High-Performance and Programmable Attentional Graph Neural Networks with Global Tensor Formulations

Maciej Besta[*]
ETH Zurich

Paweł Renc
AGH-UST
Sano Centre for Computational Medicine

Robert Gerstenberger
ETH Zurich

Paolo Sylos Labini
Free University of Bozen-Bolzano
ETH Zurich

Alexandros Ziogas
ETH Zurich

Tiancheng Chen
ETH Zurich

Lukas Gianinazzi
ETH Zurich

Florian Scheidl
ETH Zurich

Kalman Szenes
ETH Zurich

Armon Carigiet
ETH Zurich

Patrick Iff
ETH Zurich

Grzegorz Kwasniewski
NextSilicon

Raghavendra Kanakagiri
University of Illinois at Urbana-Champaign

Chio Ge
ETH Zurich

Sammy Jaeger
ETH Zurich

Jarosław Wąs
AGH-UST

Flavio Vella
University of Trento

Torsten Hoefler[*]
ETH Zurich

ABSTRACT

Graph attention models (A-GNNs), a type of Graph Neural Networks (GNNs), have been shown to be more powerful than simpler convolutional GNNs (C-GNNs). However, A-GNNs are more complex to program and difficult to scale. To address this, we develop a novel mathematical formulation, based on tensors that group all the feature vectors, targeting both training and inference of A-GNNs. The formulation enables straightforward adoption of communication-minimizing routines, it fosters optimizations such as vectorization, and it enables seamless integration with established linear algebra DSLs or libraries such as GraphBLAS. Our implementation uses a data redistribution scheme explicitly developed for sparse-dense tensor operations used heavily in GNNs, and fusing optimizations that further minimize memory usage and communication cost. We ensure theoretical asymptotic reductions in communicated data compared to the established message-passing GNN paradigm. Finally, we provide excellent scalability and speedups of even 4–5x over modern libraries such as Deep Graph Library.

[*]Corresponding authors.

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Theory of computation** → *Data structures and algorithms for data management*; • **Computer systems organization** → Distributed architectures.

Keywords

Graph Attention Models, Graph Neural Networks, Sparse-Dense Tensor Operations

ACM Reference Format:
Maciej Besta, Paweł Renc, Robert Gerstenberger, Paolo Sylos Labini, Alexandros Ziogas, Tiancheng Chen, Lukas Gianinazzi, Florian Scheidl, Kalman Szenes, Armon Carigiet, Patrick Iff, Grzegorz Kwasniewski, Raghavendra Kanakagiri, Chio Ge, Sammy Jaeger, Jarosław Wąs, Flavio Vella, Torsten Hoefler. 2023. High-Performance and Programmable Attentional Graph Neural Networks with Global Tensor Formulations. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581784.3607067

Website:
http://spcl.inf.ethz.ch/Research/Parallel_Programming/ScalableAGNNs

1 INTRODUCTION

Graph neural networks (GNNs) have become an established part of the machine learning (ML) landscape [22, 87]. GNNs are used to conduct ML tasks over interconnected data, such as nodes, links, or whole graphs. They are used in social sciences, bioinformatics, chemistry, medicine, cybersecurity, linguistics, transportation, and others [7, 9, 15, 21, 22, 25, 35, 36, 39, 44, 87, 97, 100].

The initial successful Graph Convolution Network (GCN) [49] model belongs to a class of *Convolutional GNNs (C-GNNs)* [14], with other notable C-GNN examples being Graph Isomorphism Network (GIN) [88], ChebNet [27], or Simple Graph Convolution (SGC) [85]. While C-GNNs can be used to solve many graph ML tasks, recent results in the graph ML domain illustrate that models from the class of *Attentional GNNs (A-GNNs)* are more powerful both empirically and theoretically than C-GNNs. An example A-GNN model is the seminal Graph Attention Network (GAT) [78]. One of the recent A-GNN success stories is AlphaFold, a model based on graph attention, which enabled a major breakthrough in the protein folding prediction problem [45]. Other example notable use cases include learning interactions between cells [61], predicting functional effects of gene alterations [93], solving various combinatorial problems [20], or predicting protein-protein interactions [33].

Despite their predictive power, GNN computations do require much larger scales. First, the graphs considered so far in GNNs are nowhere near the sizes used in many real scenarios. While recent efforts in graph ML have illustrated how to process *parts of* the Microsoft Academic Knowledge Graph (MAKG) [40] with ≈1.5 undirected billion edges, this graph is still tiny compared to Web Data Commons (a web crawl obtained a decade ago, with 128 billion links), the Facebook social graph (1 trillion edges already back in 2015 [24]), or the Sogou webgraph (12 trillion links in 2018 [57]). Second, the models used are miniscule compared to their analogs used in other domains such as neural language models (NLPs) [16].

In parallel and distributed computations, data communication is the primary performance bottleneck [12, 51–53, 70, 101]. To alleviate this, many powerful *communication-minimizing tensor computations* such as 2.5D matrix multiplications have been proposed. These techniques have recently been applied to the GCN model [75], reducing communication in GCN training and inference pipelines. Importantly, this was possible because a single GNN layer in the GCN model can be straightforwardly expressed as $ReLU(\mathcal{A}\mathbf{HW})$, where $\mathcal{A}$ is the graph adjacency matrix, $\mathbf{H}$ is a vertex feature matrix, and $\mathbf{W}$ is a parameter matrix. Such a formulation is called *global (GL)* [8] because one explicitly uses matrices grouping all feature vectors. The GCN model – like all other C-GNNs – also has a so-called *local (LC)* formulation, in which one specifies transformations of feature vectors of *individual vertices*. For the GCN model, the LC formulation to obtain the feature vector $\mathbf{h}_v$ for a vertex v is $\mathbf{h}_v = ReLU\left(\mathbf{W} \cdot \left(\sum_{u \in \widehat{N}(v)} \left(1/\sqrt{d_v d_u}\right) \mathbf{h}_u\right)\right)$, where[1] $\widehat{N}(v)$ is v's neighborhood together with v (i.e., $\widehat{N}(v) = N(v) \cup \{v\}$), and d_v is the degree of a vertex v. While LC formulations are widely used because they are simple to formulate, GL formulations simplify harnessing communication-minimizing routines from tensor computations, and come with higher potential for vectorization, because all vertex feature vectors and neighborhoods are grouped together.

Unfortunately, global formulations cannot be easily applied to many A-GNNs. This is because A-GNNs come with much more complex mathematical formulations than C-GNNs. For example, in GAT, the local formulation is $\mathbf{h}'_v = \mathbf{W} \cdot \sum_{u \in \widehat{N}(v)} \psi_{v,u}$, where $\psi_{v,u} = \frac{\exp\left(\sigma\left(\mathbf{a}^T \cdot \left[\mathbf{W}\mathbf{h}_v \middle\| \mathbf{W}\mathbf{h}_u\right]\right)\right)}{\sum_{y \in \widehat{N}(v)} \exp\left(\sigma\left(\mathbf{a}^T \cdot \left[\mathbf{W}\mathbf{h}_v \middle\| \mathbf{W}\mathbf{h}_y\right]\right)\right)} \mathbf{h}_u$. Here, $\mathbf{a}$ is a parameter vector and $\|$ denotes vector concatenation. The corresponding global formulation has so far been unknown. On top of that, the above expression only defines a single GNN layer of the forward propagation pass (or of the inference). To fully describe a GNN model, one would also need a GL formulation of backward propagation.

To address this, we develop novel global formulations for A-GNNs, for both training and inference (**contribution #1**). We cover all parts of a GNN pipeline, including edge message functions, arbitrary vertex aggregations (expressed with novel semirings), and normalizations (including softmax). Second, we illustrate the efficient design and implementation of our formulations using communication-minimizing sparse-dense tensor algebra kernels combined with performance-centric optimizations such as kernel fusion (**contribution #2**). While we offer a custom implementation, our general formulations enable seamless harnessing of any existing tensor library such as Combinatorial BLAS [18] or CTF [71].

We conduct a theoretical communication-cost analysis, showing that our global GNN formulations are fundamentally more efficient than the established non-global ones (**contribution #3**) Finally, the empirical evaluation illustrates the superiority of global tensor formulations, achieving significant speedups (4–5×) over state-of-the-art baselines such as Deep Graph Library (DGL) [83] (**contribution #4**). Both our formulations and design optimizations are reusable to GNN models beyond those considered in this work.

Our work covers both inference and training. In GNN training, there are two approaches, mini-batch and full-batch training. The former has been a subject of intense studies [8, 56, 58, 59, 80, 92, 98], but it suffers from information loss caused by sampling, and ultimately slower convergence [43]. Simultaneously, full-batch training has been shown to alleviate the convergence speed problems while being competitive in performance with mini-batch training [43, 75]. In this work, we focus on full-batch training. Our work significantly extends the previous schemes by Tripathy et al. [75] – which introduced communication-minimizing routines for the simple GCN model – into the more powerful but also more complex A-GNNs.

2 GRAPH NEURAL NETWORKS

Most GNN models are composed of *L layers*. Each such layer conducts two fundamental operations, *aggregation* and *update*, on the *feature vectors* of vertices and possibly of edges. Feature vectors initially contain information about an ML task at hand, aggregation combines the features of the neighbors of a vertex with that vertex' features (e.g., by using summation), and update combines the aggregation outcomes with the feature vector from the previous layer. This is usually followed by a *non-linear activation* (e.g., ReLU) and *normalization* over features. Feature vectors in intermediate layers are often referred to as hidden vectors. The last GNN layer outputs feature vectors used for the downstream ML tasks.

2.1 General Notation

The input graph G is modeled as a tuple (V, E); V is a set of vertices and $E \subseteq V \times V$ is a set of edges; $|V| = n$ and $|E| = m$. $N(i)$ denotes the set of vertices adjacent to vertex (node) i, d_i is i's degree, and d is the maximum degree in G. The adjacency matrix (AM) of a graph is $\mathcal{A} \in \{0, 1\}^{n \times n}$. $\mathcal{A}$ determines the connectivity of vertices:

[1]We follow a convention used in the GNN literature, where an individual feature vector $\mathbf{h}_v$ is a *column* vector (the weight matrix $\mathbf{W}$ multiplies it from the *left* side: $\mathbf{W}\mathbf{h}_v$), but the feature matrix $\mathbf{H}$ consists of *row* feature vectors (the weight matrix $\mathbf{W}$ multiplies it from the *right* side: $\mathbf{HW}$)

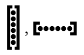
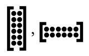
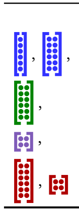
Symbol	Description
	Dense vectors, **dimensions**: $k \times 1$, $1 \times k$. They usually contain model parameters (in addition to the parameter matrix $\mathbf{W}$), e.g., in GAT.
	Dense vectors, **dimensions**: $n \times 1$, $1 \times n$. They implement transformations, such as replication, or store norms of feature vectors.
	Dense matrices, **dimensions**: $k \times k$. They keep model parameters.
	Dense matrices, **dimensions**: $n \times k$, $k \times n$. They are used primarily to maintain features or gradients for each vertex.
	Dense matrix, **dimensions**: $n \times n$. It is used when obtaining attention scores for each edge. The gray color indicates that this matrix is *always* virtual, i.e., we never instantiate it explicitly, and it is instead computed in parts, according to a specified schedule.
	Sparse matrix, **dimensions**: $n \times n$. It stores the graph structure (the adjacency matrix), or attention scores for each edge.
Color code:	
	**Blue** indicates vectors or matrices that *contain ones only*. They enable expressing various operations (e.g., row summation or replication) as tensor algebra kernels, fostering optimizations and integration with tensor libraries such as GraphBLAS [46] or CTF [71]. **Green** indicates features (usually with tall dense matrices and vectors). **Violet** indicates weights (usually with square dense matrices). **Red** indicates gradients (usually with tall/square dense matrices). Gray indicates a virtual matrix (see above). **Black** indicates none of the above or no need for using a color code.

Table 1: Important matrix and vector objects used in this work.

$\mathcal{A}(i, j) = 1 \Leftrightarrow (i, j) \in E$. In the following, to simplify expressions, we use a symbol $\mathcal{A}$ to also denote the adjacency matrix after any form of normalization. Note that we denote sparse tensors such as the adjacency matrix with calligraphic letters.

$\mathbf{h}_i \in \mathbb{R}^k$ denotes the hidden feature vector of a vertex i. These vectors can be grouped in a matrix $\mathbf{H} \in \mathbb{R}^{n \times k}$. k is the dimensionality of feature vectors; l indicates the l-th GNN layer (i.e., $\mathbf{h}_i^l$). Note that while k may be different in different GNN layers, it is always much smaller than n. For clarity of equations and without the loss of generality, we assume it to be the same across GNN layers.

2.2 Local Formulations of GNN Models

In a local formulation, the feature vector of each neighbor of a vertex i is first transformed with a function ψ. Next, for each vertex i, the resulting feature vectors of i's neighbors are aggregated with a binary function $\bigoplus$ (e.g., a sum). Third, one uses another function φ to process the outcome of $\bigoplus$. φ is often implemented as a linear projection or MLP, followed by a non-linearity. The outcome of φ becomes the feature vector $\mathbf{h}_i$ in the next GNN layer. We obtain $\mathbf{h}_i^{l+1} = \varphi\left(\mathbf{h}_i^l, \bigoplus_{j \in N(i)} \psi\left(\mathbf{h}_i^l, \mathbf{h}_j^l\right)\right)$. Importantly, depending on the details of ψ, one can distinguish several *classes of GNN models* [14]. First, in **Convolutional GNNs (C-GNNs)**, ψ outputs a fixed *scalar* coefficient that can be preprocessed. Second, in **Attentional GNNs (A-GNNs)**, ψ is a *learnable function* that returns a scalar coefficient.

We illustrate detailed local formulations of selected GNN models in the technical report; these are all known formulations published in the GNN literature, and grouped in a recent analysis [8].

3 TENSOR ALGEBRA BUILDING BLOCKS

First, we picture the used tensors with small figures to indicate their shapes, densities, and dimensions; see Table 1 for a list.

Second, we identify and list commonly occurring tensor algebra expressions in Table 2; we use them heavily in the formulations in the following sections, with this table serving as a reference point. On one hand, exposing these expressions *enables simplifying and shortening* tensor formulations of A-GNNs. On the other hand, *it facilitates compute efficiency*. This is because many of these expressions occur more than once in a single GNN layer of a given model. Thus, each such identified common expression has to be computed only once when processing that layer.

We pay special attention to express each GNN model *solely* with tensor kernels. To achieve this, we use expressions such as replication of vectors (rep) or summation of matrix rows (sum). Such an approach facilitates optimizations such as vectorization and it fosters programmability: one can seamlessly use our formulations with established tensor libraries and DSLs, for example Combinatorial BLAS [4, 18], GraphMat [72], GraphBLAST [91], or CTF [71].

We also list common tensor algebra kernels such as sparse–dense matrix product (SpMM) or sampled dense–dense matrix product (SDDMM), see the bottom part of Table 2. To facilitate performance optimizations, we also identify two new kernels that occur commonly in GNNs: SpMMM (sparse matrix–dense matrix–dense matrix product) and MSpMM (dense matrix–sparse matrix–dense matrix product). They form compute patterns in forward and backward passes, respectively. Then, when constructing the GL formulations of A-GNNs, we identify the occurrences of these kernels (cf. Fig. 1). This further fosters achieving high performance and programmability because one can simply plug in tuned implementations of these established kernels (e.g., coming from the above-mentioned libraries) instead of developing their own code.

We now proceed to develop GL formulations for A-GNNs.

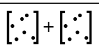
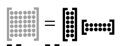
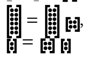
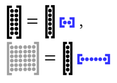
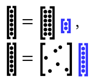
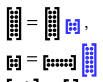
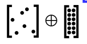
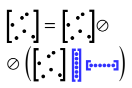
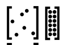
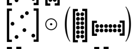
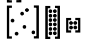
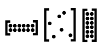
	Description	Sparsity/density
$\mathcal{X}_+ = \mathcal{X} + \mathcal{X}^T$	★ Add a matrix $\mathcal{X}$ to its transpose; the calligraphic font indicates that $\mathcal{X}$ is sparse in all used models.	
$\mathbf{X}_\times = \mathbf{X}\mathbf{X}^T$	★ Multiply a tall matrix $\mathbf{X}$ by its transpose.	
$\mathbf{H}' = \mathbf{H}\mathbf{W}$, $\mathbf{a}' = \mathbf{W}\mathbf{a}$	**Projection** Multiply a feature matrix $\mathbf{H}$ or a vector $\mathbf{a}$ with a parameter matrix $\mathbf{W}$, usually $\mathbf{a}$ also contains model parameters.	
$\text{rep}_i(\mathbf{x}) = \mathbf{x}\mathbf{1}^T$	★ **Replication** Multiply a column vector $\mathbf{x}$ by a row vector (with i ones) in order to *replicate* $\mathbf{x}$ i times ($\mathbf{x}$ can have an arbitrary dimension, usually n or k). **Transposition:** $(\text{rep}_i(\mathbf{x}))^T \equiv \text{rep}_i^T(\mathbf{x})$	
$\text{sum}(\mathbf{X}) = \mathbf{X}\mathbf{1}$	★ **Summation** Multiply $\mathbf{X}$ by a column vector of ones in order to *obtain the sum of each row* of $\mathbf{X}$ ($\mathbf{X}$ is most often a dense matrix in $\mathbb{R}^{n \times k}$, or a sparse matrix in $\mathbb{R}^{n \times n}$). **Transposition:** $(\text{sum}(\mathbf{X}))^T \equiv \text{sum}^T(\mathbf{X})$.	
$\text{rs}_i(\mathbf{X})$	★ **Composition** $\text{rs}_i(\mathbf{X}) \equiv \text{rep}_i(\text{sum}(\mathbf{X}))$. It is equivalent to a multiplication by a *matrix of ones*. **Transposition:** $(\text{rs}_i(\mathbf{X}))^T \equiv \text{rs}_i^T(\mathbf{X})$.	
$\mathcal{X} \oplus \mathbf{X}$	★ **Arbitrary aggregation** It is a sparse–dense matrix product over *arbitrary* semirings. Details: Section 4.3.	
$\text{sm}(\mathcal{X})$	★ **Softmax normalization** It can be used with any matrices; in the used models, it is used for $n \times n$ sparse matrices. Details: Section 4.2. $\text{sm}(\mathcal{X}) = \exp(\mathcal{X}) \oslash \text{rs}_n(\exp(\mathcal{X}))$.	
SpMM	Sparse matrix–dense matrix product.	
SDDMM	Sampled dense matrix–dense matrix product.	
MM	Dense matrix–dense matrix product.	
SpMMM	★ Sparse matrix–dense matrix–dense matrix product. It is a composition of SpMM and MM.	
MSpMM	★ Dense matrix–sparse matrix–dense matrix product. It is a composition of MM and SpMM.	

Table 2: Important tensor algebra expressions (top part) and selected most important compute kernels (bottom part) used throughout the paper. "★" indicates a novel building block derived or identified in this work.

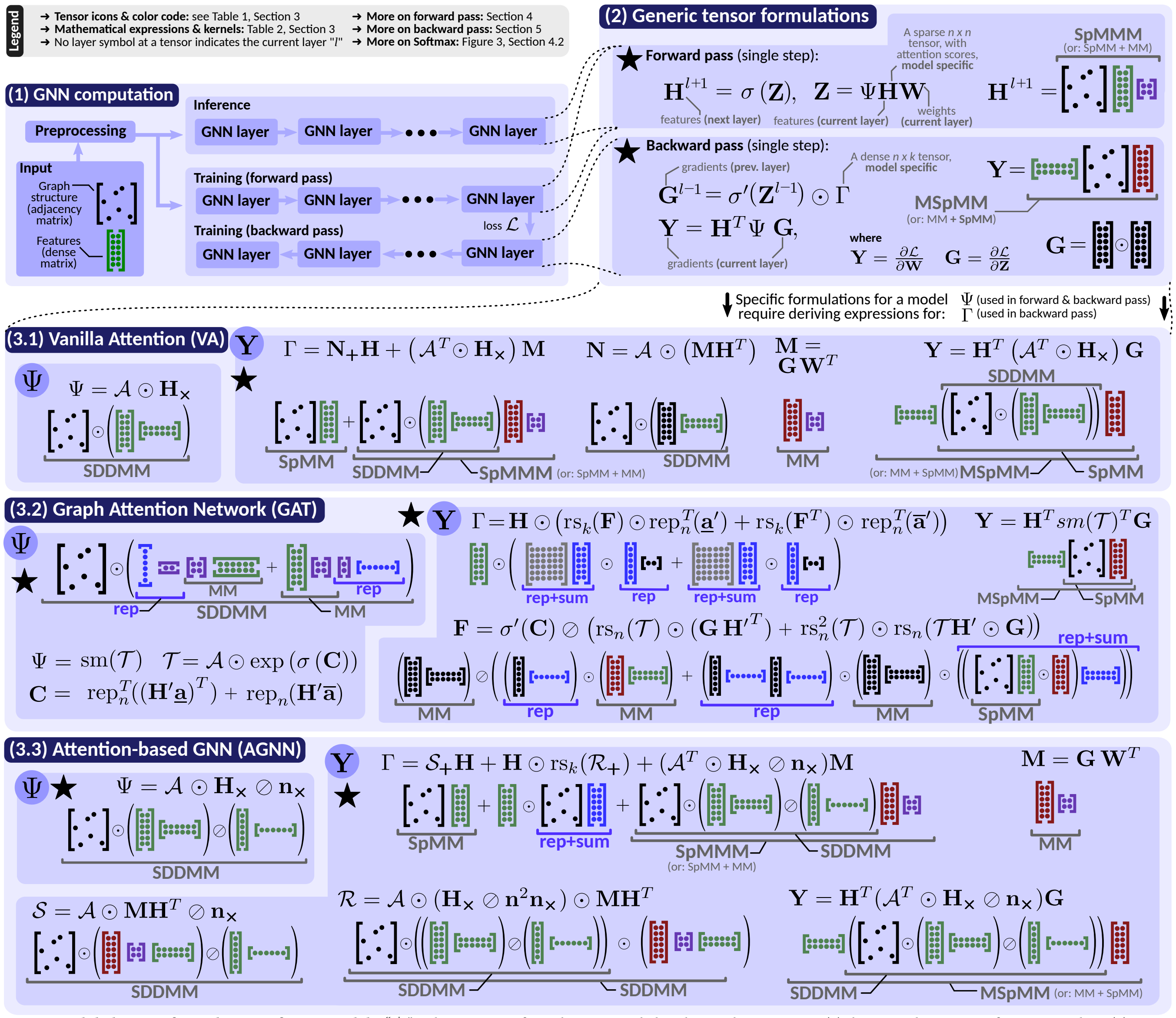

Legend
→ Tensor icons & color code: see Table 1, Section 3
→ Mathematical expressions & kernels: Table 2, Section 3
→ No layer symbol at a tensor indicates the current layer "l"
→ More on forward pass: Section 4
→ More on backward pass: Section 5
→ More on Softmax: Figure 3, Section 4.2
(1) GNN computation
Preprocessing
Input
Graph structure (adjacency matrix)
Features (dense matrix)
Inference
GNN layer
Training (forward pass)
Training (backward pass)
loss $\mathcal{L}$
(2) Generic tensor formulations
Forward pass (single step):
$H^{l+1} = \sigma(Z), \quad Z = \Psi H W$
features (next layer)
features (current layer)
weights (current layer)
A sparse n x n tensor, with attention scores, model specific
SpMMM (or: SpMM + MM)
Backward pass (single step):
gradients (prev. layer)
A dense n x k tensor, model specific
$G^{l-1} = \sigma'(Z^{l-1}) \odot \Gamma$
$Y = H^T \Psi G,$
gradients (current layer)
MSpMM (or: MM + SpMM)
where $Y = \frac{\partial \mathcal{L}}{\partial W}$ $G = \frac{\partial \mathcal{L}}{\partial Z}$
Specific formulations for a model require deriving expressions for: Ψ (used in forward & backward pass) Γ (used in backward pass)
(3.1) Vanilla Attention (VA)
$\Psi = \mathcal{A} \odot H_\times$
SDDMM
$\Gamma = N_+ H + (\mathcal{A}^T \odot H_\times) M$
$N = \mathcal{A} \odot (MH^T)$
$M = GW^T$
$Y = H^T (\mathcal{A}^T \odot H_\times) G$
SpMM
SpMMM (or: SpMM + MM)
MM
(or: MM + SpMM) MSpMM
(3.2) Graph Attention Network (GAT)
rep
MM
$\Gamma = H \odot (rs_k(F) \odot rep_n^T(\underline{a}') + rs_k(F^T) \odot rep_n^T(\overline{a}'))$
$Y = H^T sm(\mathcal{T})^T G$
rep+sum
$F = \sigma'(C) \oslash (rs_n(\mathcal{T}) \odot (G H'^T) + rs_n^2(\mathcal{T}) \odot rs_n(\mathcal{T} H' \odot G))$
$\Psi = sm(\mathcal{T}) \quad \mathcal{T} = \mathcal{A} \odot \exp(\sigma(C))$
$C = rep_n^T((H'\underline{a})^T) + rep_n(H'\overline{a})$
(3.3) Attention-based GNN (AGNN)
$\Psi = \mathcal{A} \odot H_\times \oslash n_\times$
$\Gamma = \mathcal{S}_+ H + H \odot rs_k(\mathcal{R}_+) + (\mathcal{A}^T \odot H_\times \oslash n_\times) M$
$M = G W^T$
$\mathcal{S} = \mathcal{A} \odot MH^T \oslash n_\times$
$\mathcal{R} = \mathcal{A} \odot (H_\times \oslash n^2 n_\times) \odot MH^T$
$Y = H^T (\mathcal{A}^T \odot H_\times \oslash n_\times) G$
MSpMM (or: MM + SpMM)

Figure 1: Global tensor formulations of GNN models. "★" indicates a new formulation provided in this work. We present **(1)** the general overview of a GNN pipeline, **(2)** tensor formulations for a general model-agnostic forward and backward GNN layer, and **(3)** model-specific tensor formulations for VA, AGNN, and GAT. For each model, we illustrate its tensor formulations, the associated tensor compute patterns (cf. Table 1), and we identify example decompositions of these patterns into individual compute kernels (cf. the bottom part of Table 2) and other tensor building blocks (cf. the top part of Table 2). More details on the symbols and tensor building blocks can be found in Section 3. The details of the derivations of the backward pass propagation for each model are provided in the extended technical report, and summarized in Section 5.

4 TENSOR FORMULATIONS FOR INFERENCE

To facilitate linear algebraic reasoning about GNN computations, we first develop a *generic global formulation* that is the equivalent to the general established local formulation from Section 2.2. It incorporates functions Ψ, $\oplus$, and Φ, and uses them to transform the vertex feature vectors in iteration l, grouped in a matrix H^l, into H^{l+1}, using the graph structure provided in $\mathcal{A}$. Our formulation fosters programmability – one can easily design an arbitrary A-GNN model by appropriately specifying Ψ, $\oplus$, and Φ. We have

$$H^{l+1} = \sigma\left(Z^l\right), \qquad Z^l = (\Phi \circ \oplus)\left(\Psi\left(\mathcal{A}, H^l\right), H^l\right) \qquad (1)$$

$\mathcal{A}$, together with H^l, are first transformed by Ψ, which is the equivalent of ψ in the local formulation. As such, it computes *attention scores* for edges. The output of Ψ, together with H^l, is further transformed by a composition of functions Φ and $\oplus$. We use the composition to underline the fact that, depending on the model, the user may want to apply $\oplus$ and Φ in a different order. This facilitates performance optimizations. Observe that $\oplus$ and Φ do not necessarily commute, and the model designer is responsible for using the correct order. The resulting matrix is denoted with Z^l, it is then processed with a non-linearity σ. While in the local formulation σ is implicitly incorporated into φ, in the global formulation, we *decouple* σ from Φ to enable the fact that Φ may be applied first, before $\oplus$, to achieve higher performance.

We select representative A-GNNs motivated by recent surveys [8]: vanilla attention (VA) [77], AGNN [73], and GAT [78].

4.1 Global Formulations of Ψ

We show the global formulations of Ψ of the considered GNN models in Figure 1. We also show the sparsity/density of the associated tensors (more details on the implementation are in Section 6). The simplest one, the **VA global formulation**, is known (but only for inference and forward pass – our backward pass formulation in Section 5 is novel). To obtain Ψ, one computes the product of $\mathbf{H}$ with its transpose $\mathbf{H}^T$, obtaining the dot product score for each potential edge. This operation is followed by an element-wise Hadamard product with $\mathcal{A}$ to filter the scores corresponding to each edge.

The formulations of AGNN and GAT are novel. **AGNN's global formulation** is similar to VA. The main difference is the normalization of attention scores $\mathbf{H}_{\times}$, expressed algebraically using an element-wise Hadamard division $\oslash$ by the outer product of *vectors* $\mathbf{n}$ and $\mathbf{n}^T$. The ith element of $\mathbf{n} \in \mathbb{R}^{n \times 1}$ is the L2 norm of the corresponding ith row feature vector (of vertex i) from $\mathbf{H}$.

The **GAT global formulation** of Ψ is more involved (details of derivation are pictured in Figure 2). The corresponding local expression is in Section 1. First, for each edge (i, j), one has to express the concatenation of two vectors $\mathbf{Wh}_i \| \mathbf{Wh}_j$ associated with vertices i and j, followed by a dot product with a shared vector of parameters $\mathbf{a}$. The concatenation is challenging to express with global tensor operations. To facilitate it, one can *split the dot product into a sum of two independent dot products* $(\mathbf{Wh}_i) \cdot \underline{\mathbf{a}} + (\mathbf{Wh}_j) \cdot \overline{\mathbf{a}}$, where $\mathbf{a} = (\underline{\mathbf{a}} \quad \overline{\mathbf{a}})$. To express this operation algebraically for *all* vertex pairs, we further multiply with vectors of ones ($\mathbf{1}$ and $\mathbf{1}^T$), as well as with $\mathbf{H}$ and $\mathbf{H}^T$, obtaining $\mathbf{C}$ (note that $\mathbf{C}$ is *virtual*, i.e., it is never explicitly instantiated in the actual implementation, as indicated by the gray matrix in the corresponding sparsity/density pattern). C is then transformed using element-wise non-linearity σ. The outcome is multiplied element-wise with $\mathcal{A}$ to filter out the scores of non-existing edges. Finally, the softmax normalization across vertex neighborhoods is applied.

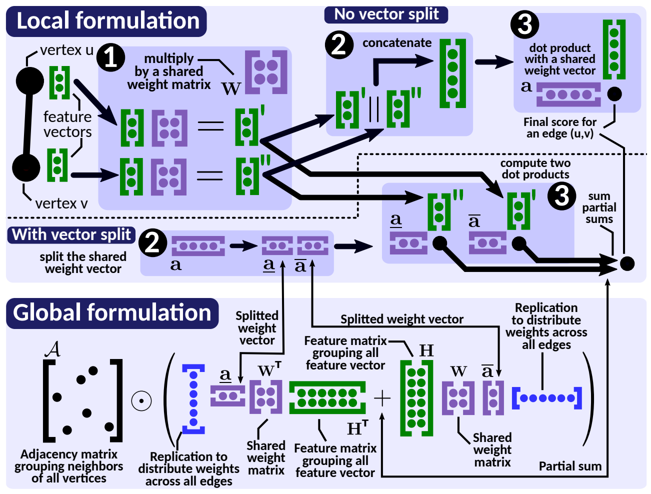

Local formulation
vertex u
feature vectors
vertex v
1
multiply by a shared weight matrix
W
No vector split
2
concatenate
3
dot product with a shared weight vector
a
Final score for an edge (u,v)
compute two dot products
3
sum partial sums
With vector split
2
split the shared weight vector
a
a
ā
ā
Global formulation
Splitted weight vector
Splitted weight vector
Replication to distribute weights across all edges
Feature matrix grouping all feature vector
H
W
Wᵀ
Hᵀ
Adjacency matrix grouping neighbors of all vertices
Replication to distribute weights across all edges
Shared weight matrix
Feature matrix grouping all feature vector
Shared weight matrix
Partial sum

Figure 2: Deriving of the global GAT formulation from the local one (softmax normalization omitted for clarity, it is plotted separately in Figure 3).

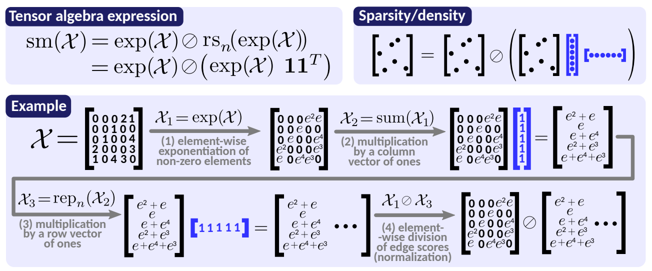

Tensor algebra expression
$\mathrm{sm}(\mathcal{X}) = \exp(\mathcal{X}) \oslash \mathrm{rs}_n(\exp(\mathcal{X}))$
$= \exp(\mathcal{X}) \oslash (\exp(\mathcal{X})\ \mathbf{1}\mathbf{1}^T)$
Sparsity/density
Example
$\mathcal{X}_1 = \exp(\mathcal{X})$
(1) element-wise exponentiation of non-zero elements
$\mathcal{X}_2 = \mathrm{sum}(\mathcal{X}_1)$
(2) multiplication by a column vector of ones
$\mathcal{X}_3 = \mathrm{rep}_n(\mathcal{X}_2)$
(3) multiplication by a row vector of ones
$\mathcal{X}_1 \oslash \mathcal{X}_3$
(4) element-wise division of edge scores (normalization)

Figure 3: Global formulation of softmax normalization.

4.2 Global Formulation of Graph Softmax

We derive the global formulation of softmax ($\mathrm{sm}(\cdot)$) applied over each vertex neighborhood, see Figure 3 and Table 2. In step (1), an element-wise exponentiation is applied to each non-zero of $\mathcal{X}$. Multiplication with a column vector of ones (step (2)) outputs another column vector $\mathbf{x} \in \mathbb{R}^{n \times 1}$, with its ith cell containing the sum of the cells in the ith row of $\exp(\mathcal{X})$. Multiplying $\mathbf{x}$ by a row vector of ones $\mathbf{1}^T \in \mathbb{R}^{1 \times n}$ (step (3)) replicates $\mathbf{x}$ n times column-wise, giving an $n \times n$ matrix, with each ith row containing n copies of the sum of values from the ith row of $\exp(\mathcal{X})$. This final matrix can be used to normalize $\exp(\mathcal{X})$ with element-wise division $\oslash$ (step (4)).

Note that in the actual design & implementation (detailed in Section 6), we never explicitly create such an $n \times n$ matrix, nor fully replicate the vectors. However, incorporating the global formulation for softmax (and other routines) fosters programmability and performance optimizations, by enabling a seamless integration with tensor libraries such as Combinatorial BLAS [4].

4.3 Global Formulation of $\oplus$

A standard sum aggregation, in which one sums the feature vectors of the neighbors of each vertex, is conducted with a sparse-dense product $\mathcal{A}\mathbf{H}$. Expressing many other aggregations (e.g., max, min, average) can be enabled by incorporating **generalized matrix products over different semirings**. A semiring is defined as a tuple $(X, op_1, op_2, el_1, el_2)$. X is a set equipped with two binary operations op_1, op_2 such that (X, op_1) and (X, op_2) are two monoids (a monoid is a set equipped with an associative binary operation). (X, op_1) is a commutative monoid with an identity element el_1, and (X, op_2) is a monoid with an identity element el_2.

Both the **max** and the **min aggregation** can be expressed using the *tropical semiring* variants [62, 63, 65, 95]. For the latter, the semiring is $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$. Here, one has to also first transform $\mathcal{A}$ by setting each off-diagonal zero entry as ∞. Then, the corresponding aggregation $\mathcal{A}\mathbf{H}$ ensures that a computed j-th feature of vertex i is equal to the minimum of the j-th features of all i's neighbors, i.e., $h_{ij}^{(l+1)} = \min_{v \in N(i)} h_{vj}$. The former is similar, i.e., the semiring is $(\mathbb{R} \cup \{-\infty\}, \max, +, -\infty, 0)$.

The **averaging aggregation** is more challenging. The semiring domain is $\mathbb{R}^2$ and each initial element x of $\mathcal{A}$ is assigned a tuple $a = (a_1, a_2) = (x, x)$. Then, the two operations op_1, op_2 are defined as follows: $a\ op_1\ b = (a_1 b_1, a_1)$ and $a\ op_2\ b = \left(\frac{a_1 a_2 + b_1 b_2}{a_2 + b_2}, a_2 + b_2\right)$. Using a tuple enables keeping track of partial sums *and* of their contributions to the final average. op_2 merges two tuples by computing the weighted average.

Note that the standard sum aggregation is actually also a matrix product over the *real semiring* $R = (\mathbb{R}, +, \cdot, 0, 1)$.

4.4 Global Formulation of Φ and $\Phi \circ \oplus$

In most GNN models, Φ is a linear projection, i.e., a multiplication with a matrix containing model parameters $\mathbf{W}$. As such, it can be applied before or after $\oplus$; we indicate this with the function composition symbol $\Phi \circ \oplus$. Our formulation gives the programmer flexibility to decide on the composition order statically or on-the-fly.

In some models, for example GIN [88], Φ is an MLP. This corresponds to a series of multiplications with different parameter matrices, interleaved with non-linearities.

Overall, nearly all A-GNNs have a formulation $\sigma(\Psi \mathbf{H}\mathbf{W})$ where $\Psi \equiv \Psi(\mathcal{A}, \mathbf{H})$ is described in Section 4.1. The corresponding sparsity/density pattern is ▦ = [⁘] × ▦ × ▦, see also Figure 1.

We note that C-GNNs have a similar general formulation of the form $\sigma(\mathcal{A}\mathbf{H}\mathbf{W})$. The difference is that instead of Ψ, one directly uses the adjacency matrix $\mathcal{A}$ (potentially normalized). *Thus, once Ψ is computed, the same execution strategies can be applied to C-GNN and A-GNN models, as their subsequent formulations are identical.*

5 TENSOR FORMULATIONS FOR TRAINING

We first derive global formulations for backpropagation for considered A-GNN. We use lowercase roman letters (i, j, k...) for vertex indices and lowercase Greek letters (α, β, γ) for feature indices.

5.1 General Backward Pass Derivation

For each model, the derivation consists of the following steps:

- **Step 1**: Obtain the derivative of $\mathbf{Z}^l$ w.r.t. $\mathbf{Z}^{l-1}$ (cf. Eq. (1)). This enables establishing the error propagation between GNN layers.
- **Step 2**: Obtain the derivative of $\mathbf{H}^l$ w.r.t. $\mathbf{Z}^{l-1}$. We observe that $H^l_{j,\beta}$ depends on $Z^l_{i,\alpha}$ iff $i = j$ and $\alpha = \beta$, based on the forward propagation pass equations. We have $\partial H^l_{j,\beta} / \partial Z^{l-1}_{i,\alpha} = \delta^{i,\alpha}_{j,\beta} \sigma'\left(Z^{l-1}_{i,\alpha}\right)$ where δ is the Kronecker symbol and it equals one iff $i = j$ and $\alpha = \beta$, and zero otherwise.
- **Step 3**: Obtain the derivative of the loss function $\mathcal{L}$ w.r.t. to $\mathbf{Z}^l$, i.e., $\partial \mathcal{L} / \partial \mathbf{Z}^l$, denoted with $\mathbf{G}^l$. It has the same dimensions as $\mathbf{Z}^l$. Using the chain rule, we have:
$$G^{l-1}_{i,\alpha} = \frac{\partial \mathcal{L}}{\partial Z^{l-1}_{i,\alpha}} = \sum_{j,\beta} \frac{\partial \mathcal{L}}{\partial Z^l_{j,\beta}} \frac{\partial Z^l_{j,\beta}}{\partial Z^{l-1}_{i,\alpha}} = \sum_{j,\beta} G^l_{j,\beta} \frac{\partial Z^l_{j,\beta}}{\partial Z^{l-1}_{i,\alpha}} \tag{2}$$
- **Step 4**: Derive a recursive relationship between $\mathbf{G}^l$ and $\mathbf{G}^{l-1}$.
- **Step 5**: Find the derivative $\mathbf{G}^l$ at the last layer L. Here, using the chain rule, we have:
$$G^L_{i,\alpha} = \frac{\partial \mathcal{L}}{\partial Z^L_{i,\alpha}} = \sum_{j,\beta} \frac{\partial \mathcal{L}}{\partial H^L_{j,\beta}} \frac{\partial H^L_{j,\beta}}{\partial Z^L_{i,\alpha}} = \frac{\partial \mathcal{L}}{\partial H^L_{i,\alpha}} \sigma'\left(Z^L_{i,\alpha}\right) \tag{3}$$
$$\mathbf{G}^L = \nabla_{\mathbf{H}^L} \mathcal{L} \odot \sigma'\left(\mathbf{Z}^L\right) \tag{4}$$
- **Step 6**: Find the derivative of $\mathcal{L}$ w.r.t. the weights $\mathbf{W}^l$; we denote it with $\mathbf{Y}^l = \partial \mathcal{L} / \partial \mathbf{W}^l$. We have:
$$Y^l_{\alpha,\beta} = \frac{\partial \mathcal{L}}{\partial W^l_{\alpha,\beta}} = \sum_{j,\gamma} \frac{\partial \mathcal{L}}{\partial Z^l_{j,\gamma}} \frac{\partial Z^l_{j,\gamma}}{\partial W^l_{\alpha,\beta}} = \sum_{j,\gamma} G^l_{j,\gamma} \frac{\partial Z^l_{j,\gamma}}{\partial W^l_{\alpha,\beta}} \tag{5}$$

After **Step 6**, one finds the update for the weights $\mathbf{W}^l$ for the next training iteration. The weights are updated in the direction of the decreasing loss, using any learning rule, such as $\mathbf{W}^l := \mathbf{W}^l - \alpha \mathbf{Y}^l$.

5.2 General Training Formulation

The general tensor algebra forward pass formulation is identical to that of inference in Section 4. To obtain a general backward pass formulation, we analyze each considered A-GNN model and we distill the outcomes into a generic backward pass formulation:
$$\mathbf{G}^{l-1} = \sigma'\left(\mathbf{Z}^{l-1}\right) \odot \Gamma^l, \tag{6}$$
$$\mathbf{Y}^l = \mathbf{H}^{l^T} \Psi\left(\mathcal{A}^T, \mathbf{H}^l\right) \mathbf{G}^l + \mathbf{G}^l \mathbf{W}^{l^T} \mathbf{H}^{l^T} \frac{\partial \Psi}{\partial \mathbf{W}^l}$$
Γ is a model-specific expression, we show and analyze it for the considered A-GNNs in Figure 1. Moreover, as stated in Section 5.1, $\mathbf{G}^l = \partial \mathcal{L} / \partial \mathbf{Z}^l$ and $\mathbf{Y}^l = \partial \mathcal{L} / \partial \mathbf{W}^l$. The whole Eq. (7) is bootstrapped at the last layer L with $\mathbf{G}^L = \nabla_{\mathbf{H}^L} \mathcal{L} \odot \sigma'\left(\mathbf{Z}^L\right)$. Finally, Ψ and $\mathbf{H}$ are the same as in the forward pass (cf. Eq. (1)), except for using $\mathcal{A}^T$ instead of $\mathcal{A}$ and $\mathbf{H}^T$ instead of $\mathbf{H}$. This reflects the fact that, *in the combinatorial interpretation, the backward pass is done on the "reversed" graph*, i.e., using the reverse edge directions (compared to the forward pass). Note that, for undirected graphs (which form the vast majority of datasets in GNN workloads [28, 41]), $\mathcal{A} = \mathcal{A}^T$. Please note that for VA and AGNN $\frac{\partial \Psi}{\partial W}$ is 0, so the second term of $\mathbf{Y}^l$ is zero as well.

We present the final global formulations of each model in Figure 1. It shows both Ψ and Γ, together with their associated compute patterns and tensors, and identified tensor kernels. Due to space constraints, we present derivations of the backward pass of AGNN and GAT in the extended technical report, and now proceed with the derivation for VA.

5.3 Vanilla Attention Backpropagation

We write the forward pass using the matrix components:
$$Z^l_{j,\beta} = \sum_{k,\gamma} A_{j,k} \left(\sum_{\zeta} H^l_{j,\zeta} H^l_{k,\zeta} \right) H^l_{k,\gamma} W^l_{\gamma,\beta} \tag{7}$$
To find the derivatives of $\mathbf{Z}^l$ w.r.t. $\mathbf{Z}^{l-1}$ (**Step 1**), we observe that only the $\mathbf{H}^l$ terms depend on $\mathbf{Z}^{l-1}$. There are three such terms and we thus use the derivative rule for the product of three functions:
$$\begin{aligned} \frac{\partial Z^l_{j,\beta}}{\partial Z^{l-1}_{i,\alpha}} = & \sum_{k,\gamma} A_{j,k} \left(\sum_{\zeta} \frac{\partial H^l_{j,\zeta}}{\partial Z^{l-1}_{i,\alpha}} H^l_{k,\zeta} \right) H^l_{k,\gamma} W^l_{\gamma,\beta} \\ & + \sum_{k,\gamma} A_{j,k} \left(\sum_{\zeta} H^l_{j,\zeta} \frac{\partial H^l_{k,\zeta}}{\partial Z^{l-1}_{i,\alpha}} \right) H^l_{k,\gamma} W^l_{\gamma,\beta} \\ & + \sum_{k,\gamma} A_{j,k} \left(\sum_{\zeta} H^l_{j,\zeta} H^l_{k,\zeta} \right) \frac{\partial H^l_{k,\gamma}}{\partial Z^{l-1}_{i,\alpha}} W^l_{\gamma,\beta} \end{aligned} \tag{8}$$
Then, we plug in the equations from **Step 2**, obtaining
$$\begin{aligned} \frac{\partial Z^l_{j,\beta}}{\partial Z^{l-1}_{i,\alpha}} = \sigma'\left(Z^{l-1}_{i,\alpha}\right) \Bigg(& \sum_{k,\gamma} A_{j,k} \delta^i_j H^l_{k,\alpha} H^l_{k,\gamma} W^l_{\gamma,\beta} \\ & + \sum_{\gamma} A_{j,i} H^l_{j,\alpha} H^l_{i,\gamma} W^l_{\gamma,\beta} + A_{j,i} \left(\sum_{\zeta} H^l_{j,\zeta} H^l_{i,\zeta} \right) W^l_{\alpha,\beta} \Bigg) \end{aligned} \tag{9}$$
Using the above and **Step 3**, we have
$$\begin{aligned} G^{l-1}_{i,\alpha} = & \sigma'\left(Z^{l-1}_{i,\alpha}\right) \sum_{\beta,k,\gamma} G^l_{i,\beta} A_{i,k} H^l_{k,\alpha} H^l_{k,\gamma} W^l_{\gamma,\beta} \\ & + \sigma'\left(Z^{l-1}_{i,\alpha}\right) \sum_{\beta,j,\gamma} G^l_{j,\beta} A_{j,i} H^l_{j,\alpha} H^l_{i,\gamma} W^l_{\gamma,\beta} \\ & + \sigma'\left(Z^{l-1}_{i,\alpha}\right) \sum_{\beta,j,\zeta} G^l_{j,\beta} A_{j,i} H^l_{j,\zeta} H^l_{i,\zeta} W^l_{\alpha,\beta} \end{aligned} \tag{10}$$

In the matrix formulation, it is (**Steps 3-5**):

$$\mathbf{G}^{l-1} = \sigma'\left(\mathbf{Z}^{l-1}\right) \odot \left(\mathbf{N}_{+}^{l}\mathbf{H}^{l} + \left(\mathcal{A}^{T} \odot \mathbf{H}_{\times}^{l}\right)\mathbf{M}^{l}\right) \tag{11}$$

where, for simplicity and to eliminate redundant matrix operations, we compute the following terms only once:

$$\mathbf{M}^{l} = \mathbf{G}^{l}\mathbf{W}^{l^{T}}, \qquad \mathbf{N}^{l} = \mathcal{A} \odot \left(\mathbf{M}^{l}\mathbf{H}^{l^{T}}\right). \tag{12}$$

We then finally arrive at the formulation for $\mathbf{Y}^{l}$ (**Step 6**):

$$\mathbf{Y}^{l} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{l}} = \mathbf{H}^{l^{T}}\left(\mathcal{A}^{T} \odot \mathbf{H}_{\times}^{l}\right)\mathbf{G}^{l} \tag{13}$$

6 DESIGN & IMPLEMENTATION

We overview our design in Figure 4. Developing a single GNN model within our framework consists of several stages. We begin with a local GNN model formulation and derive the global one using the techniques and examples from Section 4 (if a global formulation already exists, this stage is omitted). Second, we investigate the sparsity of the associated tensors and decide which ones are virtual, i.e., are never instantiated (Section 6.1). Afterward, we automatically optimize the communication volume of the model's global formulation. We achieve this by harnessing and extending the combinatorial data access model operating on Simple Overlap Access Programs (SOAP) [51]. SOAP's outcome is a parametric formulation that serves as a basis for an implementation.

6.1 Virtual Tensors for Space Optimization

Despite using distributed-memory clusters, some tensors could still be too large to be stored explicitly. We mark them as *virtual*, i.e., these tensors are never explicitly stored but rather computed in small parts using a dynamic schedule. In the considered GNN models, this happens when obtaining Ψ: when evaluating it naively, one would end with a dense $n \times n$ matrix. As n can be very large (e.g., more than 10^9), keeping such a matrix is infeasible, even on a system with thousands of compute nodes. Instead, it is stored implicitly as an outer product of matrices with smaller dimensions. In the next subsection, we describe how perform computations on virtual tensors, without instantiating them.

6.2 Fusing Optimizations

We construct the forward and backward execution DAGs of the models (Figure 5). For each execution, we traverse the DAG until we find an edge (v_i, v_j) whose output v_j is a virtual matrix. Then, we continue to traverse the graph until we meet an edge (v_k, v_l) where v_l is a sparse intermediate result. In other words, the (v_k, v_l) edge's operation involves a sparse matrix that *samples* the virtual intermediate results in the path. We proceed by fusing all the operations in this path to generate an SDDMM-like kernel. We note that we may fuse more operations than necessary to avoid instantiations, if that will provide performance benefits.

The implementation of those operations is straightforward because the output almost always has the same sparsity pattern as the adjacency matrix. This in turn allows us to effortlessly integrate them into the distributed layer computations. The basic form of the kernels iterates over the non-zero values of the sparse matrix performing the sampling and computes the corresponding elements of the virtual dense matrix.

For GPU execution specifically, we implement each fused operation as a CUDA kernel. We employ grid-stride loops for coalesced memory and warp-level primitives to accelerate reductions.

6.3 Communication Optimizations

Our distribution strategy follows the state-of-the-art A-stationary 1.5D algorithm for SpMM [68]. We extend this approach so that the same 1.5 distribution covers all the operations in a single forward or backward executon layer. To achieve this, we apply a 2D distribution of the adjacency matrix on a 2D $P_x \times P_y$ cartesian process grid. On the other hand, the layer input $\boldsymbol{H}^{l}$ is distributed in P_y blocks, each replicated P_x times, while the output $\boldsymbol{H}^{l+1}$ is distributed in P_x blocks, each further split into P_y partial sums. To link the output of one layer with the input of the next one, we reduce the partial sums and then redistribute them in P_y blocks, also replicated P_x times. All the other layer inputs and intermediate results follow distributions that satisfy the above scheme. For example, the weight matrices $\mathbf{W}$ and vectors $\mathbf{a}$ are replicated across all processes.

6.4 Implementation Details

We use Python and we harness some of its performance-oriented packages: CuPy [66], NumPy [37], SciPy [79], and mpi4py [26]. The first three Python packages provide abstractions over basic linear algebra kernels, and automatically link against optimized implementations from Intel MKL for CPU and NVIDIA cuBLAS, cuSPARSE, and CUTLASS for GPU. We write our own CUDA implementation for tensor kernels and we integrate them into our Python workflow using CuPy.

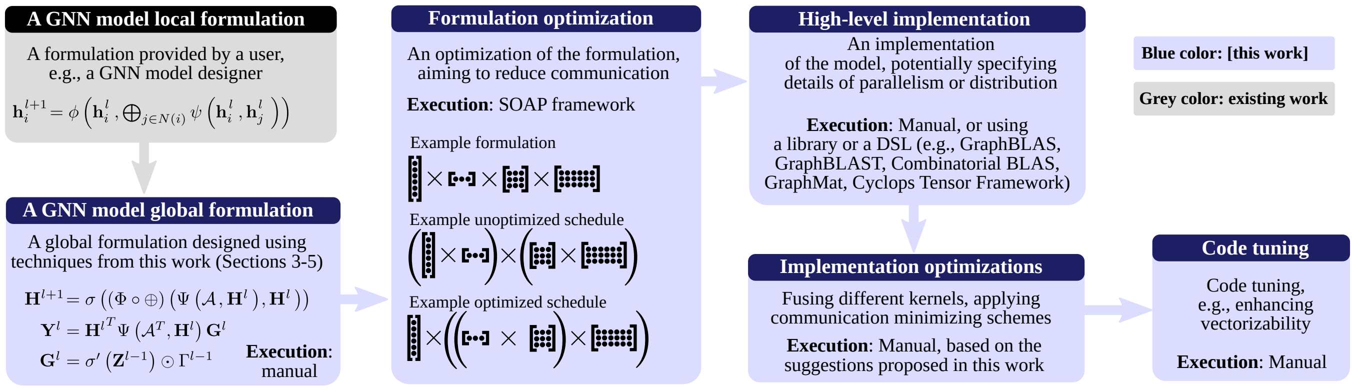

A GNN model local formulation
A formulation provided by a user, e.g., a GNN model designer
$\mathbf{h}_i^{l+1} = \phi\left(\mathbf{h}_i^{l}, \bigoplus_{j \in N(i)} \psi\left(\mathbf{h}_i^{l}, \mathbf{h}_j^{l}\right)\right)$
A GNN model global formulation
A global formulation designed using techniques from this work (Sections 3-5)
$\mathbf{H}^{l+1} = \sigma\left(\left(\Phi \circ \oplus\right)\left(\Psi\left(\mathcal{A}, \mathbf{H}^{l}\right), \mathbf{H}^{l}\right)\right)$
$\mathbf{Y}^{l} = \mathbf{H}^{l^T}\Psi\left(\mathcal{A}^T, \mathbf{H}^{l}\right)\mathbf{G}^{l}$
$\mathbf{G}^{l} = \sigma'\left(\mathbf{Z}^{l-1}\right) \odot \Gamma^{l-1}$
Execution: manual
Formulation optimization
An optimization of the formulation, aiming to reduce communication
Execution: SOAP framework
Example formulation
Example unoptimized schedule
Example optimized schedule
High-level implementation
An implementation of the model, potentially specifying details of parallelism or distribution
Execution: Manual, or using a library or a DSL (e.g., GraphBLAS, GraphBLAST, Combinatorial BLAS, GraphMat, Cyclops Tensor Framework)
Implementation optimizations
Fusing different kernels, applying communication minimizing schemes
Execution: Manual, based on the suggestions proposed in this work
Code tuning
Code tuning, e.g., enhancing vectorizability
Execution: Manual
Blue color: [this work]
Grey color: existing work

Figure 4: The toolchain provided in this work.

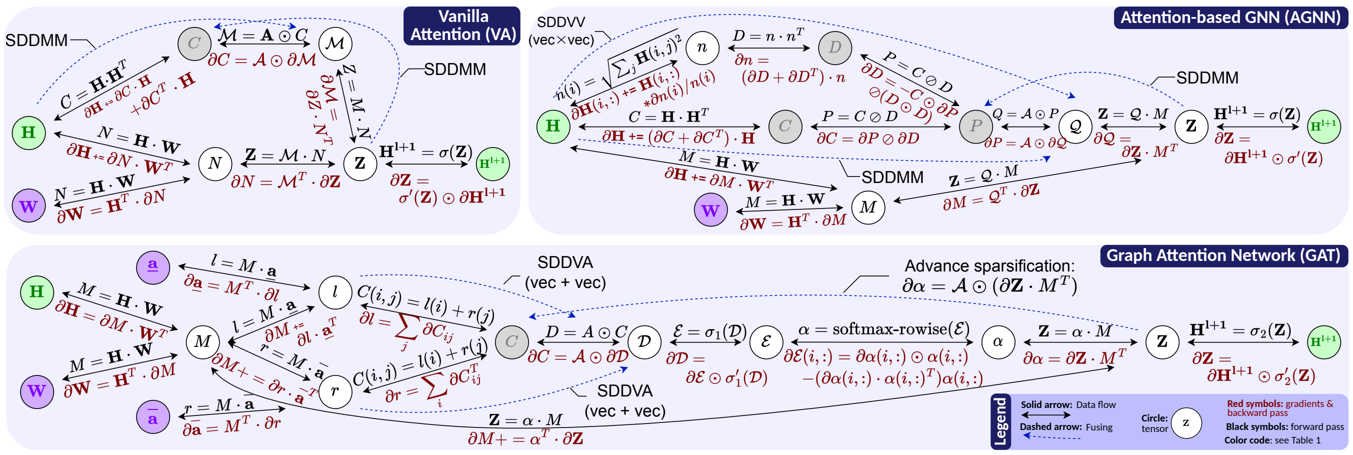

Vanilla Attention (VA)
Attention-based GNN (AGNN)
Graph Attention Network (GAT)
SDDMM
SDDVV (vec×vec)
SDDVA (vec + vec)
Advance sparsification: $\partial\alpha = \mathcal{A} \odot (\partial \mathbf{Z} \cdot M^T)$
Legend
Solid arrow: Data flow
Dashed arrow: Fusing
Circle: tensor
Red symbols: gradients & backward pass
Black symbols: forward pass
Color code: see Table 1

Figure 5: The analysis of the forward and backward execution DAGs for considered A-GNNs, and fusing optimizations (dashed arrows) proposed in this work.

7 THEORETICAL ANALYSIS

We summarize a communication cost analysis of the global and local formulations of GNNs in the Bulk-Synchronous Parallel (BSP) model [76]. We only provide the most important insights and analyses, all the details and proofs (including a full derivation of the results for the local formulation) are in the technical report. We use the BSP model, where – in each superstep – each of the p processors can send some data to the other processors, which receive the data in the next superstep. The maximum amount of words sent by any processor is the *communication volume*.

We show that, for computing one GNN layer of VA, AGNN and GAT, the global formulation takes $O(nk/\sqrt{p} + k^2)$ volume, whereas the local view takes up to $\Omega(nkd/p + k^2)$ volume, where n, k, d, p are – respectively – #vertices, #features, maximum degree, and #processes. We also show that these bounds not only hold for the forward pass, *but also for the backward pass (which is much more involved)*, as well as for inference. *Hence, for the entire GNN training process, **the global formulation is more communication-efficient than the local formulation**, for $d \in \omega(\sqrt{p})$*. This regime holds – for example – for graphs with a heavy-tail degree distribution, *which are omnipresent in modern workloads [13, 50, 54, 67].*

7.1 Communication Cost of Global Formulation

The general idea in analyzing the global view is to slice the sparse matrix $\mathcal{A}$ (and the resulting sparse matrix $\Psi(\mathcal{A}, \mathbf{H})$) into p blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, which are each assigned to a processor. For this, we give each processor a two-dimensional index (i, j) indicating which submatrix it stores. All communication is limited to sending slices of tall feature matrices and small parameter matrices and vectors. Moreover, we never materialize dense "virtual" matrices.

7.1.1 Computation of $\mathbf{\Phi}$ *and* $\mathbf{\Phi} \circ \oplus$. We first consider the feature update operation $\sigma(\Psi \mathbf{H}\mathbf{W})$, given that Ψ has already been calculated. Let $\mathbf{H}' = \mathbf{H}\mathbf{W}$ and $\mathbf{H}'' = \Psi\mathbf{H}'$ be intermediary matrices. First, we consider the product $\mathbf{H}' = \mathbf{H}\mathbf{W}$. We slice $\mathbf{H}$ row-wise into $\sqrt{p}$ blocks of equal size $(n/\sqrt{p}) \times k$, denoted as $H_0 \cdots H_{\sqrt{p}-1}$. Then, we can decompose this product into block operations $H_i' = H_i\mathbf{W}$, which we execute locally on $\sqrt{p}$ processors. Therefore, we first broadcast W to the first row of the processor grid denoted as $P_{1,1} \cdots P_{1,\sqrt{p}}$ and then distribute the blocks H_i to one of these processors each. Broadcasting W takes $O(\log p)$ steps and $O(k^2)$ communication volume. Distributing the blocks of $\mathbf{H}$ takes $O(1)$ steps and $O(nk/\sqrt{p})$ communication volume, since each processor can just send H_i to the correct $P_{1,i}$. Therefore, we have $O(\log p)$ supersteps and $O(nk/\sqrt{p}+k^2)$ communication volume overall.

Next, we turn to the product $\mathbf{H}'' = \Psi\mathbf{H}'$. Recall that since we already calculated $\mathbf{H}'$ as described above, we can assume that it is already distributed among $\sqrt{p}$ processors. As discussed before, each processor $P_{i,j}$ on the grid stores the sparse submatrix $\Psi_{i,j}$, which we do not want to move around. Therefore, it does not matter if Ψ is sparse or dense for our communication analysis. Next, note that we can rewrite our product into block operations $H_i'' = \sum_{j=0}^{\sqrt{p}-1} \Psi_{i,j} \times H_j'$, where the blocks H_i'' are of size $(n/\sqrt{p}) \times k$, such that we can operate on each $\Psi_{i,j}$ locally. Thus, we first replicate each H_j' by broadcasting it along the j-th column of the processor grid. Then, each processor $P_{i,j}$ can compute $\Psi_{i,j}H_j'$ locally. Finally, we reduce (sum) the resulting blocks along each row to calculate H_i'', which leaves us with $\mathbf{H}''$ distributed along $\sqrt{p}$ processors. The broadcast of all blocks H_j' takes $O(\log p)$ steps and $O(nk/\sqrt{p})$ communication volume, because we can broadcast along each column independently. Reducing along the rows incurs the same costs which gives a total of $O(\log p)$ supersteps and $O(nk/\sqrt{p})$ communication volume. The non-linearity σ is element-wise, requiring no additional communication. Hence, we conclude that the overall communication volume for computing $\sigma(\Psi\mathbf{H}\mathbf{W})$ is $O(nk/\sqrt{p}+k^2)$.

7.1.2 Computation of $\mathbf{\Psi}$, Υ, $\mathbf{\Gamma}$. We now summarize the analyses of Ψ, Υ, and $\mathbf{\Gamma}$; the details are in the technical report. Overall, we slice the computation of $\mathbf{\Psi}$ ($n \times n$ matrix) into p blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, and compute one block on each processor to derive bounds. $\mathcal{A}$ and $\mathbf{H}$ are sliced as in Section 7.1.1. $\mathbf{W}$ is fully replicated. This gives us a volume of $O(nk/\sqrt{p} + k^2)$. Second, the weight update operation $\mathbf{H}^T\Psi'(\sigma'(\mathbf{Z}) \odot \mathbf{\Gamma})$ only involves multiplications that are at most as expensive as the ones during the calculation of $\mathbf{\Psi}$. Thus, it also takes $O(nk/\sqrt{p}+k^2)$ communication volume to compute Υ, assuming that Γ has already been computed. Finally, we verify that calculating Γ, in each model, also has an $O(nk/\sqrt{p} + k^2)$ communication volume.

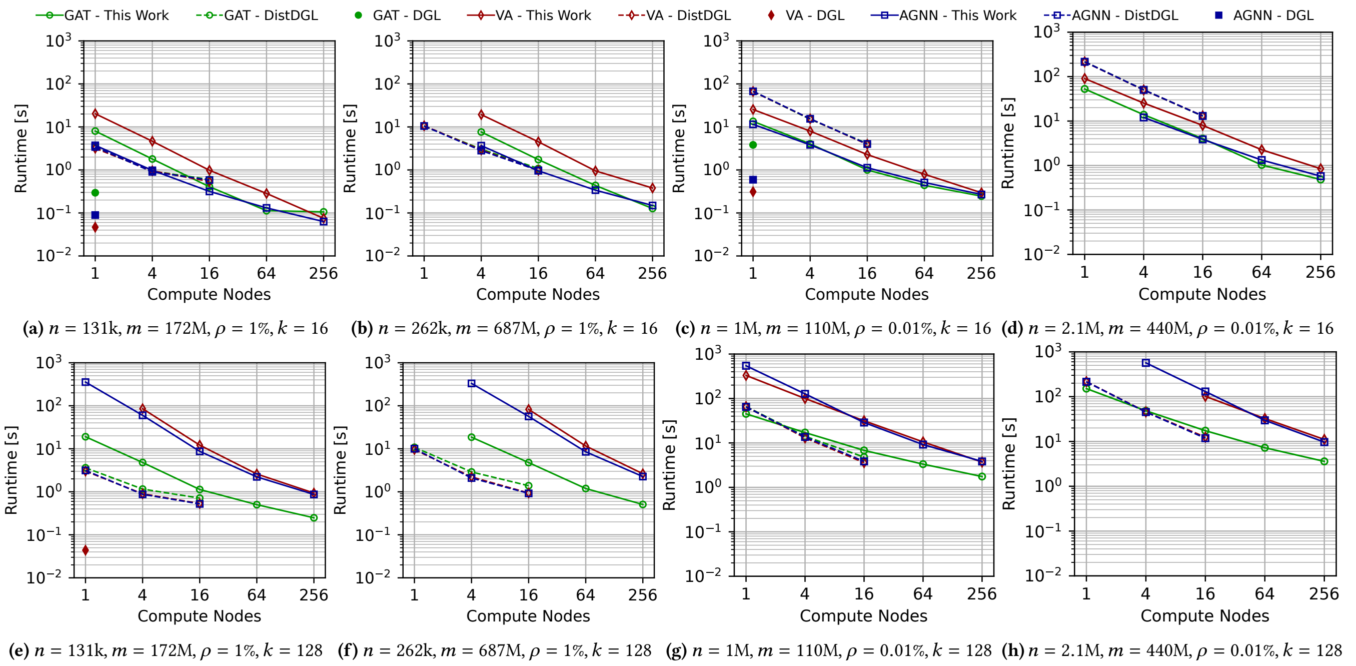


Figure 6: Strong scaling analysis of GNN training (scaling number of processes for fixed datasets). The used graphs are generated according to the heavy-tail Kronecker model. We vary the density of the adjacency matrix, defined as $\rho = m/n^2$, from 1% to 0.01%. The number of non-zeroes m ranges from ≈171 to ≈687 million. Missing data points indicate that a given baseline could not scale to a given node count, or that it did not fit in combined node memories.

7.2 Inference vs. Training

While we only need $\mathbf{\Psi}$ to do the inference pass, training also needs $\mathbf{\Gamma}$, which requires many operations (cf. Figure 1). On top of that, $\mathbf{\Psi}$ itself is required for backpropagation, making training strictly more computationally expensive than inference. *Nevertheless, we illustrate that, asymptotically, the communication needed for training is the same as for inference in the global formulation.*

7.3 Analysis for Erdős–Rényi Graphs

We also make our study more concrete and assume the Erdős–Rényi graph model $G_{n,q}$ with the random uniform degree distribution. Here, any edge exists with a constant probability q (independently of other edges) [30]. We show that the communication volume for the local formulation is in $O(n^2kq/p + \log n)$ with high probability. Thus, for denser graphs ($q > \sqrt{p}/n$), the global formulation is expected to be more efficient, while for sparser graphs we expect the difference between local and global formulation to decrease.

8 EVALUATION

We now illustrate performance advantages stemming from the proposed formulations and their underlying implementation.

8.1 Evaluation Setup

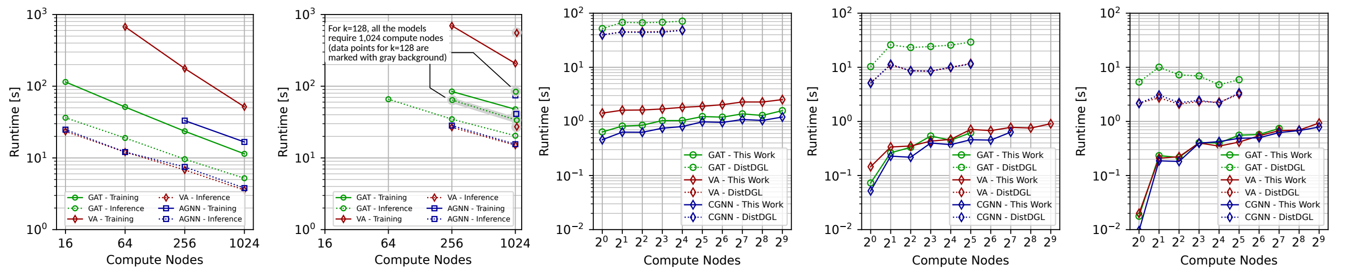
Comparison Baselines We analyzed the landscape of GNN frameworks to find the appropriate baselines. We discover that nearly all the systems for full-batch training target simple C-GNN models, which we do not focus on (e.g., CAGNET [75], PipeGCN [81], DistGNN [64], MG-GCN [5], DeepGalois [38], ROC [43], or Flex-Graph [82]). Dorylus is not compatible with our environment as it works with serverless cloud settings [74]. Finally, we identified dgNN, a very recent full-batch system that supports GAT; however, it is only single-node [94]. We also compare our work to distributed DGL (DistDGL) [83], a high-performance distributed-memory GNN library (to maximize fairness, for any scaling experiments that start with a single node, we additionally use the shared-memory DGL version [83]). While DistDGL uses mini-batch training, it serves as a valid comparison target because – as we will illustrate – our *full-batch* execution nearly always outperforms DistDGL's *mini-batch* execution, which processes *many orders or magnitude fewer vertices*. Here, we use the largest possible mini-batch size – 16k vertices – that did not cause DistDGL to crash due to OOM errors.

Parameters We vary the hidden feature dimension k to 16, 32, or 128, and experiment with #GNN layers $L \in \{2, ..., 10\}$. We measure the runtime of a complete full-batch forward pass with L layers (for inference) or a forward pass followed by backward pass, each with L layers (for training). All operations are executed in single-precision floating point arithmetic. We follow recommendations by the established GNN benchmarks [28, 41].

Architectures We run our experiments on the Piz Daint supercomputer at the CSCS supercomputing center. Each (Cray XC50) node has an Intel Xeon E5-2690 v3 @ 2.60GHz (12 cores, 64GB RAM) and an NVIDIA Tesla P100 with 16GB of memory. We note that our work and DGL utilize the GPU in all benchmarks. The interconnect architecture is Cray Aries [31], and the topology is Dragonfly [48].

Methodology For each benchmark, we measure the runtime of at least ten executions, and we plot the median and the 95% confidence interval using bootstrapping [29].

Datasets The used graphs are generated according to the Kronecker model, which ensures high load imbalance. We vary the density of the adjacency matrix, defined as $\rho = m/n^2$, from 1% to 0.01%. The number of non-zeroes m ranges from ≈171 to ≈686M. We also use the MS Academic Knowledge Graph (MAKG), a standard real-world dataset for large-scale GNNs, with 111M vertices and 3.2B edges.



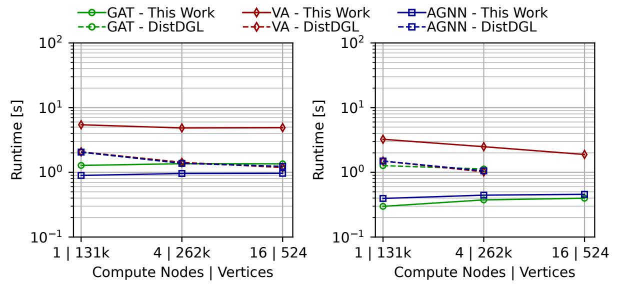
(a) MAKG, $k = 16$ **(b)** MAKG, $k = 64, 128$ **(c)** Rand, $\rho = 1\%$, $k = 128$ **(d)** Rand, $\rho = 0.1\%$, $k = 128$ **(e)** Rand, $\rho = 0.01\%$, $k = 128$

Figure 7: [The two leftmost plots: Strong scaling analysis for the MS Academic Knowledge Graph with 111M vertices and 3.2B edges; details in Section 8.3. **[The three rightmost plots]: Weak scaling for empirical verification of our communication cost analyses from Section 7;** details in Section 8.4; the graphs (Rand) used are generated according to the random uniform model. The first datapoint in each plot (i.e., the one for a single GPU / compute node) corresponds to a graph with $n \approx$ 131k, while the last datapoint (2^9 nodes) corresponds to $n \approx$ 3M (#vertices). We also vary the sparsity/density, to be between 1% and 0.01%, corresponding to m (#edges) being between ≈171 million and ≈90 billion edges. Missing data points indicate that a baseline did not scale to a given node count, or that it did not fit in combined node memories.

8.2 Performance Analysis

We first consider the strong scaling scenario. The results for different graph sparsities/densities are shown in Figure 6. The largest considered graphs have ≈3 million vertices and ≈90 billion edges.

Our implementations are in many cases faster than DistDGL when $k = 16$. For example, our AGNN and GAT implementations are more than 3–5× faster than DistDGL for graph density $\rho = 0.01\%$ in all considered node count scenarios. In addition, our VA implementation also outperforms DistDGL, reaching the speedups of 2–3× over DistDGL for $\rho = 0.01\%$. For $\rho = 1\%$, our AGNN implementation is comparable to DistDGL, while VA and GAT are slower (by up to >5×). Still, these results illustrate the difference between processing a single mini-batch of 16k vertices (DistDGL) vs. a full-batch of 131k or 262k vertices, and 172M or 687M edges. Furthermore, with the growing count of compute nodes (e.g., 16 for n = 131k, $k = 16$, the advantages of the global formulations become more prominent and they start to outperform DistDGL (note that, despite broad experimentation with configuration and parameters, we were unable to scale DistDGL to larger node counts due to OOM errors). This illustrates that the global formulation does scale better than the local formulation as implemented in DistDGL. At the same time, the communication latency is minimized due to, e.g., efficient use of collectives. The results for $k = 128$ indicate that the latency gap between the baselines increases. Here the GAT implementation has the best performance within the globally formulated models; it is comparable to DistDGL for $\rho = 0.01\%$, despite the fact that DistDGL still processes only one mini-batch.



(a) $\rho = 0.1\%, k = 16$ **(b)** $\rho = 0.01\%, k = 16$

Figure 8: Weak scaling analysis. The used graphs are generated according to the Kronecker model. We vary the density of the adjacency matrix, defined as $\rho = m/n^2$. We scale the number of vertices n proportionally to the square root of the node count. Keeping sparsity ρ fixed, the number of non-zeros m of the adjacency matrix scales proportionally to the number of compute nodes. Missing data points indicate that a baseline did not scale to a given node count, or that it did not fit in combined node memories.

We also analyze the weak scaling scenario: we scale the number of vertices n proportionally to the square root of the node count. Keeping sparsity ρ fixed, the number of non-zeros m of the adjacency matrix scales proportionally to the number of nodes. The results for different graph sparsity/density are shown in Figure 7 for random uniform graphs (the three rightmost plots) and in Figure 8 for Kronecker graphs. The largest considered graphs have ≈171 million vertices and ≈90 billion edges. Our implementations are faster than DistDGL, and they also exhibit excellent weak scaling. For example, our vanilla attention implementation retains up to 57% parallel efficiency on 512 nodes; the global formulations of other models come with similar advantages. The GAT implementation with 1% graph density has the communication overhead of 0.41s on 32 nodes in the global formulation. At the scale of 512 nodes, our implementation's communication cost for the same benchmark is only 1.13s. The achieved speedup against the state-of-the-art shows the large potential of our approach, as the global view of the GNN allows us to accelerate computation and communication. Each node's workload, expressed as a linear algebra equation, is processed efficiently, optimizing shared-memory performance. At the same time, the communication latency is minimized due to message aggregation and efficient use of collectives.

Finally, we were only able to run the single-node frameworks (DGL [83], dgNN [94]) for a few scenarios, in almost all the graphs they crashed due to OOM errors. Thus, while they were always much faster than our implementation and also than DistDGL, they are unable to process large datasets targeted in this work.

8.3 Processing Large Real-World Graphs

We also successfully scale both training and inference of the considered models on the large MAKG using our global formulations (DistDGL experienced regular OOM crashes). We show the strong scaling results in Figure 7, both for inference and for training. When using 16 features, we are able to successfully process MAKG even on 16 compute nodes for most models (the models that need more than 16 nodes are VA and AGNN). All the models exhibit excellent scaling characteristics. For $k = 64$, only GAT can be processed with 64 nodes, all other models require 256 nodes at the minimum. When using 128 features, we can process MAKG using VA and AGNN only on 1,024 nodes; GAT – which puts less pressure on the memory – can also be executed on 256 nodes. This illustrates that our global formulation fosters scaling to very large node and core counts, successfully processing the largest GNN datasets available; even for 1,024 nodes, the communication does not become the bottleneck.

8.4 Verification of Theoretical Predictions

(aka. Global vs. Local Formulation). We also analyze our predictions for the communication volume from Section 7, i.e., the global formulation has lower communication volume if the maximum degree $d \in \omega(\sqrt{p})$. For this, we use Erdős-Rényi graphs with a random uniform degree distribution (Rand). As we determined in Section 7.3, denser Rand graphs should come with more advantages in performance for the global formulation compared to the local one. In this setting, we compare our global formulation to the local formulation (represented by DistDGL) in Figure 7 (the three leftmost plots), for three densities ρ (1%, 0.1%, and 0.01%). We consider the weak scaling scenario and the inference pass; we scale the number of vertices n proportionally to the square root of the node count. Keeping sparsity fixed, the number of non-zeros m of the adjacency matrix scales proportionally to the number of nodes. The largest considered graphs have ≈3 million vertices and ≈90 billion edges. Here, DistDGL is significantly slower than our work, which is caused by excellent load balancing properties of the used random uniform matrices, combined with the fact that the inference in the global formulation only involves a single SDDMM, SpMM, and MM product (in one GNN layer), with one of the dense matrices being also small and fully replicated on each process. Yet, the more important observation is that – with the decreasing density ρ – as expected in our analysis for Erdős-Rényi graphs – the difference between DistDGL and our work consistently decreases.

To further investigate our predictions, we also consider here an example C-GNN (the simple graph convolution model). This model can be seen as a special case of an A-GNN, with a single GNN inference later consisting of one SpMM and one MM. Thus, the communication volume bounds derived in Section 7 also hold for this model. Similarly, the difference between the global and the local approach decreases with the increasing ρ also for a C-GNN, when Erdős-Rényi graphs are used.

9 RELATED WORK

Irregular computations with linear algebra building blocks A lot of work has been dedicated to expressing graph algorithms with building blocks using matrices, vectors, and operations on them, such as sparse-dense products. This includes BFS [6, 11, 17, 19], Betweenness Centrality [69], Jaccard similarity and clustering [10], and many others [1–3, 12, 34, 60, 90, 96]. Recently, efforts were made to apply this approach to C-GNNs [8, 75, 83]. We extend this line of work by offering novel GNN formulations based on tensor algebra, targeting A-GNNs. Our generic GNN formulation can be used as a blueprint for future GNN models.

GraphBLAS is a standard for formulating graph algorithms using linear algebra building blocks [46, 47]. Our global formulations could easily be used with GraphBLAS implementations such as Combinatorial BLAS [18], GraphMat [72], or GraphBLAST [91].

GNN Frameworks & Accelerators A large number of frameworks and accelerators for GNNs have recently been developed [23, 32, 42, 55, 75, 82, 84, 86, 89, 99]. Our formulations could be used together with many of these systems for more performance.

The programming model of **DGL** offers two fundamental building blocks, a generalized SDDMM (g-SDDMM) and a generalized SpMM (g-SpMM). Yet, DGL does *not* provide global formulations based on g-SDDMM and g-SpMM for A-GNN models such as GAT (instead, the g-SDDMM and g-SpMM building blocks are implemented using the local formulation when targeting many A-GNNs). Here, our Ψ kernels in GAT or AGNN could be seamlessly incorporated as the g-SDDMM routines to provide matrix-based implementations of GAT and AGNN within DGL.

10 CONCLUSION

Attentional GNNs (A-GNNs) form a very powerful class of methods in graph ML. Yet, they are very challenging to scale due to their unique and complex design.

In this work, we enhance the performance and scalability of many A-GNNs. We achieve this by designing a generic "global" mathematical formulation based on tensors grouping all feature vectors, weights, and the adjacency graph structure. We provide specialized formulations for both the forward and backward propagation passes of several models, such as the Graph Attention Network or the Attention-based GNN. Our formulations are entirely independent of the underlying implementation and can be seamlessly used with existing frameworks based on tensor algebra, for example, Combinatorial BLAS. Moreover, while we focus on full-batch training of A-GNNs, one can straightforwardly extend most of our routines to mini-batching and models outside the A-GNN family, such as Graph Networks.

We implement the devised formulations with communication-minimizing routines that harness several optimizations based on kernel fusion. Our design comes with both theoretical advantages over the competition, for example, in the amount of communicated data, and empirical speedups such as more than 5× over DistDGL for different scenarios. Overall, our work deepens the understanding of A-GNNs, as it illustrates that – despite their complex formulations – their execution can be reduced to only a few tensor kernels, and it can harness tuned existing routines such as SpMM or SDDMM.

Acknowledgements

We thank Hussein Harake, Colin McMurtrie, Mark Klein, Angelo Mangili, and the whole CSCS team granting access to the Ault and Daint machines, and for their excellent technical support. We thank Timo Schneider for help with computing infrastructure at SPCL. This project received funding from the European Research Council (Project PSAP, No. 101002047), and the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 955513 (MAELSTROM). This project was supported by the ETH Future Computing Laboratory (EFCL), financed by a donation from Huawei Technologies. This project received funding from the European Union's HE research and innovation programme under the grant agreement No. 101070141 (Project GLACIATION). This work was also supported in part by the European Union's Horizon 2020 research and innovation programme under the grant: Sano No. 857533, the International Research Agendas programme of the Foundation for Polish Science, co-financed by the EU under the European Regional Development Fund, and the Swiss National Science Foundation (Project QuaTrEx, No. 209358).

References

[1] Ariful Azad, Aydın Buluç, and John Gilbert. 2015. Parallel Triangle Counting and Enumeration Using Matrix Algebra. In *Proceedings of the International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*. IEEE, 804–811.

[2] Ariful Azad, Aydın Buluç, Xiaoye S. Li, Xinliang Wang, and Johannes Langguth. 2020. A Distributed-Memory Algorithm for Computing a Heavy-Weight Perfect Matching on Bipartite Graphs. *SIAM Journal on Scientific Computing* 42, 4 (2020), C143–C168.

[3] Ariful Azad, Mathias Jacquelin, Aydın Buluç, and Esmond G Ng. 2017. The Reverse Cuthill-McKee Algorithm in Distributed-Memory. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, 22–31.

[4] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John R Gilbert, and Aydın Buluç. 2022. Combinatorial BLAS 2.0: Scaling Combinatorial Algorithms on Distributed-Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 989–1001.

[5] Muhammed Fatih Balın, Kaan Sancak, and Ümit V Çatalyürek. 2021. MG-GCN: Scalable Multi-GPU GCN Training Framework. arXiv:2110.08688

[6] Scott Beamer, Aydın Buluç, Krste Asanovic, and David Patterson. 2013. Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search. In *Proceedings of the International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW '13)*. IEEE, 1618–1627.

[7] Maciej Besta, Raphael Grob, Cesare Miglioli, Nicola Bernold, Grzegorz Kwasniewski, Gabriel Gjini, Raghavendra Kanakagiri, Saleh Ashkboos, Lukas Gianinazzi, Nikoli Dryden, and Torsten Hoefler. 2022. Motif Prediction with Graph Neural Networks. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*. 35–45.

[8] Maciej Besta and Torsten Hoefler. 2022. Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis. arXiv:2205.09702

[9] Maciej Besta, Patrick Iff, Florian Scheidl, Kazuki Osawa, Nikoli Dryden, Michał Podstawski, Tiancheng Chen, and Torsten Hoefler. 2022. Neural Graph Databases. In *Proceedings of the First Learning on Graphs Conference*. Proceedings of Machine Learning Research, Vol. 198. PMLR, 31:1–31:38.

[10] Maciej Besta, Raghavendra Kanakagiri, Harun Mustafa, Mikhail Karasikov, Gunnar Rätsch, Torsten Hoefler, and Edgar Solomonik. 2019. Communication-Efficient Jaccard Similarity for High-Performance Distributed Genome Comparisons. arXiv:1911.04200

[11] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '17)*. IEEE, 32–41.

[12] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '17)*. ACM, 93–104.

[13] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. ACM, 595–602.

[14] Michael M Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. arXiv:2104.13478

[15] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. 2017. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine* 34, 4 (2017), 18–42.

[16] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.). Vol. 33. 1877–1901.

[17] Aydın Buluç, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David Patterson. 2017. Distributed-Memory Breadth-First Search on Massive Graphs. arXiv:1705.04590

[18] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 25, 4 (Nov 2011), 496–509.

[19] Aydın Buluç and Kamesh Madduri. 2011. Parallel Breadth-First Search on Distributed Memory Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*. ACM, Article 65, 12 pages.

[20] Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. 2021. Combinatorial Optimization and Reasoning with Graph Neural Networks. arXiv:2102.09544

[21] Deepayan Chakrabarti and Christos Faloutsos. 2006. Graph Mining: Laws, Generators, and Algorithms. *ACM Comput. Surv.* 38, 1 (June 2006), 69 pages.

[22] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2020. Machine Learning on Graphs: A Model and Comprehensive Taxonomy. arXiv:2005.03675

[23] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020. FuseGNN: Accelerating Graph Convolutional Neural Network Training on GPGPU. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD '20)*. ACM, Article 60, 9 pages.

[24] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (Aug 2015), 1804–1815.

[25] Diane J Cook and Lawrence B Holder (Eds.). 2006. *Mining Graph Data*. John Wiley & Sons.

[26] Lisandro Dalcin and Yao-Lung L. Fang. 2021. mpi4py: Status Update After 12 Years of Development. *Computing in Science & Engineering* 23, 4 (2021), 47–54.

[27] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Eds.). Vol. 29. 3844–3852.

[28] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2020. Benchmarking Graph Neural Networks. arXiv:2003.00982

[29] Bradley Efron. 2000. The Bootstrap and Modern Statistics. *J. Amer. Statist. Assoc.* 95, 452 (2000), 1293–1296.

[30] Paul Erdös and Alfréd Rényi. 1961. On the Evolution of Random Graphs. *Bull. Inst. Internat. Statist* 38, 4 (1961), 343–347.

[31] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. 2012. Cray Cascade: A Scalable HPC System Based on a Dragonfly Network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE, Article 103, 9 pages.

[32] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. arXiv:1903.02428

[33] Ziqi Gao, Chenran Jiang, Jiawen Zhang, Xiaosen Jiang, Lanqing Li, Peilin Zhao, Huanming Yang, Yong Huang, and Jia Li. 2023. Hierarchical graph learning for protein–protein interaction. *Nature Communications* 14, 1093 (2023).

[34] Evangelos Georganas, Rob Egan, Steven Hofmeyr, Eugene Goltsman, Bill Arndt, Andrew Tritt, Aydin Buluç, Leonid Oliker, and Katherine Yelick. 2019. Extreme Scale de Novo Metagenome Assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. IEEE, Article 10, 13 pages.

[35] Lukas Gianinazzi, Maximilian Fries, Nikoli Dryden, Tal Ben-Nun, Maciej Besta, and Torsten Hoefler. 2021. Learning Combinatorial Node Labeling Algorithms. arXiv:2106.03594

[36] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. arXiv:1709.05584

[37] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[38] Loc Hoang, Xuhao Chen, Hochan Lee, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2021. Efficient Distribution for Deep Learning on Large Graphs. In *Proceedings of the Workshop on Graph Neural Networks and Systems (GNNSys '21)*.

[39] Tamás Horváth, Thomas Gärtner, and Stefan Wrobel. 2004. Cyclic Pattern Kernels for Predictive Graph Mining. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '04)*. 158–167.

[40] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. arXiv:2103.09430

[41] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. arXiv:2005.00687

[42] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE, Article 71, 13 pages.

[43] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems (MLSys '20, Vol. 2)*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.). 187–198.

[44] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A Survey of Frequent Subgraph Mining Algorithms. *The Knowledge Engineering Review* 28, 1 (2013),

75–105.

[45] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature* 596, 7873 (2021), 583–589.

[46] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. 2016. Mathematical Foundations of the GraphBLAS. In *High Performance Extreme Computing Conference (HPEC '16)*. IEEE, 1–9.

[47] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. *Procedia Computer Science* 51 (2015), 2453–2462.

[48] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE, 77–88.

[49] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the International Conference on Learning Representations (ICLR '17)*.

[50] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, 1343–1350.

[51] Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler. 2021. Pebbles, Graphs, and a Pinch of Combinatorics: Towards Tight I/O Lower Bounds for Statically Analyzable Programs. In *Proceedings of the 33rd Symposium on Parallelism in Algorithms and Architectures (SPAA '21)*. ACM, 328–339.

[52] Grzegorz Kwasniewski, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Timo Schneider, Maciej Besta, and Torsten Hoefler. 2021. On the Parallel I/O Optimality of Linear Algebra Kernels: Near-Optimal LU Factorization. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21)*. 463–464.

[53] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. ACM, Article 24, 22 pages.

[54] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[55] Chris Lin, Gerald J Sun, Krishna C Bulusu, Jonathan R Dry, and Marylens Hernandez. 2020. Graph Neural Networks Including Sparse Interpretability. arXiv:2007.00119

[56] Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. 2022. A Comprehensive Survey on Distributed Training of Graph Neural Networks. arXiv:2211.05368

[57] Heng Lin, Xiaowei Zhu, Bowen Yu, Xiongchao Tang, Wei Xue, Wenguang Chen, Lufei Zhang, Torsten Hoefler, Xiaosong Ma, Xin Liu, Weimin Zheng, and Jingfang Xu. 2018. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '18)*. IEEE, Article 56, 11 pages.

[58] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th Symposium on Cloud Computing (SoCC '20)*. ACM, 401–415.

[59] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. 2021. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. arXiv:2112.08541

[60] Adam Lugowski, David Alber, Aydın Buluç, John R. Gilbert, Steve Reinhardt, Yun Teng, and Andrew Waranis. 2012. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*. 930–941.

[61] Anjun Ma, Xiaoying Wang, Jingxian Li, Cankun Wang, Tong Xiao, Yuntao Liu, Hao Cheng, Juexin Wang, Yang Li, Yuzhou Chang, Jinpu Li, Duolin Wang, Yuexu Jiang, Li Su, Gang Xin, Shaopeng Gu, Zihai Li, Bingqiang Liu, Dong Xu, and Qin Ma. 2023. Single-cell biological network inference using a heterogeneous graph transformer. *Nature Communications* 14, 964 (2023).

[62] Diane Maclagan and Bernd Sturmfels. 2021. *Introduction to Tropical Geometry*. Graduate Studies in Mathematics, Vol. 161. American Mathematical Society.

[63] Petros Maragos, Vasileios Charisopoulos, and Emmanouil Theodosis. 2021. Tropical Geometry and Machine Learning. *Proc. IEEE* 109, 5 (2021), 728–755.

[64] Vasimuddin Md, Sanchit Misra, Guixiang Ma, Ramanarayan Mohanty, Evangelos Georganas, Alexander Heinecke, Dhiraj Kalamkar, Nesreen K Ahmed, and Sasikanth Avancha. 2021. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. arXiv:2104.06700

[65] Grigory Mikhalkin. 2006. Tropical Geometry and its applications. arXiv:math/0601041

[66] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. *Proceedings of the Workshop on ML Systems*.

[67] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. *Proceedings of the AAAI Conference on Artificial Intelligence* 29, 1 (Mar 2015).

[68] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2021. Distributed-Memory Parallel Algorithms for Sparse Times Tall-Skinny-Dense Matrix Multiplication. In *Proceedings of the International Conference on Supercomputing (ICS '21)*. ACM, 431–442.

[69] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. 2017. Scaling Betweenness Centrality Using Communication-Efficient Sparse Matrix Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, Article 47, 14 pages.

[70] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2017. Trade-Offs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations. *ACM Trans. Parallel Comput.* 3, 1, Article 3 (Jan 2017), 47 pages.

[71] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops Tensor Framework: Reducing Communication and Eliminating Load Imbalance in Massively Parallel Contractions. In *Proceedings of the 27th International Symposium on Parallel and Distributed Processing (IPDPS '13)*. IEEE, 813–824.

[72] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225.

[73] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based Graph Neural Network for Semi-supervised Learning. arXiv:1803.03735

[74] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. 495–514.

[75] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing Communication in Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE, Article 70, 17 pages.

[76] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Commun. ACM* 33, 8 (Aug 1990), 103–111.

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Vol. 30. 5998–6008.

[78] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *Proceedings of the International Conference on Learning Representations (ICLR '18)*.

[79] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, and Paul van Mulbregt. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (2020), 261–272.

[80] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2022. Marius++: Large-Scale Training of Graph Neural Networks on a Single Machine. arXiv:2202.02365

[81] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. arXiv:2203.10428

[82] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. ACM, 67–82.

[83] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang,

Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. arXiv:1909.01315

[84] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Efficient Runtime System for GNN Acceleration on GPUs. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI '21)*. 515–531.

[85] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning*. Proceedings of Machine Learning Research, Vol. 97. PMLR, 6861–6871.

[86] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: Vertex-Centric Programming for Graph Neural Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. ACM, 359–375.

[87] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.

[88] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? arXiv:1810.00826

[89] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA '20)*. IEEE, 15–29.

[90] Carl Yang, Aydın Buluç, and John D. Owens. 2018. Implementing Push-Pull Efficiently in GraphBLAS. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP '18)*. ACM, Article 89, 11 pages.

[91] Carl Yang, Aydın Buluç, and John D. Owens. 2022. GraphBLAST: A High-Performance Linear Algebra-Based Graph Framework on the GPU. *ACM Trans. Math. Softw.* 48, 1, Article 1 (Feb 2022), 51 pages.

[92] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. arXiv:2003.02454

[93] Haicang Zhang, Michelle S Xu, Xiao Fan, Wendy K Chung, and Yufeng Shen. 2022. Predicting functional effect of missense variants using graph attention neural networks. *Nature Machine Intelligence* 4 (2022), 1017–1028.

[94] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2022. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. In *Proceedings of Machine Learning and Systems (MLSys '22, Vol. 4)*, D. Marculescu, Y. Chi, and C. Wu (Eds.). 467–484.

[95] Liwen Zhang, Gregory Naitzat, and Lek-Heng Lim. 2018. Tropical Geometry of Deep Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning*. Proceedings of Machine Learning Research, Vol. 80. PMLR, 5824–5832.

[96] Yongzhe Zhang, Ariful Azad, and Aydın Buluç. 2020. Parallel Algorithms for Finding Connected Components using Linear Algebra. *J. Parallel and Distrib. Comput.* 144 (2020), 14–27.

[97] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2022. Deep Learning on Graphs: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 249–270.

[98] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *10th Workshop on Irregular Applications: Architectures and Algorithms (IA3 '20)*. ACM/IEEE, 36–44.

[99] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, Qidong Su, Minjie Wang, Chao Ma, and George Karypis. 2021. Distributed Hybrid CPU and GPU training for Graph Neural Networks on Billion-Scale Graphs. arXiv:2112.15345

[100] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.

[101] Alexandros Nikolaos Ziogas, Grzegorz Kwasniewski, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. 2022. Deinsum: Practically I/O Optimal Multi-Linear Algebra. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '22)*. IEEE, Article 25, 15 pages.

Appendix: Artifact Description/Artifact Evaluation

ARTIFACT DOI

10.5281/zenodo.8081983

ARTIFACT IDENTIFICATION

We develop a novel mathematical formulation of graph attention models based on tensors that group all the feature vectors for training and inference. The formulation enables straightforward adoption of communication-minimizing routines, it fosters optimizations such as vectorization, and it enables seamless integration with established linear algebra DSLs or libraries such as GraphBLAS. Our implementation uses a data redistribution scheme explicitly developed for sparse-dense tensor operations used heavily in GNNs, and fusing optimizations that further minimize memory usage and communication cost. We developed a framework in Python using CuPy, NumPy and SciPy to implement the tensor kernels in CUDA and use mpi4py to distribute the adjacency matrix and redistribute parts of the tensor after each neural GNN layer as well as performing aggregation during the layer computation.

We ensure theoretical asymptotic reductions in communicated data compared to the established message-passing GNN paradigm. Finally, we provide excellent scalability and speedups of even 4–5x over modern libraries such as Deep Graph Library (DGL). The framework is extensively benchmarked to show that these theoretical reductions translate to performance gains in practice as well as enabling comparison to other baselines and systems.

The computational artifacts of the submission are divided into **(A)** source code, **(B)** datasets, and **(C)** benchmark data.

(A) Source Code

The source code covers three areas: **A0** GNN models, which implement inference and training. **A1** Benchmark code to measure the runtime of the GNN models. **A2** Plot scripts to generate the figures.

The code artifacts **A0** and **A1** can be found in the `src` directory, where as the code artifact **A2** is situated in the directory `plots`.

(B) Datasets

We use three types of datasets for the adjacency matrix: Kronecker graphs **(B0)** are generated in a distributed way in main memory at the beginning of the experiment based on the Graph500 Kronecker generator. The graph is further processed by removing duplicate edges and by ensuring that each vertex is connected to at least one other vertex. It is also possible to load the adjacency matrix from a file in the COO format stored in the compressed numpy (.npz) file format. The real-world graph used in the evaluation is based on the Microsoft Academic Knowledge Graph **(B1)** and was provided by Open Graph Benchmark[1]. It contains 111 million vertices and 3.2 billion edges. Erdős–Rényi graphs are used for some of the weak scaling experiments **(B2)**. These graph are generated similar to the Kronecker graphs distributed in the main memory at the beginning of the experiment.

[1]https://ogb.stanford.edu/docs/nodeprop/#ogbn-papers100M

(C) Benchmark Data

The experimental data from benchmarking the runtime of the GNN models, from which the figures in the submission originate, is located in the `plots/results` directory.

REPRODUCIBILITY OF EXPERIMENTS

Experimental Workflow

The code written for the submission is a framework based on Python **(A0)** and can be found in the `src` directory. There are three base classes in the file `gnn_models.py`: GnnLayer, GnnModel and Loss. The forward and backward methods are overloaded for each model (vanilla attention (VA), AGNN, and GAT), which allow caching of intermediate results for training. The redistribution methods of the GnnModel class are further overloaded for the distributed code to reshuffle the output data of the previous layer as input for the next layer using MPI collectives. There are two files for each model:

- `{model-name}_model.py`, which contains the GPU implementation for a single compute node and a CPU implementation as reference for validation.
- `{model-name}_model_distr.py` consists of the distributed GPU implementation.

Both files also accommodate code for validation with the reference implementation.

There are two benchmark scripts for the GNN models developed for the submission **(A1)**: `unified_single_bench.py` for benchmarking on a single compute node (without the use of MPI) and `unified_distr_bench.py` for benchmarking in a distributed environment. Each execution of the benchmark scripts performs an experiment for a single specific configuration: model, task, #vertices, #edges, #features.

```
% python3 unified_single_bench.py -m VA -v 10000
  -e 1000000
```

is an example for a running a small benchmark on a single compute node. Similarly one can benchmark the same workload in a distributed environment by executing

```
% {process launcher} -n 4 python3 unified_distr_bench.py
  -m VA -v 10000 -e 1000000
```

where the process launcher could be `mpirun` or `srun` (on SLURM systems). The task execution is repeated a number of times with additional warmup runs, the time of each task execution is measured and the results (median and standard deviation) are appended to a CSV file. The configuration of both scripts can be controlled with the help of command line parameters:

```
usage: unified_distr_bench.py [-h] [-d [{random,file,kronecker}]]
  [-s [SEED]] [-v [VERTICES]] [-e [EDGES]] [-t [{float32,float64}]]
  [-m [{VA,GAT,AGNN}]] [-f [FILE]] [--features [FEATURES]]
  [--inference] [-l [LAYERS]] [--repeat [REPEAT]] [--warmup [WARMUP]]

optional arguments:
  -h, --help            show this help message and exit
  -d [{random,file,kronecker}], --dataset [{random,file,kronecker}]
                        The source of the adjacency matrix.
  -s [SEED], --seed [SEED]
```

```
                        The seed for the random number generator.
  -v [VERTICES], --vertices [VERTICES]
                        The number of vertices in the graph.
  -e [EDGES], --edges [EDGES]
                        The number of edges in the graph.
  -t [{float32,float64}], --type [{float32,float64}]
                        The type of the data.
  -m [{VA,GAT,AGNN}], --model [{VA,GAT,AGNN}]
                        The model to test. [VA, GAT, AGNN]
  -f [FILE], --file [FILE]
                        The file containing the adjacency matrix.
  --features [FEATURES]
                        The number of features.
  --inference  Run inference only (not storing intermediate matrices).
  -l [LAYERS], --layers [LAYERS]
                        The number of layers in the GNN model.
  --repeat [REPEAT]     The number of times to repeat the benchmark.
  --warmup [WARMUP]     The number of warmup runs.
```

While weights and inputs are generated randomly, the user can specify the origin of the adjacency matrix with the help of the `-d/-dataset` command line parameter. When loading the adjacency matrix from a file in the COO format stored in the compressed numpy (`.npz`) file format, the user has to specify the path to the numpy file with `-f/-file`. In such a case, the number of vertices and edges are read directly from the file, so the respective command line parameters have no significance. It is also possible to generate the adjacency matrix during the runtime: random uniform degree distribution (`-d uniform`) or Kronecker graphs (`-d kronecker`). `-v/-vertices` specifies the number of vertices in the graph, which is identical to the dimensions of the adjacency matrix. `-e/-edges` specifies the number of edges in the graph, which allows the user to influence how sparse the graph is.

The model to benchmark can be chosen with the `-m/-model` command line parameter: VA, GAT, AGNN. With `-features` the user can specify the number of features and with `-l/-layers` the numbers. The figures in the submission show plots for benchmarks with three neural GNN layers. `-repeat` adjusts the number of times the benchmark is repeated, where as `-warmup` states the number of repetitions before the individual runs are measured. We repeated the benchmark ten times with two additional warmup executions

`-s/-seed` sets the seed for the random number generator for the random uniform degree distribution. We used the default seed in our experiments. `-t/-type` adjusts the type of the data: we choose `float32`. `-inference` lets the model run inference only without storing intermediate matrices for the backward pass.

The current Kronecker graph creation process can only generate graphs where the number of vertices is a power of two. If the user specifies a number of vertices that is not, the program will round down to the nearest number that is a power of two.

We provide two additional bash scripts (`unified_strong.sh` and `unified_weak.sh`) to generate the necessary batch job scripts for the strong and weak scaling workloads, where the command line parameters are already set to the values from the submission **(A1)**. The scripts also enqueue the jobs into the batch system. Slurm scripts for Piz Daint are used as an example and should be adapted for the target cluster accordingly. The results can be found in `unified_results.csv`. We provide an empty result file in the `src` directory, which is already initialized with the header required for the plotting script.

Once all batch jobs are finished, the Python script `create_plots_artifact.py` **(A2)** can be called to create plots similar to the ones in the submission. The Python scripts expects the experimental results in the file `results/unified_results.csv`.

As a baseline, we use DGL, a state-of-the-art framework for GNNs. DGL employs mini-batch training, which processes many orders of magnitude fewer vertices than the full-batch training used by our work. However, in the absence of other distributed full-batch training frameworks for attention GNNs at the time of submission, DGL provides a valid HPC-oriented comparison. We use DistDGL, the distributed version of DGL. For single-node execution, we compare our work with the original shared-memory DGL, which performs better (to maximize fairness).

Estimation of Execution Time

The benchmarking of the weak scaling workload took a little bit over 90 minutes on Piz Daint and 140 minutes for the strong scaling workload, when benchmarking 10 executions of each configuration.

Expected Results and Relationship to the Results in the Submission

The expected results from the artifact should exactly match those provided in the submission, assuming using the same SW and HW configuration. Thus, we now describe the detailed configuration needed for generating these results. Using different HW would provide results with the same performance trends (i.e., with similar relative differences between respective baselines). We first provide the parametrization, referring directly to the results in the submission (the results obtained from the artifact match these results).

Each configuration should be executed at least 10 times.

Strong scaling experiments with all three models (VA, AGNN, GAT) using Kronecker graphs **(B0)**:

Figures 6 (a)/(e) number of vertices = 131072, number of edges = 171798691, number of features = 16/128.
Figures 6 (b)/(f) number of vertices = 262144, number of edges = 687194767, number of features = 16/128.
Figures 6 (c)/(g) number of vertices = 1048576, number of edges = 109951162, number of featurs = 16/128.
Figures 6 (d)/(h) number of vertices = 2097152, number of edges = 439804651, number of features = 16/128.

Weak scaling experiment with three models (VA, AGNN, GAT) using Kronecker graphs **(B0)**:

Figure 7(a) number of compute nodes = 1/4/16, number of vertices = 131072/262144/524288, number of edges = 17179869/68719476/274877906, number of features = 16.
Figure 7(b) number of compute nodes = 1/4/16, number of vertices = 131072/262144/524288, number of edges = 1717986/6871947/27487790, number of features = 16.

Figure 8's (a) and (b) subplots present strong scaling analysis for the MS Academic Knowledge Graph **(B1)** with 111M vertices and 3.2B edges. The (c), (d), and (e) subplots show weak scaling results on uniformly random graphs **(B2)** with a number of vertices between 131072 and 2966016. The graphs' density also changes; 1% in the (c) subplot, 0.1% in the (d) subplot, and 0.01% in the (e) subplot.

The results for the above figures, and the figures themselves, can be generated using the described and provided reproducibility-focused infrastructure described in the Experimental Workflow paragraphs. We ensure that all the scripts and the general workflow are easy to follow and use.

ARTIFACT DEPENDENCIES REQUIREMENTS

The implementation of the attention models is based on CuPy, so CUDA-enabled NVIDIA GPUs are required to execute the respective Python scripts.

The Python scripts of the attention model implementations don't depend on a specific operating system. However, when the adjacency matrix is generated from a Kronecker graph, a shared library written in C is used. The code base of the shared library was written for Linux-based environments. If the adjacency matrix is derived from other input methods (random uniform, file loading), any operating system that supports Python can be used.

The Python scripts depend on the following Python packages:

- `argparse`
- `cupy`
- `matplotlib`
- `mpi4py`
- `numpy`
- `pandas`
- `scipy`
- `typing`

The shared library written in C for the generation of Kronecker graphs is based on Kronecker module from the Graph500[2]. We include a stripped down version of the module in the artifact.

We use two synthetic datasets to evaluate the attention model implementations. Kronecker graphs were chosen since they emulate realistic real world graphs with their heavy-tail skewed degree distribution. Erdős-Rényi graphs with random uniform degree distribution were used to support our analysis of communication volumes and the benefit of the global formulation for denser graphs. We also provide results for one real world dataset: Microsoft Knowledge Academic Graph (MAKG) with 111M vertices and 3.2B edges, which is a standard real world dataset for large-scale GNNs. We use one of the largest available GNN datasets to illustrate that the global formulation scales to very large node and core counts without the communication becoming a bottleneck.

We use DistDGL and dgNN as a baseline for comparison.

ARTIFACT INSTALLATION DEPLOYMENT PROCESS

Installation: Python Packages

time: up to 20 minutes

The Python software environment required by our attention model implementations can be installed with the help of a virtual environment automatically. Instructions on how to use use venv[3] can be found in the footnote.

step 1) create the virtual environment named gnn-global

```
% cd GNN_artifact
% python3 -m venv gnn-global
```

step 2) activate the virtual environment

```
% source gnn-global/bin/activate
```

step 3) install all necessary Python packages

```
% python3 -m pip install -r requirements.txt
```

step 4) install mpi4py using the MPI compiler of the target system

```
% MPICC=<the-correct-mpicc-in-the-system> python -m pip
  install mpi4py
```

Once the Python packages are installed, the virtual environment can later be activated by executing step 2). The virtual environment can be left by calling:

```
% deactivate
```

Installation: Piz Daint

time: 10 minutes

Piz Daint already provides most of the software environment necessary to execute the Python attention model implementations. So instead of installing the Python packages like mentioned above, only the following steps are necessary. Of course, the manual installation of the packages can also be done in a virtual environment.

```
% module load daint-gpu
% module load cudatoolkit
% module load numpy
% python -m pip install -U setuptools pip
% pip install cupy-cuda110
% pip install typing
```

Installation: Kronecker Shared Library

time: a few minutes

The shared library used to generate a Kronecker graph is based on the Kronecker module from the Graph500 benchmark.

```
% cd GNN_artifact/src/kronecker
% edit Makefile.inc
  - set CC to the MPI C compiler of your system
% make
```

result: `graph.so`

Experiments

A detailed description of the benchmarking workflow can be found in the Experimental Workflow paragraphs.

Since the submission of all relevant experiments can be handled by the job generator scripts, which hopefully should be simple to adapt to the target system, the submission time should only be a few minutes. After a few hours of waiting time hopefully all experiments were executed and all necessary results have been collected.

We used `float32` as datatype for all experiments and the default seed. Three layers were used for the experiments that were visualized in the submission. We used two runs as warm up and then executed the benchmark another ten times.

Figure 6: Strong Scaling. Benchmarks were run on Kronecker graphs with all three models (VA, GAT, AGNN). The respective job scripts can be generated and submitted with

[2]https://graph500.org/?page_id=47

[3]https://docs.python.org/3/tutorial/venv.html

`benchmark/unified_strong.sh`. We used 1, 4, 16, 64 and 256 compute nodes.

	#vertices	#edges	#features
6a	131072	171798692	16
6b	262144	687194767	16
6c	1048576	109951163	16
6d	2097152	439804651	16
6e	131072	171798692	128
6f	262144	687194767	128
6g	1048576	109951163	128
6h	2097152	439804651	128

Table 1: Experimental parameters for Figure 6

Figure 7: Weak Scaling Kronecker Graphs. Benchmarks were run on Kronecker graphs with all three models (VA, GAT, AGNN). The respective job scripts can be generated and submitted with `benchmark/unified_weak.sh`. A feature size of 16 was used for all experiments.

#nodes	#vertices	#edges
1	131072	17179869
4	262144	68719477
16	524288	274877907

Table 2: Experimental parameters for Figure 7a: sparsity = 0.1%

#nodes	#vertices	#edges
1	131072	1717987
4	262144	6871948
16	524288	27487791

Table 3: Experimental parameters for Figure 7b: sparsity = 0.01%

Plotting

time: a few minutes

The results that were used to create the figures in the submissions can be found in the `plots/results` directory. The figures from the submission can be created with the Python script `plots/create_plots.py`. The resulting PDF files can be found in the directory `plots/plots`.

To visualize the results from the artifact, we prepared a Python script (`plots/create_plots_artifact.py`). The script expects the experimental data in the file `plots/results/unified_results.csv`. The resulting PDF files can be found in the directory `plots/plots_new`.

Both Python scripts expect to be run from inside the plots directory.