

GRANII: Selection and Ordering of Primitives in GRAph Neural Networks using Input Inspection

Damitha Lenadora
University of Illinois at
Urbana-Champaign, USA
damitha2@illinois.edu

Vimarsh Sathia
University of Illinois at
Urbana-Champaign, USA
vsathia2@illinois.edu

Gerasimos Gerogiannis
University of Illinois at
Urbana-Champaign, USA
gg24@illinois.edu

Serif Yesil
NVIDIA, USA
syesil@nvidia.com.

Josep Torrellas
University of Illinois at
Urbana-Champaign, USA
torrella@illinois.edu

Charith Mendis
University of Illinois at
Urbana-Champaign, USA
charithm@illinois.edu

Abstract—Over the years, many frameworks and optimization techniques have been proposed to accelerate graph neural networks (GNNs). In contrast to the optimizations explored in these systems, we observe that different matrix re-associations of GNN computations lead to novel input-sensitive performance behavior. We leverage this observation to propose GRANII, a system that *exposes* different compositions of sparse and dense matrix primitives based on different matrix re-associations of GNN computations and *selects* the best among them based on input attributes. GRANII executes in two stages: (1) an *offline* compilation stage that enumerates all valid re-associations leading to different sparse-dense matrix compositions and uses *input-oblivious* pruning techniques to prune away clearly unprofitable candidates, and (2) an *online* runtime system that explores the remaining candidates and uses *light-weight* cost models to select the best re-association based on the input graph and the embedding sizes. On a wide range of configurations, GRANII achieves a geo-mean speedup of $1.56\times$ for inference and $1.4\times$ for training across multiple GNN models and systems. We also show GRANII’s technique functions on diverse implementations and with techniques such as sampling.

Index Terms—graph neural network, optimization, code generation, cost models, machine learning

I. INTRODUCTION

Graph Neural Networks (GNN) have gained adoption across a wide range of application domains including social media marketing [66], financial fraud detection [37], [38], drug discovery [28], [48], and systems optimization [30]. However, training GNNs is expensive and usually spans multiple hours, if not days. As a result, there have been many efforts, including software frameworks such as DGL [51], PyG [16], NeuGraph [39], and WiseGraph [25], compilers such as Graphiler [58] and SeaStar [57], and many other optimization techniques [26], [27], [47], [65] that aim at accelerating GNN computations.

Existing systems typically model GNN computations as individual phases with input-oblivious compositions of sparse or dense matrix operations (also known as primitives). In a typical GNN layer, each node initially collects and aggregates its neighbor node states (embeddings). Next, each node updates its state using these aggregated embeddings. The first computation

is modeled as a set of sparse matrix primitives (e.g., sparse matrix dense matrix multiplications (SpMM)), and the latter is modeled as a dense matrix multiplication (e.g., generalized matrix-matrix multiplication (GEMM)). Apart from these phases, different GNNs perform other required computations, such as normalization and edge weight (attention score) calculations also modeled as sparse or dense matrix primitives. Usually, these systems use a fixed *primitive composition* – selection of sparse and dense matrix primitives and an ordering between them – irrespective of the input and the GNN model configuration.

However, we notice that algebraically reassociating and reordering computations in GNNs can, in fact, change the underlying sparse and dense matrix primitive composition. For example, $A_d \cdot B_s \cdot C_d \cdot D_d$ is a subset of the computation from the Graph Convolutional Network (GCN)

model, where $\{X\}_s$ is a sparse matrix, and $\{X\}_d$ is a dense matrix. We can create two sparse and dense matrix primitive compositions by changing their association structure. $((A_d \cdot B_s \cdot C_d) \cdot D_d)$ is a primitive composition that consists of $\{sampled\ dense\ dense\ matrix\ multiplication\ (SDDMM), SpMM\}$ and $(A_d \cdot (B_s \cdot (C_d \cdot D_d)))$ leads to $\{GEMM, SpMM, GEMM\}$.

The optimal choice among these primitive compositions depends on the configurations of the GNN model and its input. Figure 1 shows speedups from the least input aware to the most aware primitive ordering strategies for the Graph Convolutional Network (GCN). Here, *static* has a single primitive ordering, *config* shows the speedup achievable by obtaining a primitive ordering by inspecting only the model configurations, such as embedding sizes, which was introduced in [60]. Going beyond these strategies, *all* inspects both the input graph and the model configurations to produce a primitive composition to

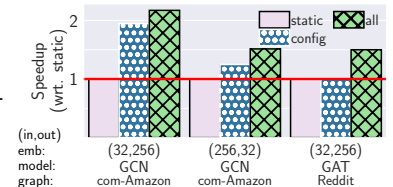


Fig. 1: Speedup for GCN with different primitive reorderings on different models, graphs, and embedding sizes. *static*:single ordering, *config*:model configuration based (embedding size), *all*:*config*+input-graph based

achieve the speedups shown. Although work that inspects the input to perform optimizations in GNNs exists [44], [53], [69], none of these systems consider automatic input-aware primitive selections or their ordering. The common practice is to build hard-coded GNN models with the same primitive compositions across different inputs and configurations. Furthermore, systems that select loop orderings such as [1], [12], [29], make input-unaware choices between adjacent kernels without considering the expanded region of the entire GNN model, nor the domain-specific optimization opportunities. As a result, existing systems fail to capture the optimization opportunities shown in Figure 1.

In this paper, to bridge this gap, we introduce GRANII, a compiler and a runtime that *automatically explores and selects* the best primitive composition in GNNs for a given input. First, GRANII exhaustively enumerates all legal primitive compositions for a given GNN through matrix re-associations using the underlying matrices’ characteristics (*dense vs sparse*, etc.). Then, GRANII selects the best among them using a lightweight machine-learning model. We had to overcome several challenges to realize this strategy.

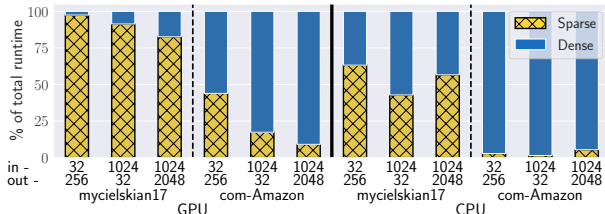


Fig. 2: Percentage of runtime (sparse/dense) across graphs for GCN, (in, out) embedding sizes, and underlying hardware.

Challenge: Exposing Primitive Compositions. Existing GNN systems [16], [51] do not facilitate efficient exploration of different primitive compositions as they implement models using the message-passing paradigm. Although the message-passing paradigm provides a natural interface for implementing GNNs using graph operations, it is insufficient to explore different primitive compositions. This is because it results in straight-line code that loses information on computations’ associativity and matrix metadata necessary to identify applicable primitives. GRANII overcomes this challenge by introducing a novel *matrix-based intermediate representation*, which stores additional metadata, enabling it to easily iterate through different re-association choices exhaustively for a given GNN model.

Challenge: Selecting the Most Performant Primitive Composition. Identifying the impact of each operation is necessary to select the most performant primitive composition of a GNN. This is challenging, as seen in Figure 2, where even the percentage of runtime taken by the basic categorization of sparse and dense matrix operations can change based on the model configurations, graph input, and underlying hardware. Thus, it is necessary to inspect all these factors, as making decisions based on just one is insufficient. To capture these complexities, GRANII uses lightweight *machine-learning-based cost models* for predicting the costs of each primitive for a given input and model configuration. Our results show that

cost estimates from these non-linear models allow GRANII to select the best primitive composition.

Challenge: Low Overhead Decision Making. To accurately predict the best primitive composition, the final decision should be taken at runtime when the input graph and model configuration are known. To reap the benefits of this optimization, the cost of making this decision at runtime must not outweigh the benefit itself. This becomes challenging as iterating through all possible primitive compositions while inspecting the input can result in a high overhead even when using GRANII’s lightweight cost models. GRANII overcomes this challenge by decoupling the decision-making process into two stages : (1) an *offline compilation stage* that enumerates all valid matrix re-associations and prunes clearly unprofitable candidates using input-oblivious rules and (2) an *online runtime system* that uses the lightweight cost models to select the best re-association given the input graph and the embedding size. This reduces the number of compositions that need inspections during runtime.

Our evaluations show that GRANII achieves significant speedups on five popular GNN models, including GCN, Graph Isomorphism Networks (GIN) [59], and Graph Attention Network (GAT) [49]. Speedups are observed across various graphs and embedding sizes on CPUs and GPUs. GRANII can also function with techniques such as sampling and models that require it (GraphSAGE [18]). We conduct our main evaluation on the state-of-the-art GNN framework WiseGraph [25], as well as the popular GNN framework DGL [51], to demonstrate the benefit of GRANII across multiple underlying systems. We make the following contributions.

- We perform a case study on five widely used GNN models (GCN, GIN, SGC [56], TAGCN [14], and GAT) and introduce different input-sensitive compositions of sparse and dense matrix primitives.
- We present an intermediate representation and a complementary technique to expose different compositions of sparse and dense matrix primitives for GNNs using re-association.
- We present GRANII, a compiler and runtime system, where given a GNN, selects the best composition of primitives based on the input using a two-stage technique designed to have minimal human intervention and overhead.
- We show that GRANII’s decisions lead a geo-mean speedup of $1.56\times$ for inference and $1.4\times$ for training.

II. BACKGROUND

GNN computations can be broken down into dense and sparse matrix primitives [51]. We briefly introduce such primitives and explain when and where they are used when computing GNN models.

A. Dense Matrix Primitives

Multiple dense matrix primitives are used in GNNs due to their inherent relation to neural networks. These are primitives where all inputs and outputs are dense matrices or vectors. Among the dense matrix primitives used in GNNs are element-wise computations such as non-linear functions and matrix multiplication variants. The latter, which includes general

matrix multiplication (GEMM), is commonly found in GNN operations such as updating features based on learned weights. Row-broadcast is another common primitive found in GNNs where a single value from a vector (e.g. a node-specific normalization value) is used to update an entire row in a matrix (e.g. node features), as shown by Equation (1).

$$c_{i,j} = d_i \times b_{i,j} \quad (1)$$

B. Sparse Matrix Primitives

We refer to matrix operations where at least one input uses a sparse matrix representation as sparse matrix primitives. For GNNs, [51] showed that all the sparse matrix operations necessary can be handled by the generalized versions of the two sparse matrix primitives: (a) sparse matrix dense matrix multiplication (g-SpMM) and (b) sampled dense-dense matrix multiplication (g-SDDMM). The former is a matrix multiplication with a sparse input, while the latter is a standard matrix multiplication guided by a sparse mask (elaborated in Appendix A). The standard SpMM and SDDMM use $+$ and \times as the addition and multiplication operators, whereas in the generalized form, the operations can come from any semi-ring [10]. We use \oplus and \otimes to denote the generalized addition and multiplication operators (e.g. $SpMM(\oplus, \otimes)$), we present a detailed example in Appendix B).

C. Matrix Primitives in GNNs

GNNs consist of two main stages: aggregation and update. During the aggregation stage, the embeddings possessed by each node or edge as a hidden state vector are passed among neighbors to form an aggregated message. The messages are then transformed into updated embeddings as the GNN layer's output during the update stage. Such computations are usually modeled as a collection of dense and sparse primitive operations [51]. Node-based aggregations are modeled as g-SpMM, edge-based aggregations are modeled as g-SDDMM, and updates are modeled as GEMMs.

Other computations apart from the aforementioned, such as normalization, open more opportunities to consider different matrix re-associations. This enables the use of different sparse-dense primitive compositions when computing GNN models.

III. CASE-STUDY: COMPOSITIONS IN POPULAR GNNs

We use this section to motivate the need for input-aware selection between compositions of sparse-dense matrix operations. We do this by elaborating on two different compositions for two popular GNN models, (a) Graph Convolutional Network (GCN) [32], and (b) Graph Attention Network (GAT) [49]. Note that there can be different operator orderings within a composition, such as deciding between performing the update (GEMM) or aggregate (SpMM) operation first.

We use Figure 3 to depict pairs of different primitive compositions for GCN and GAT along with their complexities. Here, we observe that certain compositions (detailed in the remaining part of this section) are typically more suitable for comparatively denser or sparser graphs. However, as we detail in Section VI-G, there are multiple factors, such as

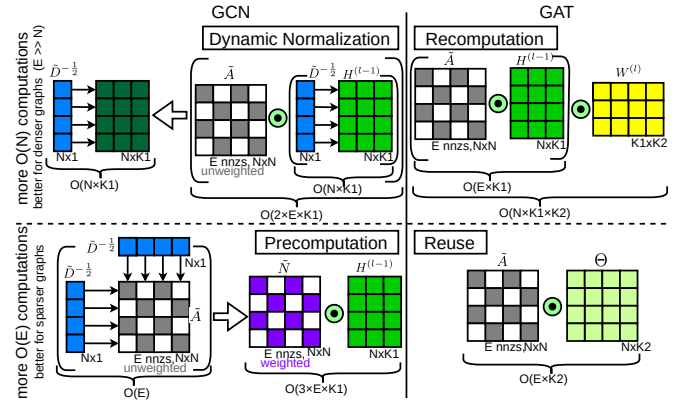


Fig. 3: Different compositions for GCN and GAT with complexities. N is the number of nodes in the graph, E is the number of edges, $K1$ and $K2$ are the input and output embedding sizes respectively. The complexities shown are per operation.

the underlying hardware, as well as the non-zero distribution of the input graph, that have a significant impact on the actual computation time. This makes the selection of the most performant composition challenging, and is a motivating factor for a data-driven approach, which we apply in our solution.

A. GCN

The GCN computation has three main calculations: normalization, aggregation, and update. The normalization value can be subsumed by the graph's node features in two different ways, creating two different primitive compositions. Our evaluations show that selecting the correct composition in GCNs can lead to a geo-mean speedup of $1.88\times$ (Section VI-C1).

Dynamic Normalization-Based Primitive Composition. This composition of GCN normalizes using two row-broadcasts as presented by Equation (2). Here, \tilde{A} is the adjacency matrix of the input graph with self-edges. \tilde{D} is the degree of nodes in \tilde{A} . $H^{(l)}$, $W^{(l)}$ are the node embeddings and weights for the l^{th} layer in the GNN. The result is updated after normalization and aggregation. Finally, the non-linear function σ is applied.

$$H^{(l)} = \sigma(\underbrace{\tilde{A} \cdot \tilde{D}^{-\frac{1}{2}}}_{\text{row broadcast}} \otimes H^{(l-1)}) \cdot W^{(l)} \otimes \tilde{D}^{-\frac{1}{2}} \quad (2)$$

Note that for unweighted graphs, a computationally less expensive aggregation operation that does not use the edge values of the graph (mentioned in Appendix B) can be used for aggregation. This composition is more beneficial for denser graphs as it reduces the burden of the aggregation when computing the final result. This is because denser graphs have a comparatively larger number of edges, which directly contribute to the complexity of the aggregation.

Precomputation-Based Primitive Composition. This composition of GCN pre-computes the normalized adjacency matrix \tilde{N} , by using an SDDMM primitive as shown in Equation (3). This normalized matrix is then used by the aggregation operations of the model. This primitive composition is more suitable for sparser graphs, as it does not perform the row-broadcast operations. This is because the complexity of these

operations is based on the number of nodes of the graph, where in sparser graphs, the nodes are more abundant.

$$\tilde{N} = \underbrace{(\tilde{D}^{-\frac{1}{2}} \cdot \tilde{A} \cdot \tilde{D}^{-\frac{1}{2}})}_{\text{SDDMM}} \quad H^{(l)} = \sigma(\tilde{N} \cdot H^{(l-1)} \cdot W^{(l)}) \quad (3)$$

B. GAT

We summarize the attention calculation as a function (*Atten*) to simplify the explanation of the model (Equation (4)). Note that this function uses the updated input embeddings of the nodes ($\Theta = H^{(l-1)} \cdot W^{(l)}$). $W_A^{(l)}$ refers to attention weights used during this calculation. The result is a sparse matrix α containing the necessary attention scores, which is then used for the aggregation as seen in Equation (5). However, based on the decision to reuse the updated input embeddings of the nodes in this aggregation, two different primitive compositions can be identified for GAT.

Reuse-Based Primitive Composition. During aggregation, this composition reuses the updated embeddings already computed in the attention calculation stage.

$$\alpha_{n \times n}^{(l)} = \text{Atten}(\tilde{A}, \underbrace{H^{(l-1)} \cdot W^{(l)}}_{\Theta}, W_A^{(l)}) \quad (4)$$

$$H^{(l)} = \sigma(\alpha^{(l)} \cdot \underbrace{\Theta}_{\text{reuse}}) \quad (5)$$

Recomputation-Based Primitive Composition. This composition ignores the reuse of the updated embeddings to generate the primitive composition by using the original embeddings for aggregation. However, this requires an additional GEMM computation, as shown by Equation (6). Thus, it is useful only in situations where the aggregation operation between the node features ($H^{(l-1)}$ or Θ) and the graph updated with the attention scores ($\alpha^{(l)}$) becomes less expensive surpassing the cost of the additional GEMM computation. i.e., when the input embedding size is smaller than the output embedding size.

$$Z^{(l)} = \underbrace{(\alpha^{(l)} \cdot H^{(l-1)})}_{\text{recomputation of } \Theta} \cdot W^{(l)} \quad (6)$$

IV. GRANII SYSTEM

```
import GRANII
graph, node_feats, labels = ...
model = GraphConv(...)
GRANII(model, graph, node_feats, labels) #<-Only change
res = model(graph, node_feats)
```

Fig. 4: Using GRANII.

We present GRANII: a compiler that produces an executable that *automatically explores and selects* the best composition of sparse-dense matrix primitives in GNNs for a given input graph and model configuration when executed on a target hardware.

A. Overview

GRANII requires minimal user interaction to set up and accelerate GNN code. GRANII is ready to be used by simply running an initialization script that gathers profiling data and trains its cost models. Once this process is done, a user only needs to provide the GNN model to be accelerated as well as the inputs (graph, node features and labels) as parameters, as shown in Figure 4. The GRANII compiler then operates on the GNN model code and replaces the existing GNN model

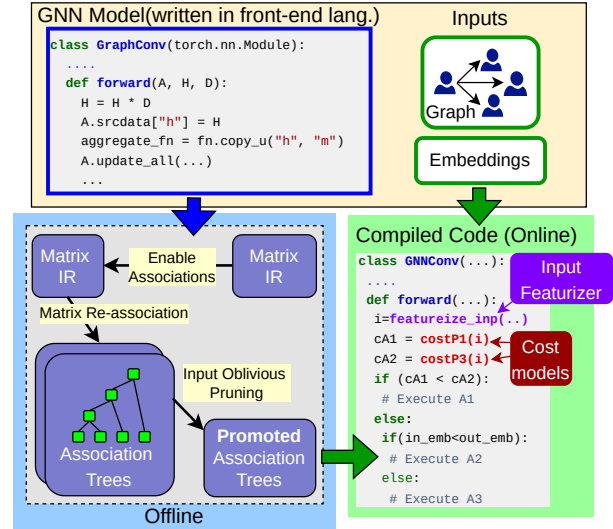


Fig. 5: Overview of the GRANII system.

with an accelerated version. The user can then run the initial code without additional changes.

Figure 5 presents the overall process of GRANII. It has two main stages: (1) an offline compilation stage for generating potential candidates with different sparse-dense matrix primitive compositions, and (2) an online stage for selecting the best primitive composition based on the input.

In the offline compilation stage, GRANII first converts a GNN model written using the message passing paradigm [51] to a matrix intermediate representation (IR) (Section IV-B). It uses this matrix IR form to generate all possible sparse-dense matrix primitive compositions that can be used to implement the given GNN model as potential candidates using operator re-association (Section IV-C). GRANII then prunes away clearly unprofitable candidates that are oblivious to the input using rules. Finally, GRANII generates an executable code with the final set of candidates, which are promoted to be selected during the online stage (Section IV-D).

In the online stage, GRANII selects the primitive composition that has the least cost out of the promoted candidates for the given input. It

evaluates each candidate's cost using a set of *input-sensitive cost models*, each specialized to a sparse or dense matrix primitive (Section IV-E). Compared to the pruning rules used in the offline stage, these cost models consume graph features and embedding sizes as inputs, allowing GRANII to make input-aware predictions about the relative cost of different compositions. GRANII selects the composition with the minimum total cost to execute using the given input.

This decoupled design requires the user to run only the online stage of GRANII for different input graphs, while the offline compilation stage only needs to run once.

TABLE I: Matrix Attributes

Attribute	Sub-attribute : details
dense	data : Contains data weight : Contains learnable weights
sparse	weighted : Uses edge values unweighted : Only NNZ positions diagonal : A diagonal matrix

B. Matrix Representation Generation

We use Figure 6 as a running example to illustrate the offline stage of GRANII for GCN (Section III-A).

Matrix Representation. We use a matrix-based intermediate representation (IR) through the initial offline stages of GRANII. We use this representation to make it easier to generate different primitive compositions via operator re-association. This representation is tree-based, where leaf nodes represent matrices and intermediate nodes represent matrix operations such as *multiplication*, *addition*, and *row broadcast*. A leaf matrix node, $A_{(X \times Y)}^{\text{attr subattr}}$ contains information regarding the matrix size ($X \times Y$), and the attributes (attr, e.g. *dense* or *sparse*) and sub-attributes (subattr, e.g. *data* or *weighted*) described in Table I. Attributes provide essential information to generate primitive compositions in the latter part of GRANII’s offline compilation stage. The leaf nodes, which represent input data, are then consumed by matrix operations in the GNN (e.g. Figure 6(b), D and H are consumed by \otimes , a row-broadcast). These operations then form a hierarchical structure based on input dependency (e.g. Figure 6(b), the result of the row-broadcast is used for the input of the matrix-multiplication). This is similar to computation graphs in tensor frameworks such as PyTorch [41]. However, in addition to the matrix attributes, our IR differs from computational graphs as we also represent operations that are associative in a single level (e.g. Figure 6(b), the multiplications between A , the result of D and H after \otimes , and W are all associative and linked to \odot). GRANII uses these details to generate associations and the final code.

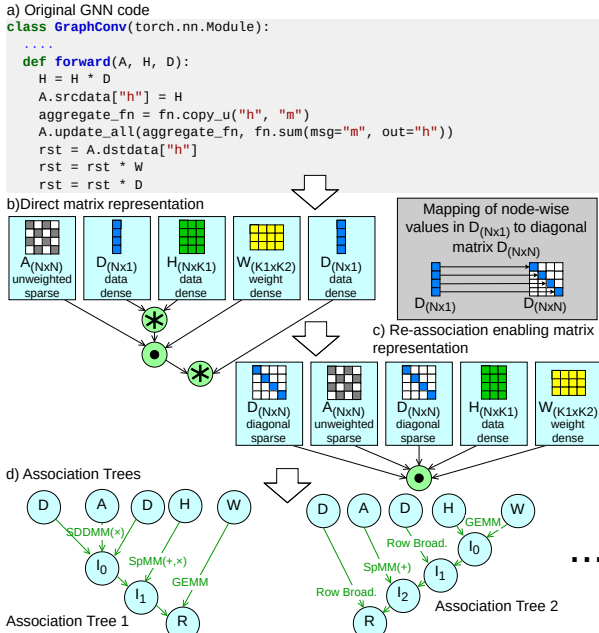


Fig. 6: Association tree generation from GNN code. (\otimes denotes row broadcast, \odot denotes matrix multiplication)

Code Translation. GRANII first translates code written using a GNN framework’s API (WiseGraph [25], DGL [51]), into the matrix-based IR. GRANII can support GNN frameworks that follow the popular message passing paradigm [51] (Figure 6(a) shows an example GNN implementation), which models graph

operations as messages passed between nodes. For example, a node feature aggregation is modeled as passing a message from a source to a destination, where at the destination, all messages are summed together to give an aggregated result. Graph operations specified in this paradigm, along with general matrix operations of the underlying Machine Learning Framework (PyTorch [41]) are then lowered down into matrix operations. We use a rule-based parser that functions on the AST of the source language (Python), for this task (e.g., `update_all graph` operation is mapped to *multiplication*). Note that we consider non-linear operations such as ReLU and SoftMax as barriers preventing re-association. This is because we only focus on semantically equivalent re-associations. GRANII’s parser also collects information from the original GNN code to fill in the attribute details for the leaf matrix nodes (e.g., the adjacency matrix A ’s attribute is *sparse*).

First, GRANII uses all the information collected to create the leaf nodes of the IR. GRANII then creates the hierarchical tree of operations in the IR based on data dependencies and rules regarding the associativity of the matrix operations in the GNN model (e.g. adjacent multiplication operations are associative and are thus consumed at a single level using a multiplication operation). For the initial code in Figure 6(a), this results in the matrix IR in Figure 6(b). The flattened version of the matrix IR in Figure 6(b) is $(A \odot (H \otimes D) \odot W) \otimes D$ (\otimes refers to a row-broadcast operation, \odot refers to matrix multiplication). In addition, GRANII also runs an additional IR rewrite pass on the matrix IR to uncover more opportunities for arbitrary re-association. GRANII does this to eliminate *row-broadcast* operations, which act as barriers for re-associations. Instead, these operations can be represented as a matrix multiplication as shown by Figure 6(c) (detailed in Appendix C).

C. Generating Sparse-Dense Matrix Compositions

Association Tree Representation. GRANII converts the matrix IR into *association trees* to concretely represent sparse or dense matrix primitives in place of matrix operations. A given association tree represents one possible matrix re-association. *Association trees* reuse the same matrix nodes used in the matrix IR as leaves. Internal nodes of the tree represent intermediate results, and edges represent sparse or dense matrix primitives. For example, consider the association tree 1 shown in Figure 6(d), where the multiplication operations in Figure 6(c) are replaced by applying the *SDDMM*, *SpMM*, and *GEMM* primitives in order.

Association Tree Generation. GRANII generates *association trees* for all valid matrix re-associations for a computation represented in the matrix IR. Each association tree uses its own set of sparse and dense matrix primitives, and different associations lead to different primitive compositions. Figure 6(d) shows two possible associations for the matrix IR. In this example, association tree 1 represents the flattened computation $((D \odot A \odot D) \odot H) \odot W$. The association of $(D \odot A \odot D)$ yields a *SDDMM* primitive. However, association tree 2 does not take this route, resulting in a primitive composition that uses row-broadcasts instead.

Algorithm 1 shows how GRANII recursively enumerates all possible association trees in a depth-first manner. It accepts the current matrix IR and an association tree as input. Then, GRANII traverses the current matrix IR to find association candidates that involve only leaf nodes (line 2). GRANII uses a set of rules to find such candidates using connected operations, attributes, and sub-attributes of the matrix IR nodes and also to specify which dense or sparse matrix primitive to use for that association (we present a few rules in Appendix D).

Algorithm 1: Generation of Association Trees

```

Input: Matrix IR of GNN code ( $mIR$ ), current association tree ( $tr$ )
Output: Association Forest ( $fr$ )
1 Function generateTree( $mIR, tr$ ):
   /* get resolved associations */
2    $cands \leftarrow getCandidates(mIR)$ ; /* rule-based */
3   if  $cands$  is empty then
4      $fr.add(tr)$  /* add tree to forest */
5   else
6     foreach  $cand \in cands$  do
7        $newIR, newTr \leftarrow apply(mIR, tr, cand)$ 
8        $generateTree(newIR, newTr)$ 
9     end
10  end

```

For every candidate in this list, GRANII creates a new tree ($newTr$) by augmenting the current tree with (a) an internal node corresponding to the result of the association, and (b) edges annotated with the sparse or dense matrix primitive that connect this node to the associated nodes. Simultaneously, it creates a new matrix IR ($newIR$) that replaces the set of associated nodes with this new internal node (line 7) in the current matrix IR. For example, consider the association tree 1 in Figure 6(d). During its creation, nodes D, A, D are first associated with the operation $SDDMM$ to produce the intermediate result I_0 . Simultaneously, GRANII also creates a new matrix IR that replaces nodes D, A, D with I_0 . This tree is further expanded recursively until no more association candidates can be found. Since we enumerate all candidate associations (line 6), this algorithm produces a forest of all valid association trees as the output. Once the trees are fully generated, GRANII scans all trees to exploit any opportunities to reuse computed values (common sub-expression elimination in the compiler domain).

Pruning Associations. GRANII prunes unprofitable candidates irrespective of the input from the generated forest of association trees. We find sets of unprofitable candidates under two scenarios: 1) the input embedding size is larger or equal to the output ($<$), and 2) vice-versa ($>$). For each scenario, we identify unprofitable candidates using the following rules:

- A subset of a candidate tree’s primitives being equal to the total set of primitives of another candidate (e.g., a candidate performing $SpMM$ and a $GEMM$ is unprofitable compared to another candidate performing only $SpMM$ on the same matrix sizes). This rule also removes duplicates.
- A candidate with the same matrix primitives as another tree, but with larger matrices as inputs to the primitives.

GRANII finds unprofitable trees that are common in both scenarios and prunes them away. At this point, it cannot prune further without inspecting the input. GRANII promotes the

remaining association trees to be inspected during the online stage. It also annotates the candidates when they were profitable ($<$, $>$) to use during the final code generation (Section IV-D).

D. Code Generation of Promoted Candidates

The final component of GRANII’s offline stage generates code for the promoted association trees, out of which the best is executed during the online stage. This is achieved by generating conditionally executed code as shown in Figure 7. GRANII supports two types of runtime conditions: (1) simpler conditions based purely on embedding sizes and (2) conditions based on more complicated cost models for matrix primitives.

Conditions Using Embedding Sizes. GRANII first identifies trees that are profitable at runtime only using embedding sizes. It does this by categorizing annotated trees that are profitable when either the input embedding size is larger than or equal to the output ($>$), and 2) vice-versa ($<$). This avoids the use of the more expensive cost models. Figure 7 shows an example where the promoted association tree $A3$ is the only one profitable when the input embedding size is smaller.

Conditions Using Cost Models. For the rest of the association trees, GRANII uses cost comparisons using per-primitive cost models that depend on both the embedding size and the input graph. Section IV-E describes how we develop these cost models. We approximate the cost of executing an association tree by the addition of the costs of each primitive given by the cost models. If multiple association trees result in the same cost, GRANII selects one tree among them as they are equivalent.

Code Generation. Once GRANII has collected the runtime conditions, it progressively generates conditional code to execute the GNN implementation with the best primitive composition. For runtime conditions that require cost comparisons, it embeds the code for cost models for each matrix primitive. Once the runtime conditionals are generated, GRANII lowers the matrix primitives of each association tree to kernel calls that are supported by the underlying GNN framework. Figure 7 shows an example of the final conditionally executable code.

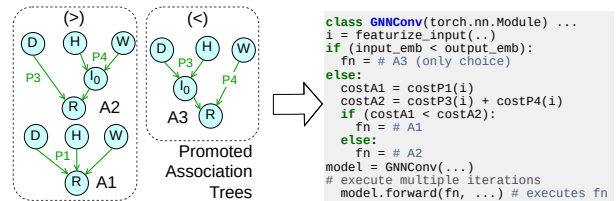


Fig. 7: Code-generation of promoted association trees (Online).

E. Cost Models for Primitives

GRANII’s cost models predict the cost of a particular matrix primitive, given the input graph and embedding size. We use an input featurizer to create an embedding of the input graph’s characteristics and GNN embedding sizes to feed into simple XGBoost-based [6] cost models.

1) **Input Featurizer:** GRANII’s cost models use the hand-crafted features such as the sparsity of the graph (detailed in Appendix E), along with embedding sizes when making predictions. The *input featurizer* efficiently inspects the input

graph at run time to obtain the necessary graph features and concatenates the resulting embedding with the GNN embedding sizes to create the final featurized input embedding. We avoided using automatic feature extraction methods such as sparse convolutional networks [68] as they are challenging to scale to larger graphs that GRANII targets to support.

2) *Lightweight Learned Cost Models*: GRANII uses XGBoost [6] regression models to predict the input-aware cost of executing matrix primitives. GRANII trains these models per each dense and sparse matrix primitive, and target hardware architecture. This is a one-time cost per target system as the number of matrix primitives, especially sparse matrix primitives used by GNNs, is limited in number, as shown by [51](g-SpMM for weighted and unweighted graphs, GEMM, row-broadcast, etc.). Thus, re-training the cost models for each new GNN is unnecessary. To be input aware, these models use the embedding given by the *input featurizer* to predict the cost of running (time taken) a particular matrix primitive. The final cost of GNN execution is the sum of the costs of these primitives. GRANII uses a set of profiling data with varying graphs and embedding sizes when training these models (Section V).

V. IMPLEMENTATION

In this section, we explain our training setup and implementation of the components of GRANII. These implementations are done in the Python ecosystem.

Offline Part of GRANII. We implemented GRANII using Python and PyTorch. Both WiseGraph and DGL possess Python APIs, which we translate to matrix IR (front-end) using Python’s Abstract Syntax Tree (AST). This translation and the association trees to kernel code (back-end) are done using a one-to-one mapping scheme.

Training Lightweight Cost Models. We collect the training data for the cost models by profiling different matrix primitives on the machines listed below.

- CPU - Intel Xeon Gold 6348, RAM 1TB, No GPU
- A100 - NVIDIA A100 GPU, Intel Xeon Platinum 8358
- H100 - NVIDIA H100 GPU, AMD EPYC 9454

We source the input graphs to train the model from the SuiteSparse matrix collection. Here, we chose a set of undirected graphs ranging from 1 million to 100 million non-zero values, which we further varied using sampling. Using this method and varying the input and output embedding sizes from 32 to 2048, we gathered approximately 700 to 8000 data points for each matrix primitive. We use a subset of these graphs as the validation set when training, and note that these do not include the graphs we use in our evaluation (test set).

VI. EVALUATION

We present our evaluation of GRANII, by comparing it against the default sparse-dense primitive compositions for GNN inference and training found in WiseGraph [25], and DGL (v.2.4, released 2024). For this evaluation, we test across various graphs (1 million - 126 million non-zeros with different sparsity patterns) in both GPU (H100 and A100) and CPU platforms. We used multiple combinations of input

and output embedding sizes to showcase different trends in the performance of different input-sensitive compositions. In the subsequent sections, we present detailed setups used for evaluation, along with performance results and analyses.

A. Research Questions

We aim to answer the following questions,

- 1) How well does GRANII optimize five popular GNN models (GCN, GIN, TAGCN, SGC, and GAT) on a diverse set of input graphs and embedding sizes? (Section VI-C).
- 2) How well does GRANII perform on end-to-end workloads? (Section VI-D)
- 3) How does sampling effect GRANII? (Section VI-E).
- 4) How does multiple GNN model layers affect GRANII? (Section VI-F).
- 5) How accurate is GRANII’s learned models? (Section VI-G)

B. Experimental Setup

TABLE II: Graphs used for evaluation

	Graph	Nodes	Edges	Source
RD	Reddit	232,965	114,615,892	DGL
CA	com-Amazon	334,863	2,186,607	SS
MC	mycielskian17	98,303	100,245,742	SS
BL	belgium_osm	1,441,295	4,541,235	SS
AU	coAuthorsCiteseer	227,320	1,855,588	SS
OP	ogbn-products	2,449,029	126,167,053	OGB

Baseline Systems. We perform an extensive evaluation to show the benefits of GRANII. We conduct our main evaluations against the WiseGraph [25], and the PyTorch [41] back-end of DGL (v2.4) [51] using available implementations of each model, and if not, implementing a competitive baseline. We selected two systems to showcase GRANII’s generality using two separate systems, while also showing performance against a state-of-the-art GNN system such as WiseGraph.

Some models in both systems’ available implementations contain operator reordering strategies that utilize the model configuration. For example, the ordering between an update operation (GEMM) and an aggregate operation (SpMM) is decided based on the embedding sizes of the model. Here, the update operation is done first if the input embedding size is larger than the output embedding size [60], and vice versa. When implementing models without an existing baseline implementation, we used a configuration-based operator reordering to showcase a more competitive baseline. Note that GRANII can automatically identify these opportunities.

GNN Models. We use 5 GNN models to evaluate GRANII, (1) GCN [32], (2) GAT [49], (3) Graph Isomorphism Networks (GIN) [59], (4) Topology Adaptive Graph Convolutional Networks (TAGCN) [14], and (5) Simple Graph Convolution (SGC) [56]. Among these models, GCN, GAT, and GIN are canonical models that have been used to evaluate multiple systems [25], [57], [58], [65], [69] and used in recent works [13], [20], [23], [35], [36], [52]. A recent study has shown that these models with proper architectural configurations can achieve significant levels of accuracy compared to more

complex models [4], [43]. The total number of compositions through re-associations and offline pruning pairs of GRANII for GCN, GAT, and GIN, respectively, are 12 and 8, 2 and 0, as well as 8 and 4. In addition, we use TAGCN and SGC to showcase GRANII’s generalizability, which have also been used in various recent studies [2], [42], [50].

Datasets/Graphs. We use graphs with multiple variations of non-zero distributions, sourced from domains related to GNN computations. These graphs are listed in Table II. Among the various types of graphs, we find road graphs (*belgium_osc*), highly dense graphs (*mycielskian17*), and power law graphs (*Reddit*). We sourced these graphs from three main locations, which are regularly used to evaluate GNN systems [5], [25], [51]. These were SuiteSparse (SS) [11] – which provided a plethora of graphs of varying characteristics, DGL and Open Graph Benchmark (OGB) [22] – which provided graphs commonly used in the GNN context. The graphs collected for the evaluation were undirected and unweighted, with no overlap with the graphs used for training.

Model Configurations. We use a wide range of embedding sizes for evaluations to showcase the scalability and practicality of GRANII. We chose embedding size combinations ranging from 32 to 2048, as we observed a larger range of embedding sizes when inspecting the top ranks of Open Graph Benchmark’s leaderboards [46], [67]. In addition to this real-world observation, we also observed similar variations in embedding sizes in multiple academic works [15], [31], [54], [64]. Note that we only evaluate increasing embedding sizes for GAT, as this is the scenario in which the primitive composition choice is non-trivial (discussed in Section III-B).

We use a single-layer GNN for evaluation since the decisions by GRANII apply to a singular GNN layer. An extension to a multi-layered GNN is achievable by chaining the decisions made for each separate layer. To prove this point, we also show the end-to-end results of GNN models comprised of multiple layers separately in Section VI-D.

Testbed Machines. We evaluate GRANII on the machines listed in Section V (A100, H100, and CPU). We include CPU evaluations as they are a valid option during inference.

C. Performance Comparison

TABLE III: Geomean speedups of GRANII across graphs and configurations for 100 iterations. Mode - Inference(I)/Training(T)

	HW	Mode	Overall	GCN	GIN	SGC	TAGCN	GAT
WiseGraph	H100	I	1.24×	1.46×	1.10×	1.24×	1.13×	1.38×
		T	1.17×	1.21×	1.05×	1.13×	1.1×	1.47×
	A100	I	4.26×	10.39×	1.09×	7.77×	6.57×	1×
		T	3.67×	7.87×	1.03×	6.44×	5.81×	1×
DGL	H100	I	1.24×	1.12×	1.24×	1.32×	1.07×	1.74×
		T	1.08×	1.03×	1.03×	1.06×	1.02×	1.49×
	A100	I	1.26×	1.18×	1.28×	1.33×	1.06×	1.8×
		T	1.09×	1.03×	1.05×	1.06×	1.02×	1.55×
	CPU	I	1.2×	1.31×	1.2×	1.24×	1.03×	1.34×
		T	1.12×	1.02×	1.1×	1.15×	1.01×	1.55×
Overall	I	1.56×	1.88×	1.18×	1.82×	1.53×	1.45×	
	T	1.4×	1.58×	1.05×	1.55×	1.45×	1.43×	

The optimal composition and ordering of primitives are input-dependent. We show that GRANII almost always selects the best input-dependent choice compared to hand-crafted heuristics found in Section VI-G, resulting in significant speedups shown in this section. We perform all our evaluations for 100 iterations across all inputs and settings. We chose 100 to represent the number of times a GNN would typically run, but this can range from one iteration during inference to thousands of iterations [55] when training. Here, we compare against existing implementations of GNNs in each GNN system (WiseGraph and DGL) for five GNN models: (a) GCN, (b) GIN, (c) TAGCN, (d) SGC, and (e) GAT. We present summarized results in Table III, while full per-graph results are presented in Figure 8. These results also include the overheads of GRANII, which are the feature extraction time and the composition selection time. We analyze general trends first, before providing an in-depth analysis of the results in Section VI-C1.

We observe varying speedups for GRANII across systems, underlying hardware, GNN models, and execution modes (inference and training). Overall, we achieve a geomean speedup of 1.56× for inference and 1.4× for training across all these settings. The training speedup being lower than the inference speedup is a common trend that we observe across systems and most models. This result can be attributed to the additional computations in the backward pass. The backward pass, where the gradient updates occur for the machine learning model’s learned weights, is part of the entire execution when training a model. However, GRANII does not perform operator selection for the backward pass, as it only optimizes the forward pass (inference). Nonetheless, GRANII still improves the overall training time as the forward pass is optimized. We observe a significant geo-mean speedup for *WiseGraph* when evaluating the GCN model on the A100 machine, while also observing no speedup for the GAT model in the same execution context. We elaborate on this point in Section VI-C1, where we perform an additional in-depth evaluation to analyze the individual speedups that GRANII obtains. In Section VI-E, we additionally show that GRANII is beneficial even with techniques such as sampling. Notably, through sampling, we can support GraphSAGE [19] with GCN aggregation.

1) *In-depth comparison:* Using Figure 8, we now perform a more in-depth analysis and explain some notable results.

Speedups. GRANII discovers two different primitive compositions (similar to the ones described in Section III) for each GNN model we evaluate. However, within these compositions, there can be different operator reorderings (e.g., update (GEMM) being performed first if the embedding size decreases). If the default implementation of a system is predicted to be the best for a given setting by GRANII, the speedup observed is 1 (blue line in Figure 8). This is the case for Figure 8(j) where the optimal composition is the default implementation. Any variation from the baseline (i.e., any point where the observed speedup is greater or less than 1), indicates that GRANII chose a different primitive composition or operator reordering. GRANII achieves significant speedups for all graphs, including our largest graph, *ogbn-products(OB)* where an overall geo-

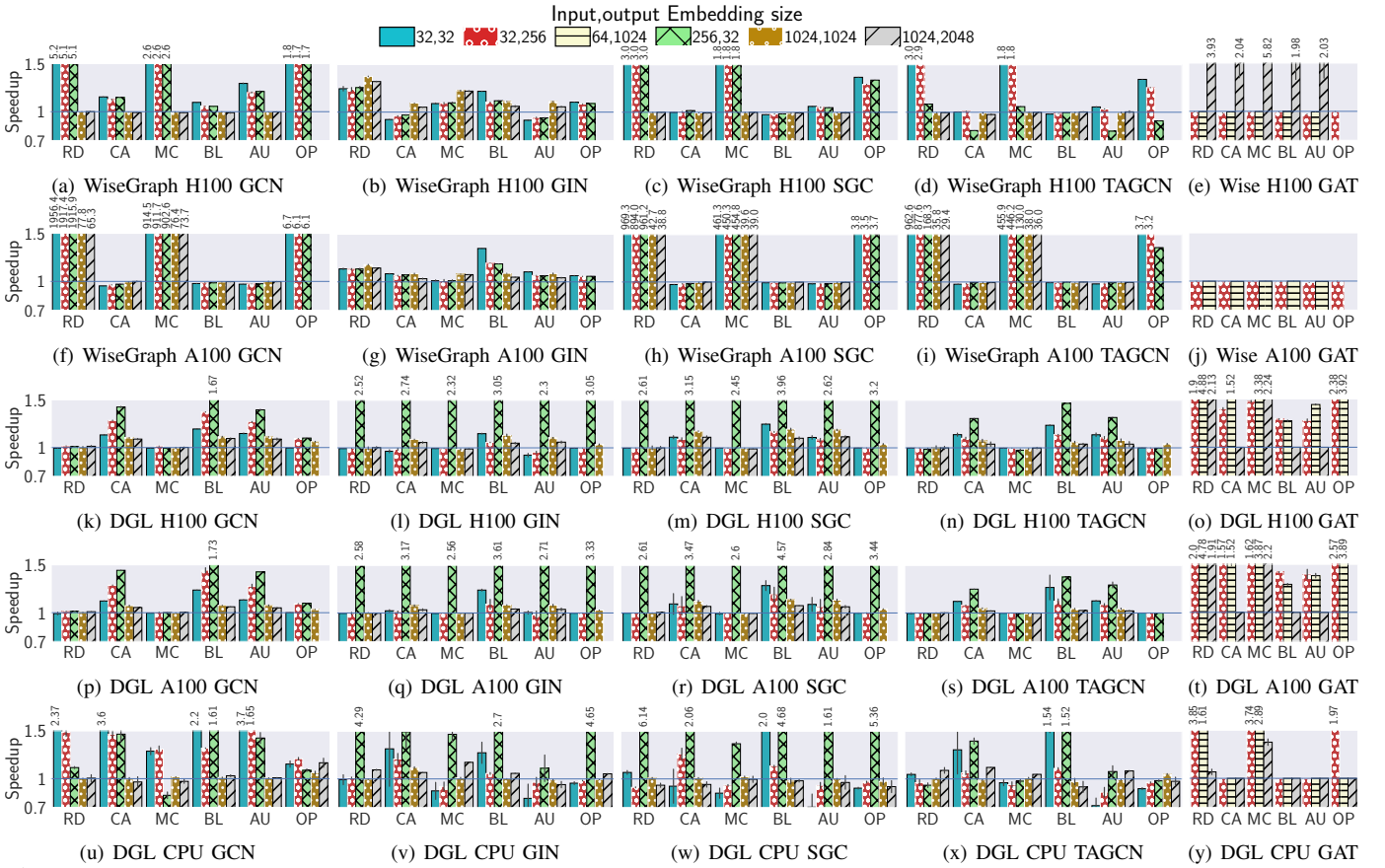


Fig. 8: GRANII’s speedups over WiseGraph and DGL, across different models, configurations, and underlying hardware for 100 iterations with runtime overheads. Empty spaces are caused by OOM failures and illegal memory access errors. We used only increasing embedding sizes for evaluating GAT, as it is the only scenario where an input-aware decision needs to be made.

mean speedup of $1.42\times$ is observed.

Most notable are speedups for denser graphs (RD, MC, and OP) for GCN, SGC, and TAGCN on the WiseGraph evaluations. Upon inspecting the default implementation and the decisions made by GRANII, we observed that the WiseGraph uses a PyTorch binning function when calculating the normalization values. This function was used to calculate the number of outgoing edges by binning each relevant edge on its outgoing node. For denser graphs, the number of bins is small compared to the overall binned values, and thus causes slowdowns due to the higher amount of atomic operations. Choosing a primitive composition that circumvents this function allows GRANII to achieve the speedups shown. GRANII’s also achieves significant speedups through operator reordering, as shown by the GIN and SGC models for DGL. Here, the default implementation for these models does not reorder the placement of the update (GEMM) operation. However, a better ordering that shifts the placement of this update operation to produce a computationally less expensive execution is automatically identified by GRANII to achieve the speedups shown. In DGL, GRANII achieves more significant speedups for these models (GCN, SGC, TAGCN) for sparser graphs (BL, AU, and CA) as DGL’s existing implementation is more suited for denser graphs (dynamic-normalization in Section III for GCN).

For GAT, WiseGraph uses the model configuration details, such as the embedding sizes, to select the primitive composition. This leads to WiseGraph always recomputing the updated node features for increasing embedding size combinations (mentioned in Section III for GAT). However, this recomputation operation can be very costly, especially for large embedding size combinations, as shown by Figure 8(e), GRANII identifies these occasions and opts to reuse the updated node features at the cost of a more expensive aggregate operation. Meanwhile, DGL does the opposite by default and always chooses to reuse the updated node features. Again, this is not always beneficial, and GRANII can correctly identify opportunities where recomputing the node embedding is more beneficial to achieve the speedups shown in Figures 8(o),(t), and (y).

Slowdowns. The decisions made by GRANII can result in slowdowns as seen in Figure 8(d). Here, GRANII fails at selecting the optimal composition (where the default is the better option). This is due to GRANII using data-driven cost models, where the costs predicted may not be perfectly accurate. This is especially true in situations where the costs are very similar. In addition, we observe significant variations in CPU runtimes as shown by Figures 8(v), (w), and (x). These effects can be mitigated further with more data to train the cost models of GRANII. In addition, GRANII still surpasses the decisions

made only using heuristics as shown in Section VI-G.

Difference Across Hardware. Considering the underlying hardware, from the H100, to A100, to CPU, we observe that dense operations gradually become more optimized. This leads to different optimal choices across different hardware as seen by the results for the 1024, 1024 embedding size combination for Reddit in GCN for WiseGraph (Figures 8(a) and (f)). This stands as evidence that GRANII’s data-driven method can adapt to different execution settings compared to hand-crafted heuristics, which an expert user would have needed to tune.

Overheads. The graph feature extraction and composition selection overheads are very minimal – 7ms at most for GPU, and 0.42s for CPU. In terms of iterations, this is a maximum of 4.4× of a single GNN iteration for the GPUs, and 1.1× for CPU. Both overheads are incurred only once during runtime.

D. End-to-End Results

TABLE IV: End-to-end results with speedups for GRANII on H100 (GRANII’s speedup is shown in parenthesis)

Graph	Node Features	Classes	GNN	Hidden Dims.	Execution time (ms)			
					Wise		DGL	
					Default	GRANII	Default	GRANII
Reddit	602	41	GCN	32	48.3	9.4 (5.14×)	17.2	16.8 (1.02×)
				256	67.3	48.8 (1.38×)	32.1	31.8 (1.01×)
				1024	96.4	78.2 (1.23×)	74.3	73.7 (1.01×)
			GAT	32	22.2	24.4 (0.9×)	114.8	114.8 (1×)
				256	41.8	41.8 (1×)	150.7	150.7 (1×)
				1024	82.9	82.9 (1×)	328.5	202.9 (1.62×)
ogbn-products	100	47	GCN	32	28.5	15.6 (1.83×)	31.7	31.1 (1.02×)
				256	55.9	50.3 (1.11×)	51.3	49.6 (1.03×)
				1024	76.18	70.63 (1.08×)	80.1	70.9 (1.13×)
			GAT	32	32.94	32.94 (1×)	118.4	118.4 (1×)
				256	63.07	63.07 (1×)	202.1	138.3 (1.46×)
				1024	Error (Illegal memory access)		547.3	215.4 (2.54×)

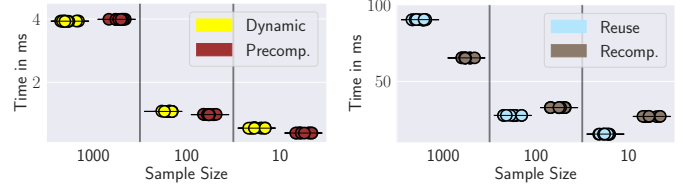
Table IV shows the runtime numbers for the forward pass on the H100 machine on end-to-end GCN and GAT models. Here, we evaluated the models using the Reddit and ogbn-products graph datasets with a single hidden layer and varying hidden dimensions. Our selections for these configurations were based on end-to-end models found in other works [25], [51], [54], [57]. Based on prior work [54], the configurations for this evaluation have been known to produce high levels of accuracy. In most occasions, GRANII archives speedups greater than WiseGraph and DGL at varying levels.

E. Sampling with GRANII

Sampling is a common technique used in GNNs to enhance the generalizability of predictions [18] and to achieve better performance. In order to check the sensitivity of GRANII’s decision to neighborhood sampling, we evaluated both compositions it discovered for GCN and GAT using 10 random neighborhood samples of sizes 1000, 100, and 10 on the H100 machine for the *mycielskian17* (MC) graph on DGL. We used embedding sizes (32, 256), (1024, 2048) for GCN, GAT respectively. We selected these embedding sizes to show clear changes using the best configuration for our selected sampling sizes.

As seen in Figure 9, the different random samples of the same sampling size exhibit minimal variation in runtime. Upon

evaluating the decisions made by GRANII with five more sampling sizes (ranging from 5000 to 5), GRANII only made incorrect decisions when there was little benefit in selecting one composition over the other. These evaluations show that a single call to GRANII can be assumed across sampled graphs without the need to inspect the sampled graphs separately and re-run the underlying cost models.



(a) Sampling for GCN - (32, 32) (b) Sampling for GAT - (1024, 2048)

Fig. 9: Scatter plots illustrating the effects of sampling for MC on H100. Each composition is run on 10 randomly sampled sub-graphs. (a, b) is the embedding sizes. Black lines show the median runtime.

F. Using GRANII Across Multiple Layers

For a multi-layer GNN, GRANII can simply select the best composition for each layer using its lightweight cost models. We observe consistent speedups

TABLE V: Runtimes with varying number of layers for Reddit (hidden-dim: 32)

Layers	Execution time (ms)	
	WiseGraph	GRANII (Speedup)
2	48.3	9.4(5.14×)
3	72.2	13.9(5.19×)
4	96.2	18.4(5.22×)
8	192.1	36.5(5.26×)

using this technique against WiseGraph for a varying number of layers, as shown in Table V.

This is possible because sparsity, which is a function of the input graph/adjacency matrix, typically does not change across layers in GNNs. While there can be specific classes of GNN models and operations that change the input graph’s sparsity across layers, the GNN models that we evaluate do not possess this characteristic. However, even with changing sparsity across layers, GRANII can make decisions at each layer using only its online component. Note that such changes in sparsity are unlikely to be significant across iterations, similar to sampling (which we show that it works well in GRANII in Section VI-E). Thus, a decision made in the first iteration can still be used across other iterations, amortizing the overhead.

G. Accuracy of GRANII’s Cost Models

It is non-trivial to select the best primitive composition, as it depends on multiple factors. To perform a deeper analysis, we compare the geomean speedup of the optimal configuration (*Optimal*), with the decisions made by GRANII using its cost models. In addition, we evaluate decisions made by an oracle that selects the best primitive composition independently considering only the model configurations such as the input and output embedding sizes (*Config.*), underlying hardware architecture (*HW*), the input graph (*Graph*), or the underlying baseline system (*Sys.*) for all five models we used in our evaluations. For example, the *Graph* oracle selects *recompute* as the best for GAT on a given graph if *recompute* is beneficial for a majority of the input configurations (ranging over configurations, target hardware, and systems).

The results in Table VI show that GRANII makes more accurate decisions than the evaluated heuristics. The *Config.* oracle performs the best among the oracle models. Nonetheless, this shows the need to consider multiple complex factors when choosing the best primitive composition as GRANII does.

TABLE VI: Speedup from GRANII vs. other heuristics

GNN	Optimal	GRANII	Config.	HW	Graph	Sys.
GCN	1.98×	1.88×	1.88×	1.64×	0.94×	1.71×
GIN	1.22×	1.18×	1.15×	1.15×	1.03×	1.11×
SGC	1.87×	1.82×	1.76×	1.69×	1.01×	1.68×
TAGCN	1.57×	1.53×	1.48×	1.45×	0.89×	1.31×
GAT	1.46×	1.45×	1.22×	1.34×	1.37×	1.39×

VII. RELATED WORK

Input Sensitive Systems. There has been limited work on systems strictly dedicated to optimizing GNNs based on input. GNNAdvisor [53] is one such system that, based on the input graph, uses a set of handcrafted functions to identify optimization opportunities. Its optimizations are tailored towards GPU-based sparse executions, and the thresholds are heuristically set instead of being learned. Ideally, GRANII’s techniques would be orthogonal to GNNAdvisor. However, GRANII cannot be directly applied to GNNAdvisor without significant engineering effort, as its underlying implementation only has limited sparse-dense primitives. uGrapher [69] is similar to GRANII as it optimizes GNNs using an input-aware machine learning model to make decisions. In addition, works such as [7], [8] use inspector-executor-based executions to perform low optimizations such as vectorization. In particular, [8] symbolically analyzes sparse codes at compile time, similar to GRANII’s offline stage. However, compared to these works, GRANII performs optimizations at a higher level, such as exploring different re-association choices. [60] considers the interplay between sparse and dense computation but focuses on deciding solely based on the model configurations. Instead, GRANII jointly considers model configurations and the input graph for a target system and hardware.

There have been multiple input-aware systems applicable to GNNs that select the underlying sparse data format. [44] proposes using a machine-learning model to predict the best data representation for sparse operations in GNNs. Similarly, WISE [63] proposes a machine-learning solution to select the best data representation for SpMV. Works such as ASpT [21] and LAV [62] present input-aware sparse optimizations that introduce new sparse representations and are coupled with specialized executions. Focusing on graph operations in general, [40] proposes a solution quite similar to GRANII, where it presents a learned solution that performs sparse optimizations based on the input graph. However, this and the aforementioned solutions only optimize sparse computations. By not considering the input-aware interplay of sparse and dense in the context of GNNs, multiple optimization opportunities are missed, as identified by GRANII.

GNN Optimizations. Graphiler [58], and SeaStar [57] present compiler-based solutions that allow GNN models written in

user-defined functions to be converted into highly optimized code based on the computations specified. [65] achieves significant speedups through operator reordering, kernel fusion, and re-computation of training results. WiseGraph [25] uses graph partitioning to optimize GNN executions. FreshGNN [24] is a general-purpose GNN mini-batch-training framework using caching for optimizations. FusedMM [45] and Graphite [17] also present GNN kernel fusions, albeit specifically on CPU. Although none of the aforementioned systems perform any input-aware optimizations, as GRANII selects the best primitive composition given the input, all the optimizations presented in such systems can compose with GRANII.

Sparse Tensor Systems. In addition to systems that optimize for GNNs specifically, kernels generated by sparse tensor systems could also be used to optimize GNN executions. General sparse tensor systems such as TACO [33] and SparseTIR [61] can perform schedule-based transformations in a single sparse kernel. However, optimization opportunities that come with kernel selection require optimizing on the end-to-end GNN workload like GRANII does. Going beyond the boundary of a single kernel, systems such as [1], [9], [12], [29] perform loop ordering optimizations across multiple adjacent kernels. SpEQ [34] and Mosaic [3] are examples that exploit existing high-performance libraries whenever possible when executing sparse implementations. However, these systems do not perform any input-aware primitive composition selections across all associative choices as GRANII does.

VIII. CONCLUSION

In this work, we propose a system to exploit the input sensitivity of different primitive sparse-dense compositions in GNN models. Our system, GRANII, is capable of traversing different primitive compositions and reasoning about their optimality based on rules and data-driven methods that inspect the input. Input parameters we consider include the input graph and embedding sizes for a target hardware architecture and GNN system. GRANII allows users to attain speedups compared to the default execution in both WiseGraph and DGL, over a wide range of graphs and embedding sizes in both GPUs and CPUs.

IX. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and our shepherd for their guidance. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA and by NSF under grant CCF-2316233.

REFERENCES

- [1] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022*, page 269–285, New York, NY, USA, 2022. Association for Computing Machinery.
- [2] Srijani Bagchi, Anasua Sarkar, and Ujjwal Maulik. Topology adaptive graph convolution network with heterogeneous entities for predicting adverse events from drug-drug-interactions. *bioRxiv*, pages 2022–05, 2022.

- [3] Many Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. Mosaic: An interoperable compiler for tensor algebra. *Proceedings of the ACM on Programming Languages*, 7, June 2023.
- [4] Maya Bechler-Speicher, Ben Finkelshtein, Fabrizio Frasca, Luis Müller, Jan Tönshoff, Antoine Siraudin, Viktor Zaverkin, Michael M. Bronstein, Mathias Niepert, Bryan Perozzi, Mikhail Galkin, and Christopher Morris. Position: Graph learning will lose relevance due to poor benchmarks, 2025.
- [5] Jou-An Chen, Hsin-Hsuan Sung, Ruifeng Zhang, Ang Li, and Xipeng Shen. Accelerating gnns on gpu sparse tensor cores through n:m sparsity-oriented graph reordering. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPOPP '25, page 16–28, New York, NY, USA, 2025. Association for Computing Machinery.
- [6] Tianqi Chen and Carlos Guestrin. XGBoost. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, aug 2016.
- [7] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. Vectorizing sparse matrix computations with partially-strided codelets. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.
- [8] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.
- [9] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. Runtime composition of iterations for fusing loop-carried sparse dependence. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.
- [10] Timothy A. Davis. Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), dec 2019.
- [11] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.
- [12] Adhitha Dias, Logan Anderson, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. Sparseauto: An auto-scheduler for sparse tensor computations using recursive loop nest restructuring, 2024.
- [13] Changxu Dong, Dengdi Sun, Zhenda Yu, and Bin Luo. Multi-view brain network classification based on adaptive graph isomorphic information bottleneck mamba. *Expert Systems with Applications*, 267:126170, 2025.
- [14] Jian Du, Shanghang Zhang, Guanhang Wu, Jose M. F. Moura, and Soumya Kar. Topology adaptive graph convolutional networks, 2018.
- [15] Tianchuan Du, Keng-hao Chang, Paul Liu, and Ruofei Zhang. Improving taxonomy-based categorization with categorical graph neural networks. In *2021 IEEE International Conference on Big Data (Big Data)*, pages 1015–1022, 2021.
- [16] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.
- [17] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. Graphite: Optimizing graph neural networks on cpus through cooperative software-hardware techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 916–931, New York, NY, USA, 2022. Association for Computing Machinery.
- [18] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.
- [19] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.
- [20] Md Abul Hasnat, Somayeh Asadi, and Negin Alemazkoor. A graph attention network framework for generalized-horizon multi-plant solar power generation forecasting using heterogeneous data. *Renewable Energy*, 243:122520, 2025.
- [21] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPOPP '19, page 300–314, New York, NY, USA, 2019. Association for Computing Machinery.
- [22] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.
- [23] Yang Hu, Wenxi Wang, Sarfraz Khurshid, Kenneth L. McMillan, and Mohit Tiwari. Fixing privilege escalations in cloud access control with maxsat and graph neural networks. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 104–115. IEEE Press, 2024.
- [24] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. Freshgnn: Reducing memory access via stable historical embeddings for graph neural network training. *Proc. VLDB Endow.*, 17(6):1473–1486, 2024.
- [25] Kezhao Huang, Jidong Zhai, Liyan Zheng, Haojie Wang, Yuyang Jin, Qihao Zhang, Runqing Zhang, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Wisegraph: Optimizing gnn with joint workload partition of graph and operations. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1–17, New York, NY, USA, 2024. Association for Computing Machinery.
- [26] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and bridging the gaps in current gnn performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '21, page 119–132, New York, NY, USA, 2021. Association for Computing Machinery.
- [27] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proceedings of Machine Learning and Systems 2020*, pages 187–198. 2020.
- [28] Wengong Jin, Kevin Yang, Regina Barzilay, and Tommi Jaakkola. Learning multimodal graph-to-graph translation for molecule optimization. In *International Conference on Learning Representations*, 2019.
- [29] Raghavendra Kanakagiri and Edgar Solomonik. Minimum cost loop nests for contraction of a sparse tensor with a tensor network. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '24, page 169–181. ACM, June 2024.
- [30] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 387–400, 2021.
- [31] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units, 2021.
- [32] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.
- [33] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.
- [34] Avery Laird, Bangtian Liu, Nikolaj Björner, and Maryam Mehri Dehnavi. Speq: Translation of sparse codes using equivalences. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.
- [35] Guannan Li, Le Zhang, Lingzhi Yang, Hang Hu, Chengliang Xu, Liangzhen Jiao, Donghua Liu, Chenglong Xiong, and Jiahui Deng. Model interpretation and interpretability performance evaluation of graph convolutional network fault diagnosis for air handling units. *Journal of Building Engineering*, 103:112048, 2025.
- [36] Zhuoran Li, Lianshan Yan, Hua Li, and Yu Wang. Environmental factors-aware two-stream gcn for skeleton-based behavior recognition. *Mach. Vision Appl.*, 36(2), January 2025.
- [37] Chen Liang, Ziqi Liu, Bin Liu, Jun Zhou, Xiaolong Li, Shuang Yang, and Yuan Qi. Uncovering insurance fraud conspiracy with network learning. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, page 1181–1184, New York, NY, USA, 2019. Association for Computing Machinery.
- [38] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. Heterogeneous graph neural networks for malicious account detection. *CIKM '18*, page 2077–2085, New York, NY, USA, 2018. Association for Computing Machinery.
- [39] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: parallel deep neural network computation

- on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 443–458, 2019.
- [40] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A pattern based algorithmic autotuner for graph processing on gpus. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, PPOPP '19*, page 201–213, New York, NY, USA, 2019. Association for Computing Machinery.
- [41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [42] Phuong Pho and Alexander V Mantzaris. Regularized simple graph convolution (sgc) for improved interpretability of large datasets. *Journal of Big Data*, 7(1):91, 2020.
- [43] Oleg Platonov, Denis Kuznedelev, Michael Diskin, Artem Babenko, and Liudmila Prokhorenkova. A critical look at the evaluation of gnns under heterophily: Are we really making progress?, 2024.
- [44] Shenghao Qiu, Liang You, and Zheng Wang. Optimizing sparse matrix multiplications for graph neural networks. In Xiaoming Li and Sunita Chandrasekaran, editors, *Languages and Compilers for Parallel Computing*, pages 101–117, Cham, 2022. Springer International Publishing.
- [45] Md Rahman, Majedul Haque Sujon, Ariful Azad, et al. Fusedmm: A unified sddmm-spmmm kernel for graph embedding and graph neural networks. In *35th Proceedings of IEEE IPDPS*, 2021.
- [46] Zhihao Shi, Jie Wang, Fanghua Lu, Hanzhu Chen, Defu Lian, Zheng Wang, Jieping Ye, and Feng Wu. Label deconvolution for node representation learning on large-scale attributed graphs against learning bias, 2023.
- [47] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514. USENIX Association, July 2021.
- [48] Wen Tong and Russ B. Altman. Graph convolutional neural networks for predicting drug-target interactions. *Journal of Chemical Information and Modeling*, 59(10):4131–4149, 10 2019.
- [49] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.
- [50] Kai Wang, Shuaiyi Lyu, Bailing Wang, and Yongzheng Zhang. Anomaly detection via semantically conjugate view learning on industrial temporal data. *IEEE Internet of Things Journal*, 12(11):15479–15490, 2025.
- [51] 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 Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs, 2019.
- [52] Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth McMillan, and Risto Miikkilainen. Neuroback: Improving cdcl sat solving using graph neural networks, 2024.
- [53] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531. USENIX Association, July 2021.
- [54] Lanning Wei, Huan Zhao, Zhiqiang He, and Quanming Yao. Neural architecture search for gnn-based graph classification. *ACM Trans. Inf. Syst.*, 42(1), August 2023.
- [55] Jaeyeon Won, Charith Mendis, Joel Emer, and Saman Amarasinghe. Waco: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2023.
- [56] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr. au2, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks, 2019.
- [57] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: Vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21*, page 359–375, New York, NY, USA, 2021. Association for Computing Machinery.
- [58] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. Graphiler: Optimizing graph neural networks with message passing data flow graph. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 515–528, 2022.
- [59] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.
- [60] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Characterizing and understanding gcns on gpu. *IEEE Computer Architecture Letters*, 19:22–25, 2020.
- [61] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetrir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023*, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.
- [62] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. Speeding up spmv for power-law graph analytics by enhancing locality and vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.
- [63] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. Wise: Predicting the performance of sparse matrix vector multiplication with machine learning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPOPP '23*, page 329–341, New York, NY, USA, 2023. Association for Computing Machinery.
- [64] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '18*, page 974–983, New York, NY, USA, 2018. Association for Computing Machinery.
- [65] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. 4:467–484, 2022.
- [66] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018.
- [67] Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. Learning on large-scale text-attributed graphs via variational inference, 2023.
- [68] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '18*, page 94–108, New York, NY, USA, 2018. Association for Computing Machinery.
- [69] Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, Xiangwen Liu, and Hanqing Wu. ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023*, page 878–891, New York, NY, USA, 2023. Association for Computing Machinery.