy` kernel is too fine-grained, leading to sub-optimal performance. To address this issue, we assign a set of `axpy` operations to each thread available. However, it is important to note that the update stage presents data-dependencies. The compression tree needs to be traversed in topological order, which means that a thread cannot begin the update for row x until the update for row r_x has been completed (possibly by another thread). The key observation that allows us to efficiently parallelize the update stage is that there are no data-dependencies across the branches of the compression tree. Therefore, to ensure the correctness and efficiency of our parallel update stage, we assign to each thread one, or more, lists containing all the edges that form a complete branch of the compression tree. To avoid unnecessary overhead during the traversal of each branch, we sort each list of edges in topological order. At this point, it becomes straightforward to parallelize the update stage with OpenMP. To do so, we use the `parallel for` directive to create a parallel loop to iterate over the lists of edges of each branch. Additionally, we include the `schedule(dynamic)` clause to address load-balancing issues caused by branches of different sizes. Note that we could also resort to task-level parallelism to accelerate the update stage of the CBM format. However, this type of implementation would be more involved and incur synchronization overheads.

C. Improving CBM with Edge Pruning

An important observation is that not all compression opportunities lead to space saving or faster multiplication kernels when using the CBM format. If the number of deltas needed to compress $\mathbf{A}_{x,:}$ in our format is close to $\mathbf{nnz}(\mathbf{A}_{x,:})$, potential gains with respect to memory or performance may be outweighed by the costs of representing and traversing the compression tree. This observation becomes clear if we consider the following example:

Example 1. *Let $\mathbf{A}_{x,:}$ be compressed with respect to $\mathbf{A}_{r_x,:}$ when $\mathbf{A}$ is represented in CBM format. Then, the number of scalar operations saved by compressing $\mathbf{A}_{x,:}$ equals*

$$|\Delta^+_{x,r_x}| + |\Delta^-_{x,r_x}| - \mathbf{nnz}(\mathbf{A}_{x,:}).$$

Now, assume that the number of deltas required to represent $\mathbf{A}_{x,:}$ is $\mathbf{nnz}(\mathbf{A}_{x,:}) - 1$. In this case, the corresponding matrix of deltas (either $\mathbf{A}'$ or $(\mathbf{AD})'$) would contain one less non-zero element. However, at least two integers would have to be added to the compression tree to represent edge (r_x, x). Consequently, had this edge not been considered, the CBM format would occupy less memory.

To avoid the scenario above we introduce a user-defined threshold α, where $\alpha \in \mathbb{N}_0$. This parameter is applied in the construction of the CBM format, during the generation of the distance graph G. Before any edge (y, x) is included in G, we check whether $|\Delta^+_{x,y}| + |\Delta^-_{x,y}| - \mathbf{nnz}(\mathbf{A}_{x,:}) < \alpha$. If this condition evaluates to true, this edge is included in G and considered during the construction of the MST. Otherwise, it is immediately discarded. Naturally, this condition might evaluate to true for only one edge direction, meaning that G is now directed. Therefore, a suitable compression tree must now be found with a Minimum Cost Arborescence (MCA) rooted in the virtual node $\mathbf{0}$. Note that our compression algorithm remains correct, since G contains an out-going edge from node $\mathbf{0}$ to all other nodes. For convenience, the MCA algorithm employed to build the CBM format runs in $\mathcal{O}(n^2 \cdot \log^2 n)$. However, there are more efficient algorithms that find an MCA of G in $\mathcal{O}(n^2 \cdot \log n)$ [12]. Consequently, the time complexity of our compression algorithm remains the same, whether it uses an MCA or an MST.

It is important to note that α serves an additional purpose. As the value of α increases, so does the branching factor of the virtual node $\mathbf{0}$ in the compression tree. Thus, by increasing α we also increase the degree of parallelism during the update stage of our matrix multiplication strategy. This comes, however, at the expense of the quality of compression of the CBM format. Additionally, as the value of α increases, the MCA algorithm considers a smaller amount of candidate edges, and therefore, a slight reduction in compression time should be expected.

VI. Experimental Evaluation

A. Experimental Setting

The experiments found in this section were performed on an Intel Xeon Gold 6130 (Skylake) CPU, a shared-memory architecture with 16 physical cores, 2.1 GHz fixed clock frequency, 32 KiB private L1 data cache, 1 MiB private L2 cache and 22 MiB shared L3 cache. This machine runs on CentOS Linux 7 (version 3.10.0) operating system. The compression algorithm of the CBM format and corresponding matrix multiplication kernels were implemented in C++, and rely on Intel MKL (version 24.0.0) sparse CSR format and corresponding single-precision (32 bits) sparse-dense matrix multiplication kernels. The C++ code developed in this work was compiled with GCC (version 10.5.0), and is called from Python 3.11, via PyTorch C++ Extensions to closely resemble common use-cases. Parallel experiments (with 16 cores) were implemented with OpenMP 4.5, and the threads were pinned to physical cores with environment variable `GOMP_CPU_AFFINITY="0-15"`.

B. Evaluation Metrics

We evaluated the CBM format on various graphs using two key metrics: compression ratio and speedup. Since the graphs considered in our experiments are very sparse, we chose the CSR format and the corresponding Intel MKL's sparse matrix multiplication kernels as our baselines. We did not

consider native PyTorch sparse matrix multiplication kernels as a baseline, as preliminary experiments revealed a significant slowdown compared to explicitly invoking the routines offered by Intel MKL using PyTorch's C++ Extensions.

Compression Ratio: is expressed as $\mathbf{S_{CSR}}/\mathbf{S_{CBM}}$, where $\mathbf{S_{CSR}}$ and $\mathbf{S_{CBM}}$ correspond to the memory occupied by the graph in MiB, when the graph is represented in CSR or CBM format, respectively.

Speedup: is represented as the ratio $\mathbf{T_{CSR}}/\mathbf{T_{CBM}}$, where $\mathbf{T_{CSR}}$ and $\mathbf{T_{CBM}}$ represent the average time of execution of the same operation (averaged over 250 runs) with CSR and CBM format, respectively. The operations considered in these experiments are the different flavors of matrix multiplication kernels proposed in this work, and the inference stage of a two-layer GCN. The time needed to represent each graph in either format was not included in the measurements of $\mathbf{T_{CSR}}$ and $\mathbf{T_{CBM}}$. The same way graphs are already offered in CSR, or in other sparse formats, we assume that these graphs could also be offered in CBM. In the spirit of transparency, Table II shows the time needed to build our format for each graph.

We verified the correctness of our matrix multiplication kernels by multiplying each graph's adjacency matrix, in CBM format, by 50 randomly generated matrices with 500 columns, where each element is between 0 and 1. We confirmed that the resulting matrices matched those of the baseline, across all graphs tested, within a relative tolerance of 10^{-5}, which we consider satisfactory.

C. Datasets

To evaluate the advantages of the CBM format with respect to compression ratio and speedup, we selected eight real-world graphs of varying sizes and average degrees, as depicted in Table I. The first seven graphs depict relationships between authors and/or academic papers, where nodes tend to share many neighbors in common. Cora and Pubmed [13] are both citation networks, where each node represents an academic paper, and an unweighted and undirected edge is placed between citing and cited papers. ca-AstroPh and ca-HepPh [14] represent co-authoring networks, where each node corresponds to an author, and there is an unweighted and undirected edge between two authors if they co-authored a paper together. COLLAB [15] is collaboration network composed by the ego-networks of several researchers. In this network, each node represents a researcher, and there is an unweighted and undirected edge between two researchers if they collaborated. coPapersDBLP and coPapersCiteseer [16] are co-papers networks where each nodes represents a paper, and two papers are connected by an unweighted and undirected edge if they share at least one author. The largest dataset that we considered was the ogbn-proteins. This graph represents a protein-protein interaction (PPI) network, where nodes represent proteins and edges indicate biologically meaningful associations between proteins. This dataset offered in Open Graph Benchmark (OGB) [17] as an undirected graph with edge features. However, to be able to represent this graph in our format we ignored the edge weights, making the graph undirected and unweighted.

TABLE I: **Networks selected to evaluate the CBM format.**

Graph	#Nodes	#Edges	Average Degree	S_{CSR} [MiB]
Cora	2708	10556	4.8	0.09
PubMed	19717	88648	5.4	0.75
ca-AstroPh	18772	396160	22.1	3.09
ca-HepPh	12008	237010	20.7	1.85
COLLAB	372474	24572158	65.9	188.89
coPapersDBLP	540486	30491458	57.4	234.69
coPapersCiteseer	434102	32073440	74.8	246.36
ogbn-proteins	132534	39561252	298.5	302.33

TABLE II: **Compression analysis of the CBM format for different datasets with $\alpha = 0$ and $\alpha = 32$. Time** represents the time needed to build the CBM format using 16 threads.

Graph	Alpha	Time [s]	S_{CSR} [MiB]	S_{CBM} [MiB]	$\frac{S_{CSR}}{S_{CBM}}$
Cora	$\alpha = 0$	0.0035 (± 0.000)	0.09	0.09	1.04
	$\alpha = 32$	0.0024 (± 0.000)		0.09	1.00
PubMed	$\alpha = 0$	0.0359 (± 0.001)	0.75	0.72	1.04
	$\alpha = 32$	0.0145 (± 0.000)		0.74	1.00
ca-AstroPh	$\alpha = 0$	0.1105 (± 0.003)	3.09	1.80	1.72
	$\alpha = 32$	0.0728 (± 0.004)		2.44	1.27
ca-HepPh	$\alpha = 0$	0.0813 (± 0.005)	1.85	0.68	2.72
	$\alpha = 32$	0.0539 (± 0.003)		0.89	2.06
COLLAB	$\alpha = 0$	5.2422 (± 0.090)	188.89	17.18	11.0
	$\alpha = 32$	4.3964 (± 0.026)		32.49	5.81
coPapersDBLP	$\alpha = 0$	7.7507 (± 0.056)	234.69	39.34	5.97
	$\alpha = 32$	6.1038 (± 0.025)		62.76	3.74
coPapersCiteseer	$\alpha = 0$	8.7924 (± 0.126)	246.36	24.95	9.87
	$\alpha = 32$	7.1432 (± 0.115)		42.56	5.79
ogbn-proteins	$\alpha = 0$	44.3282 (± 0.0797)	246.36	24.95	2.14
	$\alpha = 32$	43.3534 (± 0.0507)		42.56	2.12

D. Graph Compression with the CBM Format

Table II presents the average time to convert each dataset into CBM format and corresponding compression ratio for the smallest and largest α values considered in our experiments. For $\alpha = 0$, our compression algorithm processes the graphs in under 9 seconds, except for ogbn-proteins, which takes about 44 seconds. With $\alpha = 32$, the compression time decreases for every graph, confirming that our compression algorithm be-

comes faster as α increases. In most cases, converting a graph to CBM format will take longer than matrix multiplication with the CSR format, especially when the second operand matrix has few columns. To achieve meaningful speedup in matrix multiplication or GCN inference using our format, the graph must first be made available in CBM format as a pre-processing step. As it can be seen in Table II, the CBM format reduced the memory footprint of every dataset compared to CSR, though the extent of the improvement varied across datasets. Representing both citation graphs with our format leads to negligible compression gains with respect to CSR. We suspect that the low average degree of the citation graphs limits the compression ratio of our format. For these datasets, the compression gains in $\mathbf{A}'$ are likely to be too small to offset the memory overhead of the compression tree. For both co-authoring and PPI networks, the CBM format resulted in substantial compression gains with an average compression ratio of 2.19× for $\alpha = 0$. The COLLAB and the co-papers networks benefited the most from our format. For these graphs our format achieves an impressive average compression ratio of 8.95× for $\alpha = 0$.

E. Matrix Multiplication Kernels using the CBM Format

In this section, the experiments focus on the performance of matrix multiplication kernels for $\mathbf{AX}$, $\mathbf{ADX}$, and $\mathbf{DADX}$. In these operations, $\mathbf{A}$ represents the adjacency matrix of each graph, $\mathbf{X}$ is a dense single-precision matrix with 500 columns, and $\mathbf{D}$ is a single-precision diagonal matrix. For matrix multiplication using the CBM format, the variants of the right-hand side operand matrices ($\mathbf{A}$, $\mathbf{AD}$, and $\mathbf{DAD}$) are represented in our format as described in Section V-A. For the baseline, we assume that the right-hand operand matrices are represented by a single CSR matrix. As previously stated, the time required to represent these matrices in either CBM or CSR formats is not accounted for in the following experiments.

As discussed in Section V-C, finding α is key to improve the performance of matrix multiplication with the CBM format. Adjusting this parameter not only reduces overhead associated with traversing the compression tree, but also exposes more parallelism opportunities since it increases the out-degree of the virtual node $\mathbf{0}$. Given the importance of α we first consider the case where $\alpha = 0$ and our edge pruning technique was not applied, and then we show how adjusting this parameter impacts matrix multiplication with the CBM format.

1) $\mathbf{AX}$ *(*$\alpha = 0$*):* Figure 2 confirms that the speedup achieved for matrix multiplication using the CBM format is proportional to the compression ratio obtained by our format. The lack of compression for both citation graphs (Figs. 2a and 2b) naturally results in a slowdown for matrix multiplication, as the compression gains in the matrix of deltas do not offset the cost of traversing the compression tree during matrix multiplication using CBM. While co-authoring (Figs. 2c and 2d) and PPI networks (Fig. 2h) present a significant average compression ratio of 2.19×, matrix multiplication with these graphs resulted in an average speedup of 1.79× in sequential and 1.45× in parallel settings. The speedups obtained for these graphs are lower than we would expect given the compression ratios presented. It is also worth to point out that the co-authoring networks show a significant performance gap between sequential and parallel matrix multiplication. This disparity is not due to a lack of scalability in our solution. Rather, the baseline scales better because the original adjacency matrices do not fit in the L1 and L2 caches of a single core, but they do fit across the combined L1 and L2 caches of 16 cores. As it should be expected, the graphs that benefited the most from matrix multiplication using the CBM format were the COLLAB (Fig. 1e) and the co-papers networks (Fig. 1f and 1g), which present an average compression ratio of 8.95×. These graphs present an impressive average speedup of 3.32× in sequential and 3.95× in parallel settings, highlighting the potential of the CBM format to accelerate matrix multiplications involving the adjacency matrix of real-world unweighted graphs.

2) $\mathbf{AX}$ *(*$\alpha > 0$*):* Figure 2 confirms that adjusting the value of α speeds up matrix multiplication with every graph, though in varying degrees. For the citation graphs (Fig. 1a and 1b), setting $\alpha \geq 2$ reduces the overhead associated with traversing the compression tree, which negates the performance decay observed for sequential matrix multiplication for $\alpha = 0$. A similar pattern is seen for parallel matrix multiplication with Cora (Fig. 1a). It is also interesting to note that for these graphs setting $\alpha = 1$ increases the compression ratio obtained with CBM, indicating that the scenario described in Example 1 occurs for many rows of these adjacency matrices. Setting $\alpha = 4$ results in the lowest execution times for sequential matrix multiplication with both co-authoring networks (Figs. 1c and 1d), slightly increasing the average speedups for these datasets by 0.1. For parallel matrix multiplication the best results are obtained when $\alpha = 8$ for ca-AstroPh and when $\alpha = 2$ for ca-HepPh, increasing the average parallel speedup by 0.09. In contrast to what was observed before, adjusting α has little to no impact in sequential matrix multiplication with the remaining networks (Figs. 2e to 2h). This behavior occurs because most rows in the delta matrix $\mathbf{A}'$ have at least four fewer nonzero entries than the corresponding rows in $\mathbf{A}$. This observation is supported by the compression ratio obtained for these networks, which remain relatively constant up to $\alpha = 8$. Another important observation is that increasing the value of α promotes many parallelism opportunities in the context of parallel matrix multiplication with these graphs. This observation is supported by COLLAB (Fig. 2e) and both co-papers (Figs. 2f and 2g) networks, where the parallel speedup increases, while the compression ratio and sequential speedup decrease sharply. Setting $\alpha = 16$ results in the best parallel speedup for COLLAB, while setting $\alpha = 32$ leads to the best parallel speedups for both co-paper networks. Fine-tuning α for these graphs increases their average parallel speedup by 0.31×. We would like to highlight that parallel matrix multiplication with COLLAB achieved the highest

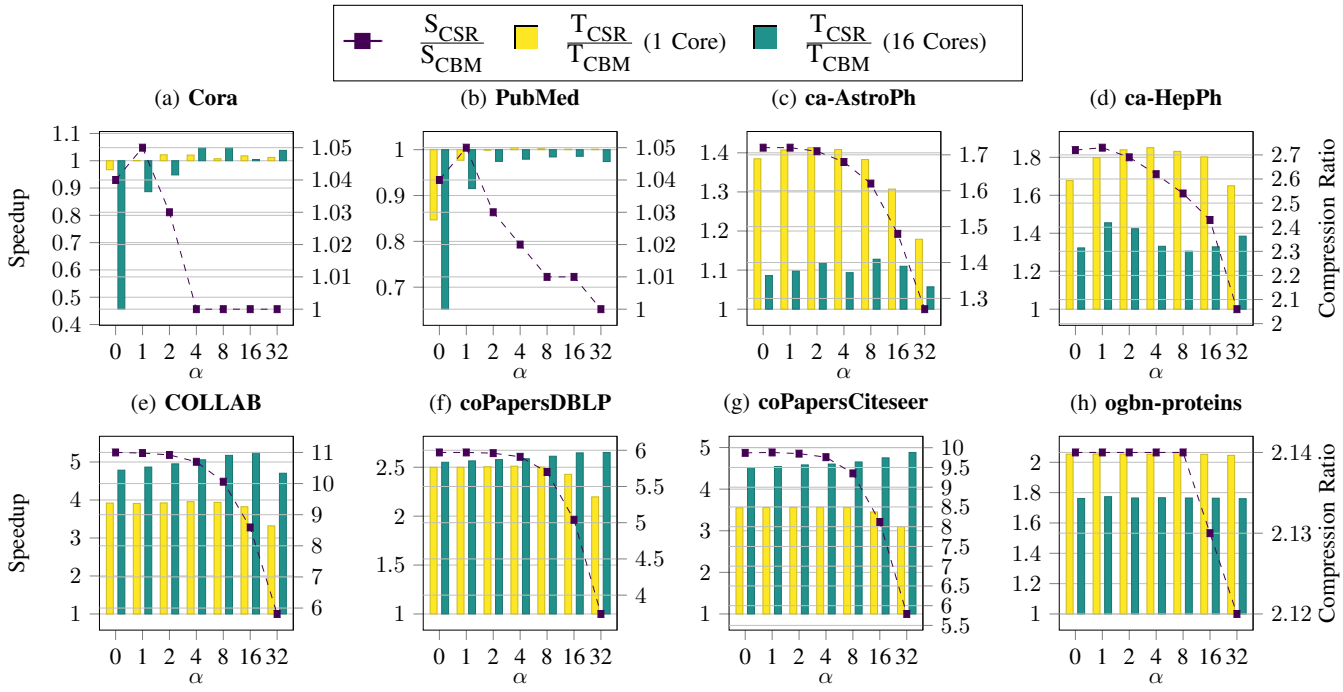


Fig. 2: **Impact of different values of α on matrix-matrix multiplication ($\mathbf{AX}$) using the CBM format.** These plots indicate the speedup achieved by the CBM format, with respect to the CSR format, for sequential and parallel matrix multiplication. Additionally, the plots present the compression ratio achieved by our format also with respect to the CSR format. The x-axis shows the different values of α considered. The y-axis (left), shows speedup relative to Intel MKL's sparse-dense matrix multiplication kernels, where the graph is represented in CSR format; the y-axis (right) shows the compression ratio relative to size of the same graph represented in CSR format. (**Note:** α is a parameter of our format, not the number of cores employed.)

average speedup of 5.25× with respect to the current state-of-the-art.

To conclude this discussion, we note that identifying the optimal values of α is relatively straightforward in the sequential case, as the best α appears to be mostly independent from the graph we would like to compress. This observation is supported by the experiments presented in Figure 2, which indicate that $\alpha = 4$ is the optimal choice for all graphs except ca-AstroPh and ogbn-proteins. In contrast, finding the best values of α in the parallel case is not as simple, since the degree of parallelism associated with each α depends on the compression tree obtained for each graph. Nevertheless, for graphs that are as well compressed as COLLAB and co-papers, any α between 8 and 32 seems to be a reasonable choice.

F. $\mathbf{ADX}$ *and* $\mathbf{DADX}$

To finalize the experimental analysis of matrix multiplication using the CBM format, we evaluate the speedup achieved by our kernels for $\mathbf{ADX}$ and $\mathbf{DADX}$ compared to Intel MKL's sparse-dense matrix multiplication kernels, which use the CSR format. To keep this discussion concise, we only consider the values α that yield the best speedup for $\mathbf{AX}$. The results of these experiments are presented in Table III. As shown, the speedup achieved for matrix multiplication using the CBM format for $\mathbf{ADX}$ and $\mathbf{DADX}$ does not exhibit substantial slowdowns compared to $\mathbf{AX}$ across the selected graphs for the different combinations of α and number of threads. As explained in Section V-A, we did not anticipate a significant difference in speedup between $\mathbf{AX}$ and $\mathbf{ADX}$. However, we were positively surprised to observe no significant differences in performance when moving from matrix multiplication with $\mathbf{AX}$ to $\mathbf{DADX}$.

G. GCN Inference with CBM Matrix Multiplication Kernels

To evaluate the potential impact of the CBM format on the training and inference times of GNNs, we employed our matrix multiplication strategy in the inference stage of the two-layer GCN, previously described in Equation 1:

$$\hat{\mathbf{A}}\ \sigma(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^0)\mathbf{W}^1.$$

In our experiments, we represent the normalized Laplacian adjacency matrix for each graph, $\hat{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}}(\mathbf{A}+\mathbf{I})\mathbf{D}^{-\frac{1}{2}}$, using CBM for matrices of the form $\mathbf{DAD}$, and apply the corresponding matrix multiplication kernels to compute the two products involving $\hat{\mathbf{A}}$ that are present in Equation 1. The node feature matrix, $\mathbf{X}$, is a dense single-precision matrix with 500 columns, while the learnable matrices, $\mathbf{W}^0$ and $\mathbf{W}^1$, are dense, single-precision, square matrices with 500 rows. These matrix dimensions were carefully chosen to reflect real-world GCN workloads. The baseline for our experiments is the same two-layer GCN, where $\hat{\mathbf{A}}$ is stored in CSR format and matrix products involving this matrix are carried-out by

TABLE III: **Performance analysis for different matrix multiplications using CSR and CBM formats.** To keep the analysis concise, we only consider the values of α that yielded the best speedups for $\mathbf{AX}$ when evaluating $\mathbf{ADX}$ and $\mathbf{DADX}$.

Graph	Alpha (Cores)	AX			ADX			DADX		
		T_{CSR} [s]	T_{CBM} [s]	$\frac{T_{CSR}}{T_{CBM}}$	T_{CSR} [s]	T_{CBM} [s]	$\frac{T_{CSR}}{T_{CBM}}$	T_{CSR} [s]	T_{CBM} [s]	$\frac{T_{CSR}}{T_{CBM}}$
ca-HepPh	α = 4 (1 Core)	0.0239 (± 0.0016)	0.0129 (± 0.0002)	1.8507	0.0235 (± 0.0006)	0.0132 (± 0.0008)	1.7882	0.0237 (± 0.0013)	0.0131 (± 0.0006)	1.8045
	α = 1 (16 Cores)	0.0025 (± 0.0001)	0.0017 (± 0.0000)	1.4552	0.0025 (± 0.0001)	0.0017 (± 0.0000)	1.4769	0.0025 (± 0.0002)	0.0018 (± 0.0001)	1.4235
ca-AstroPh	α = 2 (1 Core)	0.0482 (± 0.0004)	0.0341 (± 0.0004)	1.4132	0.0480 (± 0.0005)	0.0341 (± 0.0003)	1.4090	0.0480 (± 0.0004)	0.0345 (± 0.0010)	1.3882
	α = 8 (16 Cores)	0.0045 (± 0.0000)	0.0040 (± 0.0000)	1.1278	0.0045 (± 0.0000)	0.0040 (± 0.0000)	1.1286	0.0045 (± 0.0000)	0.0040 (± 0.0001)	1.1147
Cora	α = 2 (1 Core)	0.0011 (± 0.0000)	0.0011 (± 0.0000)	1.0217	0.0011 (± 0.0000)	0.0011 (± 0.0000)	1.0132	0.0011 (± 0.0000)	0.0011 (± 0.0000)	1.0225
	α = 4 (16 Cores)	0.0002 (± 0.0000)	0.0002 (± 0.0000)	1.0478	0.0002 (± 0.0000)	0.0002 (± 0.0000)	1.0421	0.0002 (± 0.0000)	0.0002 (± 0.0000)	0.9690
PubMed	α = 4 (1 Core)	0.0169 (± 0.0003)	0.0168 (± 0.0002)	1.0044	0.0169 (± 0.0002)	0.0168 (± 0.0002)	1.0043	0.0170 (± 0.0007)	0.0169 (± 0.0002)	1.0059
	α = 16 (16 Cores)	0.0020 (± 0.0000)	0.0020 (± 0.0000)	0.9856	0.0020 (± 0.0000)	0.0020 (± 0.0000)	0.9870	0.0020 (± 0.0000)	0.0020 (± 0.0000)	0.9835
COLLAB	α = 4 (1 Core)	1.2058 (± 0.0380)	0.3048 (± 0.0080)	3.9555	1.2117 (± 0.0801)	0.3061 (± 0.0116)	3.9582	1.2024 (± 0.0293)	0.3055 (± 0.0106)	3.9353
	α = 16 (16 Cores)	0.2466 (± 0.0024)	0.0470 (± 0.0002)	5.2513	0.2462 (± 0.0016)	0.0470 (± 0.0002)	5.2369	0.2465 (± 0.0016)	0.0477 (± 0.0032)	5.1692
coPapersDBLP	α = 4 (1 Core)	2.1310 (± 0.0303)	0.8488 (± 0.0078)	2.5107	2.1334 (± 0.0381)	0.8513 (± 0.0082)	2.5062	2.1360 (± 0.0526)	0.8687 (± 0.0241)	2.4588
	α = 32 (16 Cores)	0.4284 (± 0.0115)	0.1616 (± 0.0004)	2.6513	0.4370 (± 0.0014)	0.1608 (± 0.0005)	2.7177	0.4362 (± 0.0048)	0.1612 (± 0.0023)	2.7058
coPapersCiteseer	α = 4 (1 Core)	1.9633 (± 0.0328)	0.5510 (± 0.0139)	3.5629	1.9628 (± 0.0272)	0.5511 (± 0.0100)	3.5619	1.9632 (± 0.0362)	0.5600 (± 0.0170)	3.5058
	α = 32 (16 Cores)	0.4583 (± 0.0161)	0.0939 (± 0.0006)	4.8817	0.4810 (± 0.0244)	0.0932 (± 0.0002)	5.1632	0.4783 (± 0.0246)	0.0937 (± 0.0020)	5.1046
ogbn-proteins	α = 8 (1 Core)	9.1808 (± 0.0016)	4.4460 (± 0.0079)	2.0650	9.1920 (± 0.0020)	4.4532 (± 0.0074)	2.0642	9.1954 (± 0.0223)	4.5027 (± 0.0496)	2.0422
	α = 16 (16 Cores)	2.1987 (± 0.0011)	1.2400 (± 0.0032)	1.7732	2.1873 (± 0.0021)	1.2349 (± 0.0010)	1.7712	2.1910 (± 0.0011)	1.2381 (± 0.0060)	1.7697

Intel MKL's sparse-dense matrix multiplication kernels. The speedup achieved by the CBM format for the selected graphs can be found in Table IV. Again, to keep the discussion concise, we only consider the α values that yield the best speedup for $\mathbf{AX}$.

In general, we observed a substantial reduction in speedup when comparing the performance of the CBM format in GCN inference to its performance in matrix multiplication with $\mathbf{DAD}$. Since our format cannot optimize the matrix products between $\mathbf{X}$ and $\mathbf{W}^0$, as well as between $\sigma(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}^0)$ and $\mathbf{W}^1$, these additional operations dilute the performance gains that were previously observed. Nevertheless, the CBM format still reduces the inference time of a GCN for all graphs, except for the citation networks. For the last four networks of Table IV, our format resulted in an average speedup of 1.6× in sequential and 1.93× in parallel settings. In these experiments, we observed an interesting result that seems counterintuitive at first: while the CBM format presents the best performance for $\mathbf{DAD}$ with COLLAB, the same is not verified for GCN inference. Instead, the dataset that presents the best speedup in the context of GCN inference is the coPapersCiteseer. The sharper decline in performance from $\mathbf{DAD}$ to GCN inference for COLLAB is primarily due to a memory effect caused by the inclusion of matrices $\mathbf{X}$, $\mathbf{W}^0$, and $\mathbf{W}^1$. These matrices occupy additional cache space, resulting in COLLAB's data being evicted more frequently than when computing $\mathbf{DAD}$, where most of the graph fits across the combined private levels of cache of 16 cores. We hypothesize that these memory effects are less pronounced in CoPapersCiteseer as this graph was already too large to be stored across the combined private levels of cache during the computation of $\mathbf{DAD}$.

H. Identifying Compressible Graphs

As discussed in Section VI-E, the performance of matrix multiplication using the CBM format is strongly influenced by the compression ratio obtained. To avoid compressing graphs that are poorly suited to be represent in our format, it would be useful to identify commonly available metrics that correlate well to the graph's compression ratio. We found the average clustering coefficient to be a reasonable indicator. While our sample size is limited, the values presented in Table V suggest

TABLE IV: **Performance analysis of the inference stage of a two-layer GCN using CSR and CBM formats.** To keep the analysis concise, we only consider the values of α that yielded the best speedup for $\mathbf{AX}$. The value below the speedup indicates the difference in speedup between $\mathbf{DADX}$ and the inference of a two-layer GCN when using our format.

Graph	Alpha (Cores)	GCN		
		T_{CSR} [s]	T_{CBM} [s]	$\frac{T_{CSR}}{T_{CBM}}$
ca-HepPh	α = 4 (1 Core)	0.1252 (± 0.0006)	0.1049 (± 0.0012)	1.1941 (- 0.6103)
	α = 1 (16 Cores)	0.0182 (± 0.0003)	0.0165 (± 0.0005)	1.1050 (- 0.3184)
ca-AstroPh	α = 2 (1 Core)	0.2286 (± 0.0006)	0.2018 (± 0.0031)	1.1330 (- 0.2551)
	α = 8 (16 Cores)	0.0294 (± 0.0011)	0.0277 (± 0.0004)	1.0585 (- 0.0562)
Cora	α = 2 (1 Core)	0.0185 (± 0.0027)	0.0185 (± 0.0025)	1.0001 (- 0.0224)
	α = 4 (16 Cores)	0.0034 (± 0.0001)	0.0034 (± 0.0001)	0.9799 (+ 0.0109)
PubMed	α = 4 (1 Core)	0.1718 (± 0.0011)	0.1735 (± 0.0006)	0.9902 (- 0.0157)
	α = 16 (16 Cores)	0.0251 (± 0.0010)	0.0246 (± 0.0005)	1.0210 (+ 0.038)
COLLAB	α = 4 (1 Core)	4.9576 (± 0.0729)	3.1723 (± 0.0709)	1.5627 (- 2.3725)
	α = 16 (16 Cores)	0.8271 (± 0.0052)	0.4100 (± 0.0054)	2.0171 (- 3.1521)
coPapersDBLP	α = 4 (1 Core)	8.0435 (± 0.0585)	5.4701 (± 0.0512)	1.4704 (- 0988)
	α = 32 (16 Cores)	1.3449 (± 0.0060)	0.7964 (± 0.0053)	1.6888 (- 1.0170)
coPapersCiteseer	α = 4 (1 Core)	6.9310 (± 0.0023)	4.1237 (± 0.0417)	1.6808 (- 1.8250)
	α = 32 (16 Cores)	1.3823 (± 0.0041)	0.5566 (± 0.0050)	2.4833 (- 2.6214)
ogbn-proteins	α = 8 (1 Core)	19.7282 (± 0.0052)	10.9071 (± 0.9411)	1.8088 (- 0.2334)
	α = 1 (16 Cores)	4.5482 (± 0.0068)	2.9142 (± 0.0188)	1.5607 (- 0.2090)

a positive correlation between the graphs' average clustering coefficient and compression ratio. Nevertheless, there are cases where this metric is not informative. For instance, both citation networks have small average degrees, leading to poor compression ratios regardless of their average clustering coefficients. Additionally, ogbn-proteins exhibits a better compression ratio than ca-AstroPh, even though the latter presents a significantly higher average clustering coefficient. Finally, it is important to note that if the average clustering coefficient of a graph is not readily available, measuring it may not be worthwhile, as the time required to compute it is comparable to the time needed to compress the graph in CBM format.

VII. Related Work

The matrix-matrix and matrix-vector products have been extensively studied [18]. A particular case is the product of a binary sparse matrix by a real-valued vector or (dense) matrix,

TABLE V: **Compression ratio and average clustering coefficient for different graphs.** The compression ratio is obtained by representing the graphs in CBM format with $\alpha = 0$.

Graph	Average Degree	Average Clustering Coefficient	$\frac{S_{CSR}}{S_{CBM}} \uparrow$
Cora	4.8	0.24	1.04
PubMed	5.4	0.06	1.04
ca-AstroPh	22.1	0.63	1.72
ogbn-proteins	298.5	0.28	2.14
ca-HepPh	20.7	0.61	2.72
coPapersDBLP	57.4	0.80	5.97
coPapersCiteseer	74.8	0.83	9.87
COLLAB	65.9	0.89	11.00

where the efficient representation of the binary matrix can be exploited to improve both the memory footprint and the operation running time. Although this was not expected in some preliminary studies [19], and impossibility results exist for more complex compression schemes [20], it works for some representational compression schemes.

The Single Tree Adjacency Forest (STAF) [1] represents a binary matrix by reversing and inserting the adjacency list of each row into a trie data-structure, with common row suffixes being represented exactly once. STAF enables fast matrix-matrix products by traversing the trie in topological order, while accumulating common partial sums. The number of operations required to multiply a binary matrix represented by a STAF and a real-valued vector is proportional to the size of the trie and upper-bounded by the number of non-zero elements of the binary matrix. STAF does not exploit however row-wise similarities beyond common row suffixes, and authors proposed to represent instead sets of columns, achieving a significant speedup and memory footprint reduction against CSR and the Eigen library.

Francisco *et al.* [2] explored how succinct representations for binary matrices and graphs could speedup binary matrix products. They consider both Webgraph [21] and Biclique Extraction (BE) [22] representations, which exploit similarity among rows and clustering effects found in real-world graphs and matrices. Such methods allow to reduce the memory footprint of binary matrices and accelerate the product of compressed binary matrices and real-valued vectors. The key observation is that, in both cases, we can reuse partial results from previous computations. These representations were however designed with focus on achieving the best possible compression ratio, employing more evolved representations and requiring non-trivial pre-processing steps such as node reordering through graph clustering methods or finding maximal bicliques, an NP-hard problem in general.

Elgohary *et al.* [23] also addressed the problem that large-scale machine learning algorithms are often iterative, using repeated read-only data access and I/O-bound matrix-vector multiplications, introducing Compressed Linear Algebra (CLA)

for lossless matrix compression. CLA also executes linear algebra operations directly on the compressed representations, but it is not focused on binary matrices and it only presents performance gains when data does not fit into memory.

Our work is related to the work by Björklund and Lingas [3], that considers a weighted graph on the rows of a binary matrix where the weight of an edge between two rows is equal to its Hamming distance, and then relies on a minimum spanning tree of that graph to differentially compress the rows. Authors consider however only the single product of two binary matrices, and do not guarantee that the number of scalar operations incurred by their method is less than or equal to those of standard sparse formats. Additionally, the authors do not consider the overhead imposed by operating on a compressed representation of a binary matrix, as their results are purely theoretical.

VIII. Final Remarks

In this work we have proposed the Compressed Binary Matrix (CBM) format which simultaneously reduces the memory footprint of unweighted graphs and row- and column-scaled binary matrices, and enables the implementation of new matrix multiplication kernels that are potentially much faster than the current state-of-the-art. Experimental results showed that our format achieves compression ratios of up to 11× for real-world graphs compared to the widely-used CSR format, boosting the performance of matrix multiplication above 5× in parallel environments. Additionally, integrating the CBM format into PyTorch reduced the inference time of a two-layer GCN by 2.48×. We highlight that our format is future-proof, since future optimizations to state-of-the-art sparse-dense matrix multiplication kernels will also accelerate our matrix multiplication kernels. We found that the CBM format can be built in a reasonable amount time. However, to accelerate matrix-matrix products, the graph's adjacency matrix must be previously stored in our format as a pre-processing step, avoiding the construction overhead. Additionally, the average clustering coefficient appears to be a decent metric to identify which graphs are well-suited for our format.

The current implementation of the CBM format's compression algorithm is not memory-efficient, limiting our ability to compress networks that are larger than ogbn-proteins. This issue arises because we compute $\mathbf{AA^T}$, where $\mathbf{A}$ is the graph's adjacency matrix, to accelerate the construction of the format. While $\mathbf{A}$ is usually extremely sparse, matrix $\mathbf{AA^T}$ might be significantly denser and not fit in main-memory, meaning the graph cannot be compressed. We observed this issue with the Reddit dataset [14]. Although its CSR representation requires only 0.9 GiB, the construction of the CBM format for this graph utilized over 92 GiB of memory. Future work concerns scaling the compression algorithm of the CBM format by clustering similar rows of the graph's adjacency matrix and subsequently computing a partial CBM format for each cluster. This strategy should not only reduce the memory requirements of our compression algorithm, but also reduce the amount of work and expose more parallelism opportunities. Additionally, we plan to integrate and evaluate the CBM format in the context of different GNNs architectures, and also targeting the training stage of this networks. We also intend to implement and evaluate our format and corresponding multiplication kernels in GPU architectures.

References

[1] M. Nishino, N. Yasuda, S. i. Minato, and M. Nagata, "Accelerating graph adjacency matrix multiplications with adjacency forest," in *Proceedings of the 2014 SIAM International Conference on Data Mining*. SIAM, 2014, pp. 1073–1081.

[2] A. P. Francisco, T. Gagie, D. Köppl, S. Ladra, and G. Navarro, "Graph compression for adjacency-matrix multiplication," *SN Computer Science*, vol. 3, no. 3, p. 193, 2022.

[3] A. Björklund and A. Lingas, "Fast boolean matrix multiplication for highly clustered data," in *Algorithms and Data Structures: 7th International Workshop, WADS 2001*. Springer, 2001, pp. 258–263.

[4] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, M. Suo, P. Tillet, E. Wang, X. Wang, W. Wen, S. Zhang, X. Zhao, K. Zhou, R. Zou, A. Mathews, G. Chanan, P. Wu, and S. Chintala, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, 2024.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[6] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, 2017, p. 1025–1035.

[7] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019.

[8] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[9] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[10] Y. Shao, H. Li, X. Gu, H. Yin, Y. Li, X. Miao, W. Zhang, B. Cui, and L. Chen, "Distributed graph neural network training: A survey," *ACM Comput. Surv.*, vol. 56, no. 8, 2024.

[11] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang, "Understanding GNN computational graph: A coordinated computation, io, and memory perspective," in *Proceedings of the Fifth Conference on Machine Learning and Systems, MLSys 2022*, 2022.

[12] M. Böther, O. Kißig, and C. Weyand, "Efficiently computing directed minimum spanning trees," in *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2023, pp. 86–95.

[13] Z. Yang, W. W. Cohen, and R. Salakhutdinov, "Revisiting semi-supervised learning with graph embeddings," in *Proceedings of the 33rd International Conference on Machine Learning*, ser. ICML '16, 2016, pp. 40–48.

[14] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Jun. 2014.

[15] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann, "Tudataset: A collection of benchmark datasets for learning with graphs," in *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020.

[16] R. A. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI Conference on Artificial Intelligence*, 2015.

[17] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *Advances in neural information processing systems*, vol. 33, pp. 22 118–22 133, 2020.

[18] J. Alman and V. V. Williams, "Further limitations of the known approaches for matrix multiplication," in *9th Innovations in Theoretical Computer Science Conference (ITCS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. R. Karlin, Ed., vol. 94. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018, pp. 25:1–25:15.

[19] C. Karande, K. Chellapilla, and R. Andersen, "Speeding up algorithms on compressed web graphs," *Internet Mathematics*, vol. 6, no. 3, pp. 227–256, 2009.

[20] A. Abboud, A. Backurs, K. Bringmann, and M. Künnemann, "Impossibility results for grammar-compressed linear algebra," *Advances in Neural Information Processing Systems*, vol. 33, pp. 8810–8823, 2020.

[21] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 595–602.

[22] C. Hernández and G. Navarro, "Compressed representations for web and social graphs," *Knowledge and Information Systems*, vol. 40, no. 2, pp. 279–313, 2014.

[23] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, "Compressed linear algebra for declarative large-scale machine learning," *Communications of the ACM*, vol. 62, no. 5, pp. 83–91, 2019.