GALA: A High Performance Graph Neural Network Acceleration LAnguage and Compiler

DAMITHA LENADORA, University of Illinois at Urbana-Champaign, USA
NIKHIL JAYAKUMAR, University of Texas at Austin, USA
CHAMIKA SUDUSINGHE, University of Illinois at Urbana-Champaign, USA
CHARITH MENDIS, University of Illinois at Urbana-Champaign, USA

Multiple frameworks and optimizations have been proposed for accelerating Graph Neural Network (GNN) workloads over the years, achieving sizable runtime performance improvements. However, we notice that existing systems usually explore optimizing either at the intra-operator level or at the inter-operator level, missing synergies that exist due to their compositions. Further, most existing works focus primarily on optimizing the forward computation of GNNs, often overlooking opportunities for training-specific optimizations.

To exploit these missed optimization opportunities, we introduce GALA, a domain-specific language (DSL) and a compiler that allows composing optimizations at different levels. The GALA DSL exposes intra-operator transformations as scheduling commands, while we introduce novel inter-operator transformations as part of the compiler. The composition of these transformations is made possible through the introduction of two novel intermediate representations (IR) in the GALA compiler that tracks and composes transformations at both the intra- and inter-operator levels. Further, the IRs maintain a global view of the GNN program, including its training process. This allows us to introduce training-specific transformations to aggressively optimize GNN training. Our evaluations show that GALA achieves a geo-mean speedup of 2.55× for inference and 2.52× for training across multiple systems, graphs, and GNN models. We also show that GALA performs well across different graph sizes and GNN model configurations, as well as allows users to explore different methods of performing similar optimizations leading to different tradeoff spaces.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Domain specific languages**; **Compilers**.

Additional Key Words and Phrases: Graph Neural Networks, Intermediate Representations

ACM Reference Format:
Damitha Lenadora, Nikhil Jayakumar, Chamika Sudusinghe, and Charith Mendis. 2025. GALA: A High Performance Graph Neural Network Acceleration LAnguage and Compiler. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 335 (October 2025), 29 pages. https://doi.org/10.1145/3763113

1 Introduction

Graph neural networks have shown superior prediction performance on graph-structured data that occur in multiple domains, including cosmology [17], biochemistry [7], social networks [57], and finance [43]. GNNs benefit from the relations of the input data represented by the input graph, which is fed as part of the input to a neural network-based machine learning model.

GNN computations are usually expressed as a mix of dense- and sparse-matrix operations. Dense matrix operations are used to perform standard neural network computations. An example

Authors' Contact Information: Damitha Lenadora, University of Illinois at Urbana-Champaign, Urbana, USA, damitha2@illinois.edu; Nikhil Jayakumar, University of Texas at Austin, Austin, USA, nikhiljayakumar@utexas.edu; Chamika Sudusinghe, University of Illinois at Urbana-Champaign, Urbana, USA, chamika2@illinois.edu; Charith Mendis, University of Illinois at Urbana-Champaign, Urbana, USA, charithm@illinois.edu.

ACM 2475-1421/2025/10-ART335
https://doi.org/10.1145/3763113

Table 1. Comparing GALA against other systems. Categories: 1) Sparse tensor systems, 2) GNN frameworks and compilers, 3) The system presented by this paper: GALA. (✗- Not supported, ✓- Fully supported)

Categoty	Examples	Intra Operator Optimizations	Inter Operator Optimizations		① Explore both Inter- and Intra Op. Optimizations	③ Different Implementations of Transformations
			Operator Reorder & Selection	② Training Specific		
1	SparseTIR, dgSparse, TACO	✓	✗	✗	✗	✗
2	DGL, Graphiler, WiseGraph	✗	✓	✗	✗	✗
3	GALA	✓	✓	✓	✓	✓

is generalized matrix-matrix multiplication (GEMM), which is used when updating a node or an edge embedding, using learned weights. Sparse matrix operations are used to perform graph computations, such as aggregating features of nodes in the graph. [45] showed that these operations can be lowered down to generalized sparse-matrix dense-matrix multiplications (SpMM) or sampled dense-dense matrix multiplications (SDDMM). As these sparse operations are not natively supported by generic deep-learning (DL) frameworks (such as PyTorch [38]), researchers have developed specialized frameworks on top of them to perform GNN computations. Popular examples include PyTorch Geometric (PyG) [18] and Deep Graph Library (DGL) [45]. These frameworks directly map operations written using the message-passing abstraction [45] to sparse primitives. Although these allow users to easily implement GNNs, many optimization gaps still remain.

Thus, many subsequent works have been proposed to improve the runtime performance of GNNs. We summarize these approaches, comparing the optimizations they propose in Table 1. We broadly categorize them into two classes: ones that perform optimizations within individual operators (intra-operator, e.g., within an SpMM) and ones that perform optimizations between operators (inter-operator). Tensor compilers such as TACO [30] and SparseTIR [54] are proficient in producing highly efficient sparse operations (intra-operator) to enable GNN acceleration. Frameworks such as dgSparse [56] also optimize at an intra-operator level, as they select the best execution for a sparse operation among multiple implementations. In comparison, specialized GNN compilers such as Graphiler [51] perform operator-selection optimizations (inter-operator) that eliminate redundancies. Although these existing systems provide notable speedups, we observe that they miss the following optimization opportunities that lead to substantial performance improvements.

Opportunity ①: Synergistic Optimization at both Inter-operator and Intra-operator Levels. Composing optimizations that function on different levels of a GNN computation leads to a much larger optimization space that can yield significant and synergistic performance benefits. For example, manipulating the underlying sparse representation while also changing the workload per thread for an aggregation operation (intra-operator) and then selecting the order of operations (inter-operator) for an end-to-end GNN computation lead to sizable performance improvements (more than 1.67× speedup compared to the application of each transformation class in isolation as seen in Figure 3(b)). However, existing GNN or sparse tensor compilers do not enable compositions at both the intra-operator and inter-operator levels, as their intermediate representations (IRs) do not capture a holistic view of a GNN model. For example, the iteration graphs of TACO [30] can generate a primitive for different sparse formats but cannot perform operator reordering and selection optimizations that the SeaStar representation of [50] enables, and vice-versa. To go beyond existing systems, an ideal GNN system should capture program information at different levels.

Opportunity ②: Optimizations Specific for Training. Most optimizations proposed for GNNs [19, 24, 41, 47, 50, 60] only consider the forward pass and do not perform *targeted* optimizations for the backward pass necessary for training. For example, consider a scenario of training on a directed graph input. The backward pass does not use the same graph as the forward pass, but rather, its transpose. This results in a graph with a different non-zero distribution and, thus, different optimal optimization choices due to the input-sensitive nature of graphs [58]. Further, optimizations such as moving invariant computations out of the model training loop (e.g. graph aggregations) are typically not enabled by other systems (leading to speedups of 1.71× over optimizations applicable to the forward pass as seen in Figure 4). Automatically enabling these and other novel training optimizations (Section 6.2.1) requires a GNN system to have a representation that captures an end-to-end global view of the model unrolled across layers and its surrounding training code.

Opportunity ③: Selecting between Different Methods of Implementing a Transformation. There can be different methods for implementing the same optimization transformation. For example, consider *neighborhood sampling* in a graph, where some number of neighbors are selected for each node. This transformation can be performed by either preprocessing the underlying graph to create a subgraph (data-sampling) or altering the underlying kernel to sample the graph on the fly (kernel-sampling). The former transforms the underlying data with an additional preprocessing overhead and memory, while the latter directly transforms a GNN's computations. The former is suitable for faster GNN training, while the latter is suitable under memory-constrained scenarios. The existing GNN systems do not provide flexibility in selecting such alternative transformations. To explore different trade-off spaces, an ideal GNN system should expose such choices to the user.

Our Solution. To exploit the above opportunities, we introduce GALA, a domain-specific language (DSL) and compiler for programming and optimizing GNNs. We use GALA's language as a means of obtaining the necessary information to populate the two novel IRs (Data-IR and Compute-IR) that we introduce as part of GALA for a given GNN model. These separate IRs enable the tracking of transformations that mutate either data or computations separately to exploit **Opportunity** ③. The IRs themselves and the different types of relations we track between the IR nodes (Section 6.1) facilitate performing compositions at different levels to exploit **Opportunity** ①. Furthermore, the IRs allow GALA's compiler to maintain a global view of both the forward and the backward pass to enable context-aware training optimizations to exploit **Opportunity** ②. This enables the GALA compiler to perform novel global optimizations (e.g., context-aware optimizations, Section 6.2.1) not possible with existing GNN frameworks. In addition, note that we design GALA's language with a separate (a) algorithm (which specifies the execution logic and is similar to common GNN APIs [18, 45]) and (b) schedule (to expose optimizations) for better exploration of transformation choices across different graph inputs. In addition to the automatic inter-operator optimizations that GALA can perform using its IRs, the schedules of GALA expose intra-operator optimizations with complex input-sensitive behavior for users to optimize. We make the following contributions in this paper.

- We present GALA, a DSL and compiler to implement and optimize GNNs. The GALA language is used to specify the computations (algorithm) of a GNN and any intra-operator optimizations (schedule) applied (Section 5). The GALA compiler composes both intra- and inter-optimizations when producing the final executable.
- We introduce two novel IRs as part of the GALA compiler – Data-IR and Compute-IR – to separately track data-level and compute-level transformations at different granularities (inter-operator and intra-operator). (Section 6.1)
- We introduce novel compiler-driven GNN optimizations (e.g. training-aware subgraphs) and provide algorithms for transforming the IRs to enable these optimizations, other optimizations specified by the scheduling language, and compositions amongst them. (Section 6.2)
- We perform extensive evaluations against four state-of-the-art frameworks and compilers, four GNN models, and six graph datasets to show that GALA achieves geo-mean speedups of 2.55× for inference and 2.52× for training across multiple systems. (Section 8)

GALA is publicly available at https://github.com/ADAPT-uiuc/GALA-GNN-Acceleration-LAnguage.

2 Background

This section describes GNNs and their computations, including the backward pass and a brief overview of applicable optimizations. We then present the computation of a Graph Convolutional Network (GCN), a popular GNN model we use as our running example.

2.1 Graph Neural Network Computations

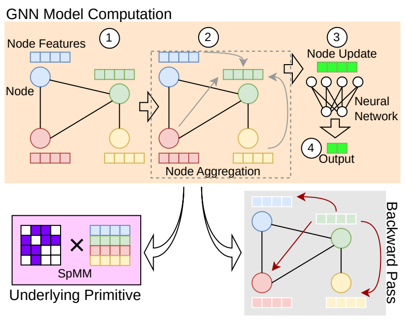


Fig. 1. GNN model computations, primitives, and backward pass

Graph Neural Networks (GNNs) combine graph operations with established neural network operations. Figure 1 shows the basic building blocks of a GNN model computation. Typically, inputs to a GNN (①) are the input graph and its node features. These features are usually represented as low-dimensional vectors, which are called embeddings. In ①, the node features for each node are represented with matching colors. Next, GNNs use these node features in graph operations, such as the node feature *aggregation* shown by ②. During this process, node features are passed along as messages from source to destination and aggregated into a single message using some aggregation function. Finally (③), these aggregated messages are passed through neural network operations (e.g., a fully connected feedforward layer) to arrive at the final *updated* messages, which are used to update the corresponding nodes' features. This process of aggregation and update can be applied multiple times, leading to a multi-layer GNN. In addition to aggregate and update operations, certain GNN models can require other operations. These include edge-based attention calculation operations and normalization operations.

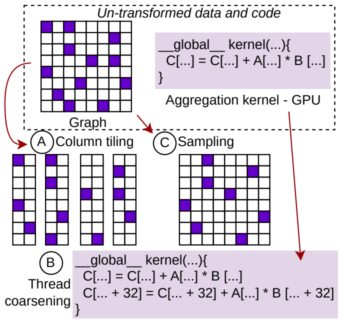
Kernels and Representations. GNNs use sparse matrix representations and operations to perform the relevant graph operations. For example, the aggregation operation in ② is lowered to a sparse matrix dense matrix multiplication (SpMM) [45]. This is essentially matrix multiplication, with one input matrix being sparse. Multiple formats, including compressed sparse row (CSR) and Ellpack (ELL), can be used to represent the adjacency matrix of the input graph as a sparse matrix. In addition to SpMM, another matrix primitive commonly found in GNNs is sampled dense-dense matrix multiplication (SDDMM). SDDMM is used for edge-based aggregations and is intuitively a matrix multiplication between two matrices, where the output is masked by a sparse matrix to produce only a subset of the total multiplication. The generalized versions of these two operations can be used to perform a majority of the graph operations in GNNs, as shown by [45]. In addition to graph operations, GNNs also contain standard neural network operations. For example, the update operation in ③, is lowered to a general matrix multiplication (GEMM) between the said node features and the corresponding learned weights. Broadcasting values of a vector along the row-dimensions of a matrix (which we term as row broadcast) can be taken as another example, where it is the lowered operation for certain node feature normalizations.

Backward Pass. As GNNs are machine learning models, they must undergo a training process where their learned weights are updated. This process uses a backward pass over the model, which computes a gradient to update the model's learned weights. Note that a graph operation in the forward pass has a corresponding graph operation in the backward pass. For example, the backward pass of `SpMM(`A`, `H`)` must compute `SpMM(`A^T`, `dZ`)` to get the derivative relative to the input node features. Here, A represents the adjacency matrix of the input graph, A^T represents the transposed input graph (to emulate the message traveling backward), H represents the input node features, and dZ is the derivative of the output. This computation is depicted in Figure 1, where the backward pass operation for the node aggregation in ② involves sending the derivative in *reverse* of the initial communication between nodes.

2.2 Example Optimizations and Transformations Used in GNNs

This section gives examples of different transformations common in the GNN domain. Figure 2 depicts the transformations we list below. We reiterate that these optimizations are commonplace, and we introduce them as background to facilitate the discussions in the rest of the paper.

Transforming the Underlying Data: Column Tiling Ⓐ. We define the *column tiling* transformation as a pre-processing step that transforms a graph into a set of sub-graphs. Each sub-graph only contains edges where the nodes represented by the columns of its adjacency matrix fall within a specific range. For example, as shown by Ⓐ in Figure 2, the adjacency matrix of the original untransformed graph is broken down into sub-graphs, each containing two columns. This transformation results in better data access patterns, similar to [22].



```
__global__ kernel(...){
  C[...] = C[...] + A[...] * B [...]
}
```

```
__global__ kernel(...){
  C[...] = C[...] + A[...] * B [...]
  C[... + 32] = C[... + 32] + A[...] * B [... + 32]
}
```

Fig. 2. GNN transformations

Transforming Computation Kernels: Thread Coarsening Ⓑ. Thread coarsening is a common GPU optimization where more work is assigned to a single thread than an operation's parallelizability. For example, in Ⓑ in Figure 2, the transformed kernel aggregates two values instead of one, improving memory access patterns and reducing multi-threading overheads.

Graph Sampling Ⓒ. Sampling is an approximate transformation regularly used in the GNN domain that enables better performance and significant accuracy boosts [20] (also shown by our results in Section 8.5.4). The latter is especially true for better generalization when performing inductive learning, where a GNN model is trained without utilizing the full graph. Note that there are different methods of performing this transformation, such as changing the underlying graph as shown by Ⓒ in Figure 2 or even by changing the kernel operating on the graph to limit its computation to the sampled number of edges per node.

2.3 Graph Convolutional Network (GCN)

Graph Convolutional Network (GCN) [29] is a simple yet popular GNN model. The inference computation for the $(l)^{th}$ GCN layer is presented by the equation, $H^{(l)}=\sigma(\tilde{D}^{-\frac{1}{2}}\cdot\tilde{A}\cdot\tilde{D}^{-\frac{1}{2}}\cdot H^{(l-1)}\cdot W^{(l)})$. We use this model as our running example in Section 3. Here, $\tilde{A}$ represents the input graph with self-edges as an adjacency matrix. $\tilde{D}$ represents the degree matrix of the input graph. $H^{(l)}$, $W^{(l)}$ represent the node feature embeddings and the learning weights for the l^{th} layer in the GNN model.

3 Motivation for a Compiler-Based Solution

GALA was motivated by two observations: (a) the composition of optimizations at different levels leads to better GNN performance, and (b) context-aware global optimization leads to better end-to-end GNN executions. These opportunities, missed by prior work, can be captured by GALA's DSL and compiler-based approach through its novel IR and transformations. In the remainder of this section, we describe these points and show their benefits through the speedups observed in GALA.

3.1 Compositions of Optimizations at Different Levels

GALA can compose GNN optimizations at both the (a) intra-operator level and the (b) inter-operator level. Intra-operator optimizations focus on improving the speedup of a single operator. An example of this is applying thread coarsening for a node-aggregation operator (SpMM) on GPUs. On the other hand, inter-operator optimizations, such as operator reordering and selection, focus on improving the end-to-end speedups of an entire program. Composing these optimizations can lead to significant speedups unseen in isolation.

The synergistic benefit of performing both intra-operator (*thread coarsening, column tiling*) and inter-operator optimizations (*operator reordering*) is shown in Figure 3 for two graph datasets:

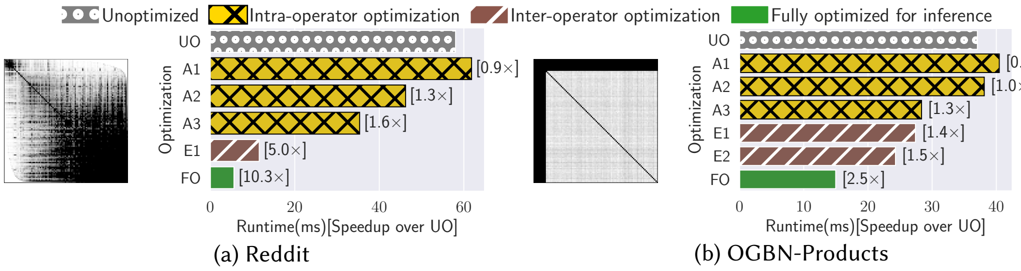


Fig. 3. Graph's adjacency matrix, and runtime breakdown with optimizations. We use a 2-layer GCN with a hidden dimension of 32 for our evaluation. Breakdown of the optimizations applied - (UO):unoptimized and using library-based implementations (cuSparse) for sparse operations, (A1):generate sparse kernels that have thread coarsening, (A2):A1+column tiling, (A3):A2+augment the sparse kernels to operate on unweighted graphs (as the input graph is unweighted), (E1):Reordering operations to have better algorithmic complexity (applied to UO), (E2):E1+sparse operator rewrites (Section 6.2.2), (FO):Fully optimized for inference with all beneficial optimizations applied. A1 to A3 are intra-operator optimizations, and E1 and E2 are inter-operator. Note that we only apply transformations that result in a speedup. This is why E2 is not applied for Reddit.

(a) *Reddit* and (b) *OGBN-Products*. Here, *UO* is an unoptimized 2-layer GCN inference (forward pass only) execution. *A1* to *A3* are gradual additions of intra operator optimizations, while *E1* and *E2* are inter operator optimizations. *FO* is the execution with all intra- and inter-operator optimizations combined and stands as evidence of the synergistic benefits of combining both types of optimizations. For example, the intra-operator optimizations (A3) achieve a speedup of 1.3× on the *OGBN-Products* dataset, while the inter-operator optimizations (E2) achieve a speedup of 1.5×. When both optimizations are combined (FO), the result (2.5×) is greater than the product of both types of optimizations individually (1.95×). Observations such as these motivated us to create a language and compiler that can generate efficient code composed of optimizations at different levels. These compositions are enabled in GALA by its two intermediate representations (IRs) (Section 6.1) and how they interact with one another. This is not achievable in existing systems as they either optimize a single operator and do not retain information about all computations in a GNN (sparse tensor systems such as TACO [30] and SparseTIR [54]), or they use static implementations of primitives without performing any operator optimizations (GNN systems such as Graphiler [51]).

In addition, by generating compositions of different optimization classes, GALA increases the total optimization space beyond what sparse tensor or GNN systems can do in isolation. This increased search space enables more optimization opportunities that existing systems overlook. To explore this optimization space efficiently, we designed our language as separate algorithm- and schedule-languages, inspired by works such as Halide [40] and TACO [30]. This is necessary as the optimal optimization parameters can differ from input to input. The executions of the two graph datasets in Figure 3 are examples, as the input-sensitive optimization parameters, such as the column-tiling factor, differed.

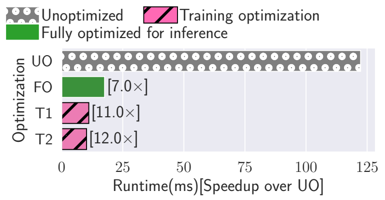
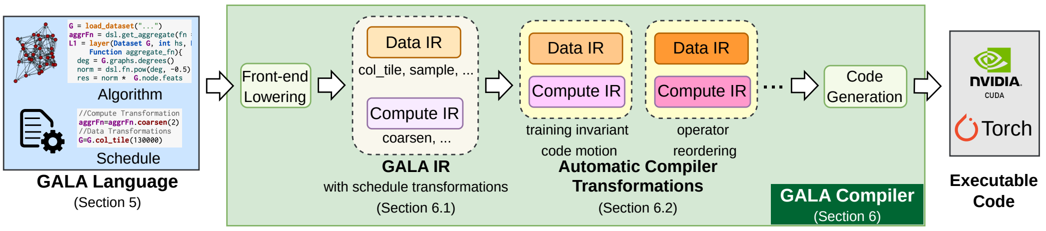


Fig. 4. Speedups of training optimizations for (Reddit). Optimizations-(UO): Unoptimized, (FO): Optimized for inference, (T1): FO+training invariant code motion (Section 6.2.3), (T2):T1+sub-graph for training (Section 6.2.1)

3.2 Context-Aware Global Optimizations

GALA can generate code optimized explicitly for an execution context (training, inference) of a GNN. This is especially true when generating code for training as optimizations such as moving invariant computations across training iterations out of the training loop (Section 6.2.3) and computing on the sub-graph with only the training data (Section 6.2.1) can be applied. Figure 4 shows the example when training-specific optimizations are applied to the *Reddit* graph dataset's execution in Figure 3. The optimizations applicable irrespective of the context (in Figure 3) can speed up training the GNN model by 7×. However, this can be further improved by 1.71×, giving a total speedup of 12× over an unoptimized implementation. Existing GNN systems cannot perform such optimizations, as they build on DNN system pipelines and ignore optimizations specific to GNNs that require knowing the execution context. GALA's ability to have the global view of the entire GNN program through its IRs enables these optimizations.

4 Overview of GALA



```
G = load_dataset("...")
aggrFn = dsl.get_aggregate(fn
L1 = layer(Dataset G, int hs,
     Function aggregate_fn){
  deg = G.graphs.degrees()
  norm = dsl.fn.pow(deg, -0.5)
  res = norm *  G.node.feats
```

```
//Compute Transformation
aggrFn=aggrFn.coarsen(2)
//Data Transformations
G=G.col_tile(130000)
```

Fig. 5. Overview of the GALA's lowering process.

Figure 5 shows the overview of GALA, a domain-specific language and compiler. The latter is where the main innovations lie, while the language is used as a means of generating the intermediate representations (IR) necessary for the compiler. We design the GALA Language taking inspiration from the popular algorithm-schedule separation popularized by Halide [40], and subsequently used in languages such as TACO [30] and Graphit [58]. GALA's algorithm language (Section 5.1) specifies the GNN computation, while the scheduling language Section 5.2) specifies a subset of optimization transformations (mostly intra-operator) applied to this computation when creating the final executable. When generating the final executable, GALA first lowers the front-end language's algorithm and schedules (detailed in Section 6.1.4) into GALA's novel IRs: (a) Data-IR (DIR, Section 6.1.1) and (b) Compute-IR (CIR, Section 6.1.2). The former tracks data dependencies and transformations, while the latter tracks the order of computations and transformations on compute operations. Any transformation specified in the schedules of GALA is directly represented in either of the two IRs after the front-end lowering. Following the initial lowering, GALA performs automatic novel inter-operator transformations on these IRs based on the information of the GNN retained by the IRs (Section 6.2). The burden on the user is reduced by this *hybrid* approach of GALA as it can perform *automatic domain-specific transformations* that are known to be beneficial. In the end, GALA generates the target code with composed intra- and inter-operator optimizations based on the fully transformed IRs (Section 6.3).

```
G = load_dataset("...")
aggrFn = dsl.get_aggregate(fn = dsl.fn.mul_sum)
L1 = layer(G, hs, nonln_fn, aggregate_fn){
  deg = G.graphs.degrees()
  norm = dsl.fn.pow(deg, -0.5)
  res = norm *  G.node.feats
  res = aggregate_fn(G.graphs, res)
  res = dsl.nn.ffn(res, out=hs)
  res = norm * res
  G.node.feats = nonln_fn(res)
}
M1 = model(G, nonln_fn, aggregate_fn){
  l1 = L1(G,32,nonln_fn,aggregate_fn)
  l2 = L1(l1,G.labels.size(),null,aggregate_fn)
}
m1 = M1(G, dsl.nln.ReLU, aggrFn)
m1.train(loss=dsl.nn.RMSE, optimizer=dsl.nn.
    ADAM, iters=100, test_step=5)
```

Fig. 6. Implementation of a 2-layer GCN model in GALA's algorithm language

5 The GALA Domain Specific Language

In this section, we elaborate on the (a) Algorithm and (b) Schedule languages of GALA.

Table 2. A subset of the programming API for GALA's algorithm language

Syntax	Description
Graph dataset operations	
· `load_dataset(path)`	Load the graph dataset in `path`. Assumes input is multiple independent graphs.
· `degrees`	Get the degrees of nodes in the graphs from the dataset, as a tensor.
GNN computations	
· `get_aggregate(semi)`	Gets the aggregate function based on the semiring function passed by `semi`.
· `aggr_fn(g, mtx)`	Performs aggregation on the input graph list `g` and tensor `mtx`.
· `non_lnl(inp)`	Performs the non-linear function on `inp`.
· `pow(mtx, p)`	Get the `p` power of the tensor `mtx`.
Modeling a GNN and executions	
· `layer(....)`	Defines a composition of GNN operations. (e.g. GCN [29] or GAT [44])
· `model(....)`	Defines an end-to-end GNN model, which is composed of GNN layers. (e.g. a 2-layer GNN model made up of 2 GCN layers)
· `train(lossFn, opt, epochs, val_step, test_step)`	Trains the GNN model using the loss function `lossFn`, and optimizer `opt` for `epochs` number of epochs. Only compute the validation and test sets of the model at steps defined by `val_step` and `test_step`.

5.1 Algorithm Language

The syntax of GALA's algorithm language augments the syntax found in traditional DNN systems with graph syntax in the form of the message-passing paradigm. We show a subset of the language constructs of GALA's algorithm language in Table 2. An example of a 2-layer GCN model implemented using GALA's algorithm language is shown in Figure 6. The initial 2 lines in this figure perform the initializations necessary for executing the GNN model: (a) loading the input graph and (b) initializing the aggregate function used in the model. Lines 3 – 11 define a GCN layer (Section 2.3), while lines 12 – 15 define the end-to-end model. Both *layers* and *models* are parameterized, allowing configurability while minimizing code repetition. For example, instead of the single aggregation function (`aggrFn`) passed to the model in Figure 6, two separate aggregation functions that perform different computations could be passed through the model to its underlying layers (which also enables layer-wise optimization through GALA's schedules). The final 2 lines initialize and then train the model.

GALA's language follows the GNN model implementation style of DGL [45] with subtle differences. One key difference is how the steps for computing test set results are specified. For example, in Figure 6, the results of the test set are only calculated at every 5th epoch when training. This allows for GALA to optimize the execution by only computing a subset of the total graph computation, unlike existing systems (detailed in Section 6.2.1).

5.2 Scheduling Language

The scheduling language of GALA fulfills two purposes: (a) it allows users to specify intra-operator transformations and (b) aids the compiler to perform inter-operator transformations by allowing users to pass meta-data of the GNN and its input. We list a subset of scheduling commands of GALA in Table 3. These scheduling commands can be applied to variables defined in the algorithm of the model (e.g. `G, aggrFn, res, M1` etc. in Figure 6).

```
//Set meta-data
G=G.set_undirected(true)
G=G.set_unweighted(true)
G=G.feature_size(605)
G=G.label_size(41)
//Compute Transformation
aggrFn=aggrFn.coarsen(2)
//Data Transformations
G=G.col_tile(37000)
```

(a) Reddit

```
//Set meta-data
G=G.set_undirected(true)
G=G.set_unweighted(true)
G=G.is_sparser(true)
G=G.feature_size(100)
G=G.label_size(47)
//Compute Transformation
aggrFn=aggrFn.coarsen(2)
//Data Transformations
G=G.col_tile(1400000)
```

(b) OGBN-Products

Fig. 7. Schedules used to optimize for results in Figure 3

Table 3. A subset of schedule commands provided by GALA's scheduling language. Brackets in the top-right corner of each row indicate whether it is a data-intra-operator transformation, a compute-intra-operator transformation, or meta-data.

`coarsen(factor)`	`[comp]`
Gives the maximum thread coarsening `factor` GALA should generate for the compute kernel specified.	
`sample(size)`	`[data,comp]`
Transforms either a graph (i.e. data) into a sampled sub-graph, or alters a compute kernel to sample during execution. This transformation is not semantically equivalent and is completely *optional.*	
`col_tile(number)`	`[data]`
Column tile a given input based on the `number` given. This can be specific to the data used by a single operator.	
`set_undirected(bool) / set_unweighted(bool)`	`[meta]`
Set if a given graph is undirected/unweighted.	
`is_sparser(bool)`	`[meta]`
Set if a given graph is comparatively sparser. GALA uses this to perform automatic transformations (Section 6.2.2)	

Figure 7 shows the schedules we used to achieve the speedups observed in Figure 3. Note that the two schedules result in two different executions at the lower level, where applying the schedule in Figure 7(a) to the *OGBN-Products* graph results in an execution that takes 4.2× longer than 7(b). In this particular scenario, the schedule of 7(a) applies a more aggressive tiling factor, which increases the overhead of tiling to outweigh its benefits.

The intra-operator transformations enabled by GALA are common optimizations (Section 2.2) with complex input-sensitive behavior (e.g. `col_tile` with different optimal tiling factors that depend on the input). Thus, we decided to expose these transformations at the language level to allow customizations per graph input (e.g. different `col_tile` factors for *Reddit* and *OGBN-Products* in Figure 7). In addition, the schedule also allows users to pass meta-data of the GNN and its input to the compiler (e.g. input feature and label sizes as in Figure 7). The compiler uses this information to aid its automatic transformations, which are always beneficial in terms of runtime. GALA composes both of these transformations (schedule-applied intra-operator and automatic inter-operator) to expose a large optimization space and generate the final executable binary. Next, we discuss some intriguing details of GALA's scheduling language.

Data vs. Compute Transformations. We classify the intra-operator transformations exposed through the schedules of GALA into two categories based on what they impact: (a) data transformations and (b) compute transformations. We use this categorization when designing GALA's IRs that enable efficient code generation and different methods for implementing the same transformation. Data transformations modify the underlying data structure (`col_tile` the input graph), while compute transformations alter the underlying kernels of operations (`coarsen` the kernel used for node aggregation). Note that certain optimizations can be either data or compute transformations. An example would be `sample`, where you can either implement sampling by creating a sampled sub-graph (data transformation) or by altering sparse matrix primitives to function on a subset of edges in the input graph (compute transformation).

Keyword **`backward`** - We introduce the `backward` keyword in the scheduling language to allow users to optimize data structures and operations in the backward pass of a GNN model independently from their counterparts used in the forward pass. This is necessary when operating on GNNs, as the behavior of the components (data and operations of the model) used in the forward pass can be significantly different from the backward pass (detailed in Section 2.1). To maintain transparency, we do not perform any transformations on the components of the backward pass unless specified or always beneficial. Thus, we introduce the `backward` keyword to allow a user to tap into and transform a component's backward-pass counterparts without explicitly writing custom operators and data structures.

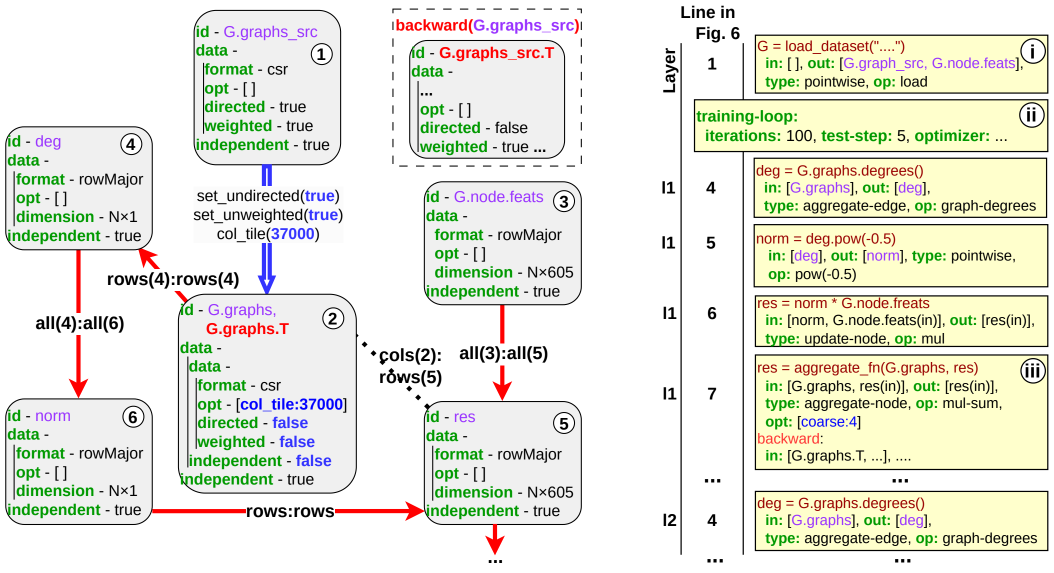
6 The GALA Compiler

This section details the components of the GALA compiler. We first introduce our novel intermediate representations, followed by our novel automatic inter-operator optimizations, and then by the final target code generation.

6.1 Intermediate Representations and their Generation

GALA's separate IRs, Data-IR (DIR) and Compute-IR (CIR), enable more global high-performance optimizations (Section 6.2) and efficient code generation (Section 6.3) without complex analysis. GALA's IRs store the GNN model unrolled across layers to enable optimizations that are aware of the execution context and span multiple layers. These global domain-specific optimizations set GALA's IRs apart from other Sparse Tensor IRs [2, 30] and GNN IRs [49–51]. When generating code, GALA only needs to traverse through the IR in two passes, as the IRs store all the necessary transformations and information for optimizations.

We use the GNN algorithm code written in GALA in Figure 6 along with the schedule for Reddit in Figure 7(a) as the running example to describe the compilation process. Figures 8(a) and 8(b) represent a part of the DIR and CIR that GALA lowers to from this example. In the rest of this section, we will first introduce each IR, elaborate on their design and interactions, and finally detail how they are generated during the lowering process from the front-end code.



(a) Data-IR

(b) Compute-IR. Left columns are the model's layer, and the line in the code in Figure 6

Fig. 8. Intermediate Representations for GALA. Shown for a subset of computations/data in Figure 6 with the scheduling transformations for the *Reddit* Graph in Figure 7(a) (running example)

6.1.1 Data-IR (DIR). DIR is a directed acyclic graph (DAG) with nodes that track (1) information of data objects (e.g., the input graph being undirected), (2) transformations on the data objects specified by the scheduling language of GALA (e.g., performing column tiling on a graph), and edges that track (3) relationships between data objects. The data objects are either sparse or dense tensors used in the GNN execution (e.g., the input graph is a sparse tensor and the node feature tensor is a dense tensor).

Table 4. Edge types and attributes of DIR

	Edge Type	Attribute	Description
⟶	Dependency	*relation*	Input to output dependency
⟹	Transformation	*transforms*	Original to transformed data
· · ·	Association	*relation*	Associations between data

Edge attributes

relation	Relation between data in DIR nodes Can be between rows, columns or all data
transforms	Data transformations performed on a *dir-node*

Table 5. DIR node grammar

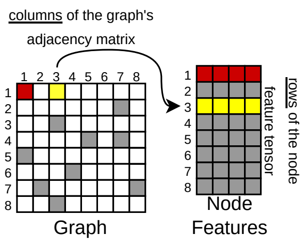
DIR node grammar	
<dir-node>	::= *<dir-id>* *<data-level>*
<data-level>	::= *<data>* *<is-independent>*
<data>	::= *<data-level>* | *<data-attributes>*

We list the grammar of a DIR node in Table 5 and explain important components as follows.

- *data-level* - This contains the *data* non-terminal and *is-independent* flag as components. Note that *data* can become another *data-level*, leading to a hierarchical structure. This hierarchy allows GALA to represent stacks of transformations that create sub-components within the same *dir-node*. An example of this is applying `col_tile` to a dataset with multiple graphs (PPI dataset in [20]). All the graphs of this dataset can be directly loaded by the `load_dataset` operation as it assumes that the initial input can be a list of independent graphs. When `col_tile` is applied to these graphs, dependent sub-graphs must be created within each graph independent of others. Here, the hierarchical structure of *data* in *dir-nodes* can represent these details.
- *is-independent* - Independent graphs and dependent sub-graphs (from `col_tile`) have different behaviors during execution. If graphs depend on one another, any operation that uses this data must aggregate all the dependent results to get the final output. Otherwise, any computation can function independently. The *is-independent* attribute tracks this dependency and is used during the final code generation.
- *data* - This stores attributes of the data represented by the *dir-node*. These attributes include information such as the underlying data format (necessary for code-generation), the transformations performed (in blue in ② in Figure 8(a)), and if the *dir-node* represents a graph, information such as the graph being un-directed or un-weighted. GALA uses this information for optimizations. For example, GALA can use the same graph in both the forward and `backward` computations without transposing for operations such as node aggregation when the directed attribute is false as in ② (Figure 8(a)).

The nodes in the DIR are connected through various edge types to help reduce analysis during later transformations. We list the edge types in Table 4 with examples from Figure 8(a),

- Data *dependencies* between the data represented by DIR nodes (in red). For example, using ② as an input to create ④.
- Data *transformations* (in blue). E.g., transforming the original input graph (①) to ②. The transformations are shown in blue.
- *Association* relations between DIR nodes (a dotted black line). For example, the relation between the graph, represented by ②, and the node features of the graph, represented by ⑤. Each *row* of ⑤ corresponds to a *column* in ② when both are used by ⅲ's aggregation operation.


columns of the graph's adjacency matrix
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8
Graph
1 2 3 4 5 6 7 8
rows of the node feature tensor
Node Features
Fig. 9. Relation between the adjacency matrix of a graph and tensor representing node features

In addition to the edge type, we encode information on how the related data in these DIR nodes map to one another as an edge attribute. Data transformations can have the *transformations* performed from the source to the target as edge attributes, while both data dependencies and associations can have *relations* between connected DIR nodes as edge attributes.

These *relations* are between the matrix dimensions of the data objects represented by the connected DIR nodes. Valid dimensions include `rows`, `columns`, or `all` (i.e. both `rows` and `columns`). The relations are represented as `<node-1-dimension>(<node-1-id>):<node-2-dimension>(<node-2-id>)` in the DIR representation. For example, the relation between the graph (DIR node ②) and its node features (DIR node ⑤) is represented as `cols(2):rows(5)`. We use Figure 9 to elaborate on this. When the graph and its node features are used as inputs of an aggregation operation ((iii) in Figure 8(b)), the columns in the graph's adjacency matrix correspond to the rows in the node features tensor. Thus, the relation between the DIR nodes ② and ⑤ is an `cols:rows` relation. GALA's compiler uses the computation operations specified in GALA's algorithm language and a set of rules to determine the relations between different DIR nodes connected via dependency and association edge types. We detail this in Appendix A. These relations aid in downstream transformations, such as *reordering* the node features if the graph is reordered.

```
  <gnn-execution>   ::= <gnn-oplist> | <gnn-oplist> <training-loop>
   <training-loop>  ::= <training-configs> <gnn-oplist>
      <gnn-oplist>  ::= <cir-node> <gnn-oplist> | <cir-node>
        <cir-node>  ::= <forward-op> | <forward-op> <backward-op>
      <forward-op>  { Stores the operation's type (type), specialization (op), transformations (opt),
     <backward-op>  { and maps to DIR nodes of inputs and outputs (in and out).
```

Fig. 10. GALA's Compute-IR Grammar

6.1.2 Compute-IR (CIR). The CIR of GALA stores the computations and their compute transformations as specified by the GALA language. The CIR also tracks the input and output data objects of computations by mapping them to their corresponding DIR nodes. We list the grammar of CIR in Figure 10 with an example in Figure 8(b). The CIR consists of,

- *gnn-oplist* - A list of *cir-node*s that represents a set of computations in a GNN. Note that this list stores the computations of the GNN model in an unrolled view. This allows GALA to perform optimization and transformation across layers.
- *training-loop* ((ii)) - This stores *training-configs* which are the configurations for training, such as the loss function (necessary for the final code generation), and the *gnn-oplist* of the model when training. Knowing the configurations of training enables GALA to perform a novel optimization when training (Section 6.2.1).

Components of a cir-node. (iii) is an example for a *cir-node* in Figure 8(b). In this particular example, GALA maps this CIR node to the DIR nodes ② and ⑤ in Figure 8(a), as they are inputs for (iii)'s forward pass. A CIR node also stores, (a) the operator *type* (`type`), which can either be a pointwise, aggregate, or update operation affecting either node or edge values (e.g. `update-node`, `aggregate-edge`), (b) its specializations (`op`, e.g. pointwise with `add`, aggregation with the multiply-sum semiring [15] i.e. `mul-sum`, update with ReLU [1]), and (c) the transformations on the operator (`opt`) which were specified in the schedule (e.g. `coarse`, `sample`). GALA's compiler determines a *cir-node*'s operation type and specialization using rules, where we present a subset in Appendix B. For (iii), which is an aggregation operation on node features, initialized in line 2 (with the `mul_sum` function) and used in line 7 of Figure 6, the operation type is *node-aggregation*, with the multiply-sum semiring and the thread coarsening transformation (from Figure 7(a)).

Knowing the operation type is necessary for optimizations that perform rewrites using these operation types. In addition, this type information and transformations on compute operations can be used to identify any GNN computation not naturally supported by LibTorch [38] (used by GALA as a backbone to implement models). For these operations, GALA must generate the necessary

AutoGrad function that contains a backward pass. Thus, when necessary, GALA separately stores information related to the backward pass in the same `cir-node`. This separation between the backward and the forward pass allows GALA to optimize computations in the backward pass independently from the forward pass. This is necessary as the underlying data (especially for directed graphs) and the operation used to compute the result of the backward pass can be significantly different from the forward pass. During code generation, GALA uses the transformations and the detailed operation stored in the *cir-nodes* to generate the final code.

6.1.3 IR Design and Interactions. The nodes in the two IRs of GALA were designed to *separately* capture the supported intra-operator transformations (enabled by the scheduling language of GALA) for both computations (in `cir-node`), and the underlying data (in `dir-node`). This separation avoids unnecessary analysis across both computations and the underlying data, focusing instead on one for intra-operator transformations. However, these IRs heavily interact when performing inter-operator transformations (Section 6.2) as well as the final code generation (Section 6.3). This is where the *edges* between these IR nodes, as well as compute operations to input/output data mapping from the CIR to the DIR are used. Concretely, we highlight a few examples of these interactions.

- In order to carry out the *training-aware subgraph transformation* (Section 6.2.1), GALA must traverse the CIR to identify the number of aggregation operations performed from the final result (across GNN model layers) to create the necessary `dir-nodes` for the subgraphs.
- The *sparse rewrite transformation* (Section 6.2.2), which changes the CIR, will respect the data dependencies of a GNN program tracked by the DIR. This is done by inspecting the mapped input and output `dir-nodes` of a `cir-node`, and edges in the DIR.
- When generating a kernel in the final code generation, GALA uses both the information on the compute operation captured by the CIR, and the information on the underlying data captured by the DIR. For example, different kernels will be generated depending on the underlying format being COO or CSR, or the input graph being column tiled or not. (Section 6.3.1)

6.1.4 IR Generation. Using a syntactic and semantic parser, GALA lowers the front-end language to an abstract syntax tree (AST). GALA then performs a pre-order traversal on this AST while generating a *cir-node* for each operation in the original front-end algorithm of GALA. At each operation, GALA checks their outputs and creates a corresponding *dir-node* in the DIR. During this process, GALA creates the edges in the DIR using the data transformations in the schedule (for transformations), the inputs and outputs of operations (for dependencies), as well as operation-specific knowledge (for associations). This is a rule-based process, which we detail in Appendix A. To give an example, GALA creates ① and ③ in Figure 8 from the compute operation ⓘ. These `dir-nodes` have no relation between them. However, the aggregation operation ⓘⓘⓘ would add a `cols(2):rows(5)` associative edge between the `dir-nodes` ② and ⑤, as they are the graph and node-feature inputs to it. Once the AST has traversed the algorithm of the GNN model, GALA then adds the scheduling transformations for each corresponding *dir-node* or *cir-node* as attributes. Note that for data transformations, GALA creates a new *dir-node* and updates all existing relations between other *dir-node*s as well as mappings to *cir-node*s.

6.2 Automatic Domain-Specific Transformations

GALA's IRs and the information they track allow GALA's compiler to perform novel, domain-specific, inter-operator optimization transformations automatically. These transformations are guaranteed to benefit performance (i.e. runtime) while lessening the user's burden when creating schedules. The GALA compiler automatically composes these transformations along with intra-operator transformations specified in the schedule when generating the final executable code.

6.2.1 Training-Aware Subgraph Creation. In machine learning, the training set is a subset of the dataset used to train a model. In addition to the training set, the validation and test sets are respectively used to validate and test the model. Generally, in machine learning, you only need to compute the training set when training a model. However, in GNNs, aggregation operations prevent training on the training set in isolation. This is because each aggregation operation expands the training set to include the dependent nodes of the training set up till that point. This expansion happens in reverse, starting from the final result of the model. This is depicted in Figure 11, where the training set (in purple) grows with each dependent node (in green). Thus, regardless of whether only 10% of the entire dataset (e.g. OGBN-Products) is used for training, the standard practice in existing GNN systems is to compute using the entire graph dataset and then calculate the loss on the training set. This leads to unnecessary computations that are not used for any outcome (in grey in Figure 11). They can be removed by computing on the sub-graph that only contributes to the final result. Creating the corresponding sub-graph for each aggregation requires knowing the total number of aggregation operations performed from the result up to that point. Thus, lacking this information, existing GNN systems cannot automatically support this optimization.

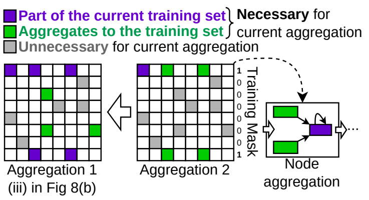

Part of the current training set
Aggregates to the training set
Necessary for current aggregation
Unnecessary for current aggregation
Training Mask
Aggregation 1 (iii) in Fig 8(b)
Aggregation 2
Node aggregation

Fig. 11. Sub-graphs calculated in reverse from the training mask

GALA's IR enables this optimization as it allows GALA to know the context of the workload through the training loop and can track the total number of aggregations across all model layers. This is done by traversing the CIR nodes (which stores the GNN model unrolled across layers) in reverse while checking their operation types. Considering the running example, for the aggregation operation $\textcircled{\scriptsize iii}$ in Figure 8(b), GALA is aware of a second node-aggregation operation that occurs later on through the CIR. Thus, it creates a sub-graph that includes the dependent nodes of a single aggregation, as shown in Figure 11. The sub-graphs created by GALA in this manner are added to the DIR as additional nodes and linked to the related aggregation operations in the CIR. Note that this optimization creates two sets of sub-graphs (for the forward and backward pass) for each aggregation in the model. As this results in additional memory consumption, GALA exposes an optimization knob to turn off this transformation when operating on limited memory.

6.2.2 Sparsity-Aware Rewrite Rules. In GNNs, there can be different methods of computing a mathematically equivalent result using different compositions of operations due to the associativity of certain computations. These compositions can result in optimizing sparse tensor computations (graph operations) at the cost of more dense tensor computations and vice-versa. If a graph is comparatively denser, then it is beneficial to optimize its sparse tensor computations (or use fewer of them) due to the higher non-zero-to-node ratio, which makes graph operations more significant to the total runtime. Thus, the most performant among these compositions for a given GNN, depends on the input graph. More specifically, on its sparsity. To exploit this optimization opportunity, we create a transformation pass in GALA to perform peephole-style (local to a specific scope, and thus, only requiring analysis of a sequence of operations via the CIR) operator composition selections, which are simply different associative choices for the same final result. This transformation is applied using the information passed to GALA via the `is_sparser` scheduling command.

GALA uses the operator type and specialization stored in the `cir-nodes` and information of its inputs and outputs from the mapped `dir-nodes`, such as their tensor dimensions, to perform novel sparsity-aware rewrites not enabled in other systems. Equations 1 to 2 are example rewrite rules currently used in GALA. On the left-hand side of the equations, we show the equivalent set of operations for comparatively *denser* graphs, while on the right-hand side, we show the set of operations for comparatively *sparser* graphs. Note that in the equations, operator types are listed in order of execution (with $\rightarrow$) with the specialization itself added as a subscript (e.g. $\texttt{update\_node}_{\text{row broadcast}}$ refers to an `update_node` operation that uses `row broadcast` as the specialization for the update process). In addition, the rewrites require the inputs and outputs of the operations to match certain conditions. These include the operation increasing the dimensions of the output ($n \times k_2$) from the input (where the tensor dimension is $n \times k_1$), and the graph input to the operation being `weighted` or `unweighted`. These details are obtained from `dir-nodes` and are added in the rules within '( )'. For example, `aggregate-node`(`weighted`, k_1) performs node aggregation using a weighted graph where the node features are of k_1 size. When performing the rewrite, GALA replaces the `cir-nodes` matching to a pattern with nodes corresponding to the resulting rewrite and links the relevant `dir-nodes`.

$$[dense]\texttt{update-node}_{\text{row broadcast}} \rightarrow \texttt{aggregate-node}_{+}(\texttt{unweighted}) \rightarrow \texttt{update-node}_{\text{row broadcast}} \\ = \texttt{update-edge}_{\times} \rightarrow \texttt{aggregate-node}_{(\times,+)}(\texttt{weighted})[sparse] \quad (1)$$

$$[dense]\texttt{update-node}_{\text{learn}}(in{:}dir1_{n\times k1}, out{:}dir2_{n\times k2}, k_1{<}k_2) \rightarrow \,..(\text{other operations}) \rightarrow \\ \texttt{aggregate-node}_{(\times,+)}(in{:}dir1_{n\times k1}, out{:}dir3_{n\times k1}) \rightarrow \texttt{update-node}_{\text{learn}}(in{:}dir3_{n\times k1}, out{:}dir4_{n\times k2}, k_1{<}k_2) = \\ \texttt{update-node}_{\text{learn}}(in{:}dir1_{n\times k1}, out{:}dir2_{n\times k2}, k_1{<}k_2) \rightarrow .. \rightarrow \texttt{aggregate-node}_{(\times,+)}(in{:}dir2_{n\times k2})[sparse] \quad (2)$$

We now present concrete examples of when these rewrites are applicable. In addition, for each rewrite, we use an applicable example to demonstrate the equivalence of *LHS* and *RHS* IR sequences.

***Equation* (1).** Graph convolution-based models such as GCN [29], SGC [48], and TAGCN [16] are examples where this rewrite is applied. Taking GCN as a concrete example, the subset of its computation that corresponds to the rewrite in matrix form is $\tilde{D}^{-\frac{1}{2}} \cdot \tilde{A} \cdot \tilde{D}^{-\frac{1}{2}} \cdot H$ (complete computation detailed in Section 2.3). Note that while $\tilde{D}$ is originally a diagonal matrix (i.e. of $N{\times}N$ size, where N is the number of nodes), it can be compressed to a vector (i.e. of $N{\times}1$ size) as it only contains a single value per row. This computation can be performed in the following ways using different mathematically equivalent associative choices matching the *LHS* and *RHS* of Equation (1).

- ***LHS.*** *Associative choice:* $\tilde{D}^{-\frac{1}{2}} \cdot (\tilde{A} \cdot (\tilde{D}^{-\frac{1}{2}} \cdot H))$. Compute $L_1 {=} \tilde{D}^{-\frac{1}{2}} \cdot H$ by broadcasting the values of the $\tilde{D}^{-\frac{1}{2}}$ vector along the rows of H. This is an update-node operation type with a row broadcast specialization. Use the result in an aggregate-node (SpMM) operation with $\tilde{A}$ and a sum specialization (sum node features over all incoming edges). This is applicable, as edge values are all 1 in an unweighted graph. This produces, $L_2 = \tilde{A} \cdot L_1$ Compute the final result $L_3 = \tilde{D}^{-\frac{1}{2}} \cdot L_2$ using another update-node operation type with a row broadcast specialization.
- ***RHS.*** *Associative choice:* $(\tilde{D}^{-\frac{1}{2}} \cdot \tilde{A} \cdot \tilde{D}^{-\frac{1}{2}}) \cdot H$. Compute $R_1 = \tilde{D}^{-\frac{1}{2}} \cdot \tilde{A} \cdot \tilde{D}^{-\frac{1}{2}}$ using an update-edge operation with a multiplication($\times$) specialization. This can be lowered to an SDDMM (detailed in Appendix C), where $\tilde{D}^{-\frac{1}{2}}$ is used as both of the dense inputs. This results in a sparse matrix that represents the adjacency matrix of a weighted graph. Using this graph with H for an aggregate-node operation with a multiplication-sum specialization (multiply node features with edge value and sum over all incoming edges), results in, $R_2 = R_1 \cdot H$.

While *LHS* uses more dense computations, and *RHS* uses more sparse computations, they both produce the same result with different associative choices. Thus, mathematically, the final results of both *LHS* and *RHS* are equivalent.

***Equation* (2).** A matching set of operations necessary for this rewrite is seen in graph attention-based models such as GAT [44] and its derivatives (GATv2 [8] etc.) Taking GAT as a concrete example, the subset of its computation that matches the rewrite is $A \cdot H \cdot W$. Here, W is a matrix representing the learned weights in a neural network, and $H \cdot W$ (node-update) has already been computed prior to reaching this point in the computation. Considering the associative property of the operations in this computation, it is possible to compute this result in the following two ways, each matching the *LHS* and *RHS* of Equation (2).

- ***LHS.*** *Associative choice:* $A{\cdot}(H{\cdot}W)$. Compute $A{\cdot}(H{\cdot}W)$ (node-aggregation) by reusing the pre-computed $(H{\cdot}W)$.
- ***RHS.*** *Associative choice:* $(A{\cdot}H){\cdot}W$ Compute $(A{\cdot}H)$ (node-aggregation), and then multiply the result with W (node-update).

Thus, mathematically $LHS = RHS$. In this scenario, if the node feature size increases with the multiplication by W, *LHS* would make the $A{\cdot}(H{\cdot}W)$ computation more computationally expensive. Meanwhile, *RHS* would perform a considerably less computationally expensive $(A{\cdot}H)$ operation, but would require an additional multiplication with W.

6.2.3 Training Invariant Code Motion. Certain operations in GNNs may produce results that are invariant across training iterations. These operations can be significant, taking a majority of the total execution time of a model. An example is the GCN model's first layer's aggregation and normalization calculations, as seen in Figure 8(a) (till ⓘⓘⓘ). The output for this computation does not change across training epochs as no learning weights are involved in the calculations up to this point. GALA can move computations like these out of the training loop, similar to *loop-invariant code motion* commonly found in the domain of compilers. However, existing GNN systems cannot automatically perform this optimization as it requires the unrolled view of the entire GNN model. This is because this optimization can only be applied to the first layer's operations that do not depend on any learned weights, while the operations in the later layers would depend on the results from learned weights in layers before it. GALA performs this optimization by traversing the CIR nodes in the training loop, marking down computations that use learned weights or are dependent on results of computations that use learned weights (tracked by the dependency relations in the DIR). The CIR nodes that do not use or depend on learned weights are then moved before the training loop in the CIR. Note that for this optimization, information about the workload (training or just inference) of the model implemented in GALA is necessary. This is a detail that GALA tracks, while existing GNN systems do not.

6.2.4 Complexity-Based Operator Re-ordering. This optimization performs peephole-like, semantically equivalent reorderings to reduce the overall complexity of a GNN program. Thus, we do not reorder operations that are not associative. For example, reordering non-linear operations such as ReLU can result in a different final output. This optimization is found in existing work where it was introduced to GNNs in [53] and to general sparse tensor computations in [3]. An example is given in Figure 12, where the update operation (`dsl.nn.ffn`) reduces the node feature embeddings from 605 to 32. By moving this operation before aggregation, the aggregation can then execute on node embeddings of size 32 instead of 605. GALA performs this transformation by traversing the CIR in order, considering adjacent pairs of CIR nodes that would still produce semantically equivalent results when re-ordered. If they can be reordered, GALA checks their computational complexity using the dimension information of the DIR nodes of their inputs. If reordering the operations can reduce the overall computation of both operations, GALA swaps their order in the CIR list as shown in Figure 12, and updates the relevant DIR nodes. This process continues iteratively until no reordering is performed.

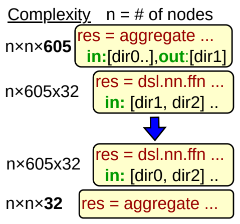


Fig. 12. Operator reordering in GALA

6.3 Code Generation

GALA efficiently generates C++ code with LibTorch and CUDA, using the information specified in the CIR and DIR. GALA does this code generation using two passes over the IR: (a) the custom kernel generation pass and (b) the end-to-end code generation pass. Note that we use LibTorch as the underlying ML framework to use the existing infrastructure for training machine learning models and to implement the Neural Network operations of GNNs. We generate custom CUDA kernels whenever we cannot use an existing operation from LibTorch.

6.3.1 Custom Kernel Operations. The first pass of GALA's code generation process generates custom kernels not supported by the underlying ML framework. GALA does this by first traversing the entire CIR node list from end-to-end and identifying all the custom compute kernels necessary throughout the GNN program, including those necessary for the backward pass. GALA identifies the kernels necessary for a backward pass through a rule-based identification that uses the information stored in the CIR nodes and information in the input and output DIR nodes. For example, ⅲ in Figure 8, which performs node-aggregation on ② using ③ in the forward pass, would *also* perform a node-aggregation on ② (no need to transpose as ② is undirected) using the derivatives of the output in the backward pass. Thus, the same kernel used in the forward pass can be used in the backward pass if no special transformations are specifically made for the backward computation or its inputs. During the traversal of the CIR, GALA ensures that no duplications or kernels already supported in the underlying machine learning system are added to this list. GALA achieves this by using the information in the computation's CIR node and its inputs' and outputs' DIR nodes. For example, even if the underlying system supports a node-aggregation operation (underlying kernel is SpMM) needed for ⅲ in Figure 8, because ② is unweighted, it can use a kernel for node-aggregation that does not read the edge values of the graph. This would require GALA to generate a custom kernel different from a generic node-aggregation kernel. In this instance, the relevant kernel must be added to the custom kernel list. After completing this list by traversing the entire CIR, GALA generates the necessary custom kernels with the transformations specified in the relevant CIR nodes and the DIR nodes. For example, using ② and ⅲ in Figure 8, GALA generates a node-aggregation kernel (SpMM), that does not read the edge values of the graph, operates on a column tiled graph, and has a thread coarsening factor of 2.

Algorithm 1: GALA's Code Generation

```
Input: Data-IR ($dIR$), Compute-IR ($cIR$)
Output: Torch/Cuda executable ($code$)
Function generateExecutable($dIR, cIR$):
    /* $1^{st}$ pass: Custom kernels */
    $customKerns \leftarrow []$;
    foreach $node \in cIR$ do
        if needCustomKernel($node.forward$) and not in $customKerns$ then
            $customKerns.add(node.forward)$
        end
        if needCustomKernel($node.backward$) and not in $customKerns$ then
            $customKerns.add(node.backward)$
        end
    end
    $generateKernels(customKerns, code)$;
    /* $2^{nd}$ pass: End-to-end */
    foreach $node \in cIR$ do
        if needCustomKernel($node.forward$) then
            $generateAutograd(node, code)$;
        end
        $generateCode(node, code)$;
        if node is dataload then
            $generateTransforms(node, code)$;
        end
    end
    $config \leftarrow getTrainingConfig(cIR)$;
    $generateTraining(config, code)$;
```

6.3.2 End-to-End Code Generation. GALA's second pass over its IR generates end-to-end code for the GNN program. In this stage, GALA traverses the CIR, generating the code for the computation of each traversed node. For each dataset loading (loads to memory) or data generation operation encountered, GALA adds the data transformations performed on the outputs of these operations.

These transformations can be identified from the DIR nodes of the outputs in the CIR nodes of the data-loading operations. For example, in Figure 8, ⓘ loads ① and ③ on to memory. GALA further transforms ① through the `col_tile` operation to produce ② (represented as transformation relation in the DIR). Note that GALA performs these transformations on CPU instead of GPU. After the transformation has been performed, GALA adds the necessary memory transfer operations to copy the now transformed data from host memory (CPU) to device memory (GPU). GALA can then use this data in subsequent computations that it encounters while traversing the CIR. GALA uses the custom kernels that it generated, along with existing kernels of the underlying system, for the code generation of these computations. Note, for compute operations *in the training loop* that use custom kernels, GALA must additionally generate Autograd functions with both forward and backward operations. After traversing through the entire CIR and generating the relevant code, GALA generates the code necessary for training based on training configurations. GALA stores this information in the training loop node in the CIR. This is a straightforward code generation that uses the built-in functions of LibTorch.

7 Implementation

7.1 Compiler

We developed the GALA compiler in C++ with LibTorch and CUDA as the target languages. We build the front-end parser using Flex and Bison [33], which lowers the front-end language to an AST. We use this AST to generate the two IRs of GALA for a given GNN program (Section 6.1.4). We use the code generation process detailed in Section 6.3 to generate an executable based on LibTorch and CUDA. Note that we only generate primitive kernel code in CUDA for the kernels of computation operations we intrusively optimize. For other instances, we use cuSparse [37] for sparse computations and LibTorch [38] for dense DNN computations. In addition, we store input datasets in a binary representation to make data loading faster.

7.2 Input-Aware Compilation

```
G=G.opt_input("/path/to/dataset");
```

Fig. 13. Schedule for input-aware code generation in GALA

The input sensitivity of optimizations is a property inherent in any graph-based sparse computation. As GNNs have irregular sparsity due to their graph input, finding the best schedule in GALA for a particular input would be challenging for a user. As a solution, we provide an additional feature for GALA, which can inspect the input dataset to predict the best schedule. This is done using the scheduling command, `opt_input`, as shown in Figure 13. Through this, GALA can know the input dataset at compile time and perform an $O(E)$ (E is the number of edges in the graph) analysis on the graph of the dataset to predict the best scheduling parameters. GALA predicts these parameters using heuristic rules. For example, by inspecting the input graph, the compiler can know its sparsity (E/N^2, N being the number of nodes in the graph) and use it to determine if to apply GALA's sparse rewrite rules. We show a quick evaluation of this feature of GALA in Section 8.5.5.

8 Evaluation

This section evaluates the performance of GNN models implemented in GALA in comparison with multiple existing works. We conduct these evaluations separately for GNN inference and training, and perform further ablations as well as sensitivity studies to demonstrate the capabilities of GALA.

8.1 Research Questions

(1) How much faster are GNN models implemented in GALA compared to existing systems for,

   (a) Inference: optimizing only the forward pass (Section 8.3)
   (b) Training: optimizing the GNN model for training, which includes both the forward pass and the backward pass (Section 8.4)
(2) How does each IR of GALA contribute to the overall speedup a GNN? (Section 8.5.1)
(3) How well does GALA scale? Specifically, when the number of hidden layers and hidden dimensions of the GNNs change, as well as when the graph scales? (Section 8.5.2)
(4) What is the memory consumption of GALA compared to existing systems? Do different optimization choices of GALA also affect memory consumption? (Section 8.5.3)
(5) How do GALA's data and compute transformation for sampling perform? (Section 8.5.4)

8.2 Experimental Setup

GNN Models and Hyper-parameter Configurations. - We evaluate GALA using 4 GNN models. For the main evaluations, we use Graph Convolutional Networks (GCN), Graph Attention Network (GAT) [44], Graph Isomorphism Network (GIN) [52], and GraphSAGE (SAGE) with the mean aggregation [20]. We selected these four models due to their ubiquitous use in recent studies [14, 21, 23, 28, 34, 36, 46] as well as in the evaluation of multiple GNN systems, including our baselines [19, 24, 45, 50, 51, 54, 56, 60]. We use two layers for all models with a hidden feature dimension of 32. We believe this is a representative configuration as it is used in prior work [29, 59] to achieve high levels of accuracy, with similar configurations used by our baselines for their evaluations [24, 45, 50]. However, we conduct a further study with varying layers and hidden feature dimensions in Section 8.5.2 to show GALA's performance with varying hyper-parameter configurations. In addition, to showcase sampling in GALA, we evaluate the effect of node sampling on a GraphSage model with the GCN pooling strategy.

Datasets. - We list the graphs used for our evaluation in Table 6. We source these graphs from two widely used sources: (a) from DGL's graph datasets and (b) from Open Graph Benchmark's (OGB) node property prediction datasets. We made this selection as these graph datasets cover various sizes and non-zero distributions while being used for evaluation in other GNN systems [45, 50, 51, 54, 56, 60]. For our graph scalability analysis in Section 8.5.2, we use the *OGBN-papers100M* dataset with node sampling.

Table 6. Graphs used for evaluation

Graph	#Nodes	#Edges	#Features	#Classes	Source	Train:Val:Test
Cora	2,708	10,556	1,433	7	DGL	9:30:61
Pubmed	19,717	88,651	500	3	DGL	4:32:64
CoraFull	19,793	126,842	8,710	70	DGL	70:15:15 [1]
Reddit	232,965	114,615,892	602	41	DGL	66:10:24
ogbn-arxiv	169,343	1,166,243	128	40	OGB	54:17:29
ogbn-products	2,449,029	126,167,053	100	47	OGB	8:2:90

Testbed Machines. - Machine learning models are typically run on a GPU. Thus, we perform our evaluations on two GPU machines, which used (a) an NVIDIA H100 GPU with 94 GB of memory, the host being an AMD EPYC 9454 CPU with a RAM of 377GB and (b) an NVIDIA A100 GPU with 80 GB of memory, the host being an Intel Xeon Platinum 8358 CPU with a RAM of 256GB.

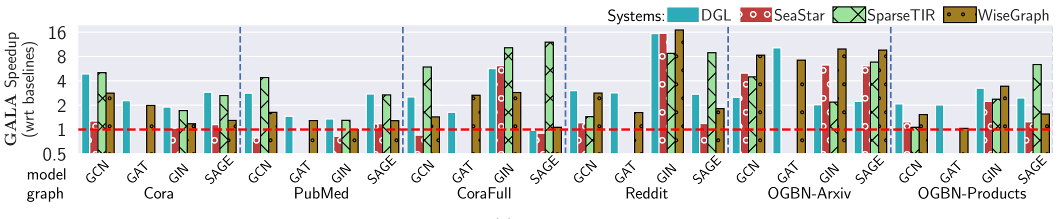
Baselines. - We use multiple baselines to show that GALA can achieve significant speedups compared to existing systems by composing multiple optimizations at different levels in a GNN model. We make comparisons against both (a) GNN systems: DGL (v2.4, commit #7738, released

[1]We performed a $train : validation : test$ set split at a $70 : 15 : 15$ ratio for this graph dataset as it did not have any inherent dataset separation.

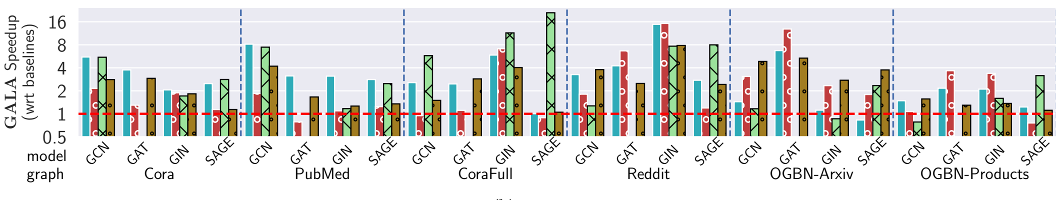
2024) [45], SeaStar [50], and WiseGraph [24], as well as (b) Sparse Tensor systems: SparseTIR [54]. In their evaluations, SparseTIR and WiseGraph compare against other GNN systems, such as GNNAdvisor[47] and dgSparse[25], showing speedups over them. Note that we did not extend the existing implementations of primitives within these baselines. Thus, we only have evaluations for a subset of the models for certain baselines. For example, we only compare GCN, GIN, and SAGE for SparseTIR, as the released artifact does not have the primitives necessary for GAT. To show GALA's usability, we conducted a simple line-number study. To implement an optimized GCN model for the *Reddit* dataset, GALA only requires *23* lines of code, while DGL requires *323* (with the necessary optimizations added as custom kernels) and SparseTIR requires *91* lines of code. Note that this is only a proxy for a proper usability study.

As a sanity check to ensure that GALA's and the baselines' results were accurate, we compared the test accuracy of all evaluations. This led us to discover a bug in the artifact of WiseGraph for GAT. We resolved this issue after discussions with the authors and presented results after the fix (where the authors suggested an alternative code). We discovered a similar issue with SeaStar for GAT, specifically on the H100 machine. In this instance, we avoided reporting the numbers as the issue seemed to be hardware-specific, and our best efforts did not yield a fix.

8.3 GNN Inference Runtime Performance



(a) H100



(b) A100

Fig. 14. Speedup of GALA over baselines for model inference. Here, a speedup > 1 indicates GALA outperforms a baseline. The $y - axis$ is in log_2.

Figure 14 shows the speedups of GALA compared to the baselines evaluated. We observe significant speedups against the majority of the baselines evaluated. We observed the execution times for GALA (as well as in our baselines) varied significantly based on the graph dataset: 0.26ms for Cora, and 14.85ms for ogbn-products, as well as the GNN model: for Reddit, 5.62ms for GCN, 17.73ms for GAT. Results below the dotted red line at 1.0 indicate that the baseline performs better than GALA, while higher values indicate that GALA outperforms the respective baseline.

For GALA, we observe geo-mean speedups of 2.74× over DGL, 1.97× over SeaStar, 3.43× over SparseTIR, and 2.37× over WiseGraph. Model-wise, we observe geo-mean speedups of 2.4× for GCN, 2.63× for GAT, 3.12× for GIN, and 2.16× for SAGE when averaging across all four baselines and two machines. Machine-wise, we observe geo-mean speedups of 2.62× for H100, and 2.48× for A100 when averaging across all our four baselines and models. Considering the example for the *Reddit* Dataset for GCN on the H100 in Figure 3(a), each of the intra-operator optimizations, column-tiling and using operators specialized for unweighted graphs, results in speedups of 1.3×. When combined, the total speedup achieved through intra-operator optimizations is 1.6×. However, when composed with the operator reordering inter-operator optimization (5× speedup), the composition achieves a synergistic speedup greater than the product of their independent speedups in isolation ($10.3\times > (8\times = (1.6 * 5)\times)$). This ability to compose inter- and intra-operator optimizations makes GALA a powerful tool to achieve significant speedups for GNNs over existing systems.

We observed slowdowns compared to the baseline systems on only a few occasions (12 out of 174 in Figure 14). These slowdowns were mainly observed in smaller (*PubMed*) or sparser (*OGBN-Products*) graphs but were inconsistent across baselines and models. For these datasets, we noticed that the specialized optimizations applied by some of our baselines were beneficial. For example, the load-balancing optimization for sparse operations that Seastar applies. However, GALA still achieves considerable speedups overall, being close to or over 2× for each baseline.

8.4 GNN Training Runtime Performance

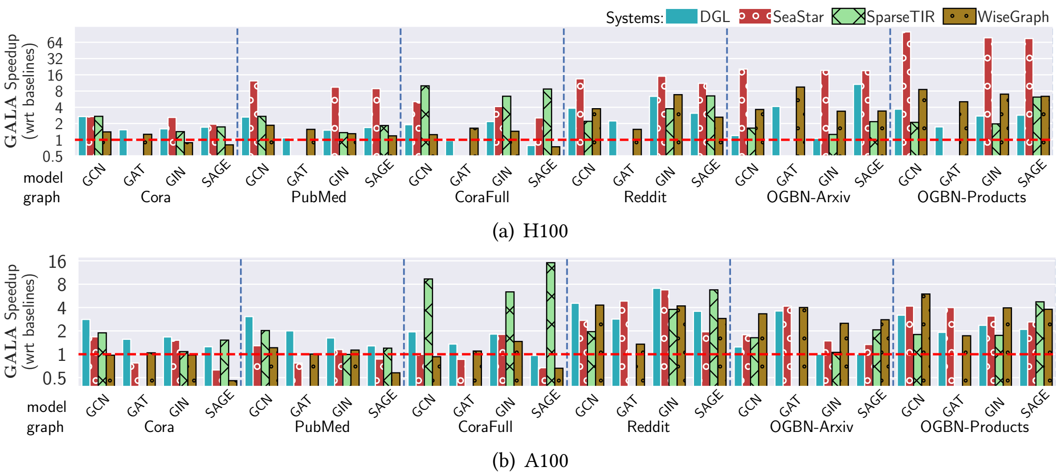


Fig. 15. Speedup of GALA over baselines for model training for 100 epochs. Here, a speedup > 1 indicates GALA outperforms a baseline. The $y - axis$ is in log_2.

Figure 15 shows the speedups of GALA compared to the baselines when training GNN models for 100 epochs. Results below the dotted red line at 1.0 indicate that the baseline performs better than GALA, while higher values indicate that GALA outperforms the respective baseline. We observe geo-mean speedups of 2.1× over DGL, 3.83× over SeaStar, 2.69× over SparseTIR, and 2.01× over WiseGraph, when training. Model-wise, we observe geo-mean speedups of 2.97× for GCN, 1.9× for GAT, 2.64× for GIN, and 2.44× for SAGE. Machine-wise, we geo-mean speedups of 3.35× for the H100 machine and 1.94× for the A100 machine. As Figure 4 shows, a 7× speedup can be achieved for training a GCN model for the *Reddit* dataset with only inference optimizations. However, this can be increased significantly to 12× by training specific optimizations such as training-mask-based sub-graph creation (Section 6.2.1) and training invariant code motion (Section 6.2.3). These training-specific optimizations allowed GALA to achieve a geo-mean speedup of 2.52× over other baselines, compared to the geo-mean speedup of 2.08× achievable with only optimizations for inference.

We observe a comparatively lesser speedup for GAT, as optimizations such as training-invariant code motion (Section 6.2.3) cannot be performed as the aggregation operation involves learned weights. The few slowdowns we observed (20 out of 174 in Figure 15) were mainly against SeaStar and WiseGraph for smaller graphs (*Cora*). However, we observed significant speedups for larger graphs (*Reddit*). This points to these systems being more optimized for smaller graphs.

8.5 Ablation and Sensitivity Studies

8.5.1 Isolated Contributions of IRs. We use Table 7 to showcase the importance of each IR in GALA and the benefit of their combination. Note that we designed our IRs to be heavily interconnected. Isolating each IR limits the transformations that are applied to just one type of intra-operator transformation. With only CIR, transformations are limited to intra-operator compute transformations (e.g. thread coarsening), while with only DIR, transformations are limited to intra-operator data transformations (e.g. column tiling). Using *both* CIR and DIR simultaneously allows for inter-operator transformations (Section 6.2) but still falls short of the speedup achievable when all transformations are combined. With this, we again see the synergy of applying multiple types of transformations leading to higher speedups (For RD, 10.3>(0.9×1.36×5)), similar to what we observed in Section 3.

Table 7. Speedup from CIR and DIR separately, only from *both* CIR and DIR, and combining all for *Reddit*(RD) and *OGBN-Products*(OP) (2-layer GCN, hidden dim. of 32)

IRs used	**Speedup**		**Transformations applied**
	RD	OP	
Only CIR	0.9×	0.9×	Thread coarsening
Only DIR	1.36×	1.26×	Column tiling, unweighted graph kernels
CIR *and* DIR	5×	1.5×	Inter-operator transformations
Combining all	10.3×	2.5×	Combining all transformations

8.5.2 Scalability. Figure 16 shows the speedups of GALA against WiseGraph across 2, 3, 4, and 8 layers and hidden dimensions varying from 32 to 1024. For this evaluation, we tested the GCN model for the Reddit dataset. We observe that the speedups generally increase as the number of layers increases. This stands as evidence of GALA's capability to optimize each layer of a given GNN model to contribute to the final overall speedup. We also observe a decrease in speedup as the hidden dimension size increases. This is due to the greater increase in the overall runtime of the update operations (GEMM primitive) where we use the same implementation as the baseline (Torch, which WiseGraph also uses), compared to the sparse operations, such as aggregate, which GALA optimizes more compared to the baseline. To elaborate, given that N is the number of nodes in the graph, E is the number of edges, and K is the hidden dimension size (assuming both input and output sizes are same for simplicity), the complexity of the update operation is $O(N \cdot K^2)$, while the aggregate operation is $O(E \cdot K)$ (growing at K^2 vs. K). Nonetheless, we still observed significant speedups across all the layers and hidden dimensions we evaluated.

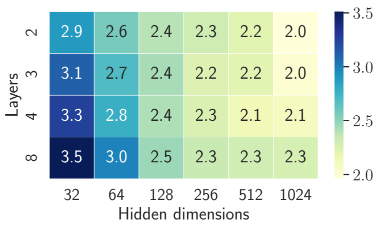


Fig. 16. Heatmap of speedups over WiseGraph for GALA with varying number of layers and hidden dimensions (*Reddit*,GCN).

We use the *OGBN-papers100M* dataset with node sampling to show GALA's scalability with increasing graph sizes. Here, we sample the first $n\%$ of nodes when performing the node sampling. Figure 16 shows the results of this evaluation by presenting the runtimes of GALA, DGL, and WiseGraph with different node-sampling percentages. We observe that GALA consistently achieves the best runtimes across the different graph sizes. These, as well as the previous results with differing layers and hidden dimensions, show that GALA achieves considerable speedups across different layers, hidden dimensions, as well as graph sizes.

Table 8. Runtimes for OGBN-papers100M with node sampling

Sample	#Nodes	#Edges	GALA	DGL	WiseGraph
1 %	1.11M	1.31M	3.11	4.19	3.56
2 %	2.22M	3.01M	6.07	7.64	6.81
5 %	5.55M	9.73M	15.62	18.75	16.9
10 %	11.11M	29M	31.61	39.63	41.85
20 %	22.21M	49.77M	62.98	94.52	*OOM*

8.5.3 Effect of Memory Consumption on Different Schedules. Figure 17 shows runtime and memory consumption when training a 2-layer GCN model, for the *Reddit* graph dataset, for DGL, WiseGraph, and two different scheduling choices for GALA. The first schedule optimizes for memory and then performance, while the latter further optimizes performance at the cost of a higher memory consumption (2.05× more memory to run 1.23× faster). This additional memory consumption comes from tiling the graph (which is larger than the graph without tiling) and creating subgraphs based on the training set (Section 6.2.1). By allowing the control of such choices through a schedule, GALA can facilitate the execution of larger graph datasets at the cost of performing sub-optimally (yet still faster than existing systems).

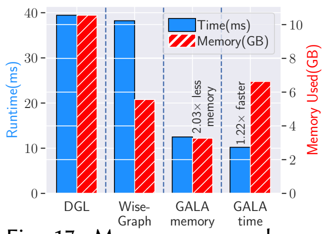


Fig. 17. Memory use and runtime of DGL, WiseGraph, and GALA with memory and time optimized schedules (2-layer GCN)

8.5.4 Sampling. We use neighborhood sampling to inspect the impact of functionally similar data and compute transformations in GALA. We observe that the different methods of sampling result in varying outcomes, especially with other optimizations in play. The neighborhood sampling methods in GALA are, (a) `data sampling` - creates a sampled sub-graph of the original input graph, and (b) `kernel sampling` - samples during the kernel execution itself based on random values passed to the kernel. Table 9 shows the time and test accuracy observed for a GraphSage model with GCN pooling, for GALA with (i) no sampling (best schedule), (ii) with data sampling (e.g. `G.sample(..)`), (iii) with kernel sampling (e.g. `aggr_fn.sample(..)`), and (iv) with dynamic kernel sampling (e.g. `aggr_fn.sample(..).dynamic()`). Note that in the latter sampling method, the samples differ across epochs (thus, dynamic) and are not semantically the same as (ii) and (iii). As seen in Table 9, the different sampling methods show speedups against the model without sampling. Notably, sampling applied to *OGBN-Products* produced a higher accuracy than the un-sampled execution due to inductive learning [20].

Table 9. Accuracy and inference times of different sampling methods in GALA for *Reddit* (RD) and *OGBN-Products*(OP) (Sample size = 20, GCN)

Sampling Method	**Time(ms)**		**Test Acc(%)**	
	RD	OP	RD	OP
No sampling	6.47	12.53	94.51	71.04
Data sampling	0.95	6.38	92.07	73.31
Kernel sampling	1.07	7.26		
Dynamic Kernel sampling	1.07	7.26	93.13	75.12

The most suitable sampling method may differ depending on the objective of executing the GNN model. While data sampling has the fastest inference time per epoch, it incurs a pre-processing overhead and consumes more memory. This overhead and memory consumption are dependent on the graph and were 716 MB (20% of the total memory consumption) and for 1.9 ms *Reddit.* In situations with significant constraints on memory or if the GNN model was only run for a single pass, the kernel sampling approach would be more beneficial. If the objective is accuracy, the more advanced dynamic kernel sampling is the most suitable as it dynamically samples across epochs. This is shown for the *OGBN-Products* dataset, where it achieves the highest accuracy.

8.5.5 Manual Schedules vs. Input-Aware Compilation. The input-aware compilation (Section 7.2) found schedules that were within 10% of hand-tuned schedules. By inspecting the dataset, GALA predicts the best configuration for most schedules deterministically (such as `set_undirected`). GALA predicts a parameter for other non-trivial schedules such as `col_tile` using heuristics.

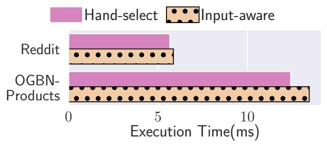


Fig. 18. GALA's input-aware code-generation compared to hand-tuned schedules

8.5.6 Overheads. GALA's compilation overhead is very minimal, averaging only 5*ms*. In addition to the compilation time, the data transformations of GALA also introduce preprocessing overheads to the GNN program. Notably, `col_tile` can have a significant overhead, especially for smaller tiling factors. In our evaluations, we use `col_tile` only for Reddit. The sub-graph creation in Section 6.2.1 also introduces an overhead when performing training. However, these overheads can be amortized, especially when training, as a GNN model can train for a large number of epochs [6]. The overall preprocessing overhead we observe for Reddit is approximately 25% of a training execution with 1000 epochs. This overhead is justifiable as even with it added to the execution time, we still see a speedup of 3.02× over WiseGraph. Moreover, transformations such as these that produce overheads can be turned off through GALA's scheduling language based on a user's need. In addition, although we observed significant overheads (over 500s in certain instances) in the other baseline systems, such as SparseTIR, we make comparisons purely on runtime.

9 Related Work

DNN Systems. Throughout the years, multiple systems have been developed to optimize deep neural networks, which are primarily composed of dense operations. Compilers such as XLA [42] and PyTorch 2 [4] have achieved significant speedups through various compiler optimizations and have enabled the efficient execution of large DNN models such as Large-Language Models (LLMs). However, DNN systems cannot efficiently implement GNNs as they do not support irregular sparse computations, which brings its own challenges to the table.

DSLs with Scheduling Languages. We draw inspiration from Halide [40], the work that first popularized the separation of algorithms and schedules for generating optimized code and targeted image processing applications. An extension of Halide also introduced the ability to schedule computations of operations used in the backward pass [35] similar to GALA. However, it only focused on the computations themselves, whereas in GALA, transformations can be performed on both the computations and data used in the backward pass. Slapo [9] is another system that uses Halide as inspiration and presents a scheduling language for Large Deep Learning Models. Graphit [58], and TACO [30], are scheduling languages much closer to GNNs as they focus on graphs and sparse tensors, respectively. Compared to these languages, GALA opts for a hybrid scheduling-automatic optimization approach focusing on GNNs.

Sparse Tensor Systems. The irregularity of most sparse computations led to multiple works that achieve optimizations through innovative techniques applicable beyond regular-dense computations. A classical optimization is changing the underlying sparse data format. Adaptive Sparse Tiling [22] is a well-known work in this domain that tiles the adjacency matrix of a graph based on the degree of a node. There have been other works that select the best sparse format based on the input [55], with some among them specifically focusing on GNNs [39]. Considering input sensitivity, there have been multiple works that achieve significant speedups through inspector-executor-based executions [10, 11]. TACO [30] is another milestone work in the domain of sparse computations, as it produced a novel intermediate representation that allowed the general representation of multiple sparse formats along with scheduling transformations. Many works have followed which improve upon TACO, such as [3, 13, 27], which perform optimizations on loop ordering in sparse computations, and the fusion of kernels [12, 13]. In addition, going beyond traditional CPU implementations, sparse tensor systems that function on GPUs, such as SparseTIR [54] and GSparse [26] have also been developed. Systems such as SpEQ [31] and Mosaic [5] provide a new direction to lowering sparse matrix operations by leveraging existing high-performance implementations whenever possible. However, compared to the end-to-end GNN optimizations that GALA enables, most sparse tensor systems focus on a single operator or a set of adjacent operators to improve performance for general sparse computations. This leads to these systems missing domain-specific optimizations explored under GALA.

GNN Systems. Multiple GNN systems, ranging from general-purpose GNN frameworks to compilers targeting specialized GNN models, have been developed over the years. DGL [45], and PyG [18] still stand as popular frameworks due to their programmability and regular updates that continuously improve existing implementations. Graphiler [51] and SeaStar [50] are compilers specialized for optimizing GNNs with user-defined functions. Hector [49] is a system that uses a data layout and kernel IR separation similar to GALA to optimize for Relational GNNs. In addition to these systems, there have been multiple works that achieve significant speedups in GNNs through various optimizations. [56] uses multiple optimization techniques, such as operator reordering, kernel fusion, and re-computation of training results to achieve significant speedups. GNNAdvisor [47] exposes the importance of input aware-optimizations in GNNs, such as reordering. uGrapher [60] is another input-aware system that optimizes GNNs by selecting through different parallelization schemes. WiseGraph [24] uses graph partitioning to accelerate GNN models. In addition, operator fusion is another optimization commonly seen applied in the domain of GNNs [19, 41]. Compared to these systems, GALA composes intra- and inter-operator optimizations to achieve significant speedups. Through GALA's IR design, GALA can also retain a global view of an entire GNN program to enable optimizations specific to the training context and enhance the end-to-end performance of GNNs.

10 Limitations and Future Work

GALA shows the synergistic benefits of performing both intra- and inter-kernel optimizations in GNNs while enabling novel automatic domain-specific optimizations. We believe that GALA can be extended to include even more optimizations, which we consider as valuable future work. Here, we list some of these along with potential methods to support them.

- GALA does not expose all possible schedule optimizations, such as fine-grained control of memory (in GPUs, storing and using in global or shared memory) and thread mapping. Specifically, these optimizations can be exposed to the user by extending the scheduling language and storing the specified transformations in the CIR or DIR.
- Operator fusion is a powerful technique that can be used to attain significant speedups (e.g. one of the techniques used by SeaStar to attain the speedups in Figure 14 for GAT). GALA can add this transformation to its arsenal by adding a new transformation pass to fuse nodes in the CIR.
- GALA performs code transformations that are either (a) based on the input schedule (intra-operator) or (b) are automatic transformations that always improve runtime (inter-operator). However, more intricate transformations (such as operator fusion as described in the point above) would require complex input-aware decisions to be made at both the intra- and inter-operator levels. Extending the current transformation infrastructure to incorporate tools such as cost models for simulating runtime costs would enable better search strategies and allow such transformations to be supported by GALA.

11 Conclusion

In this work, we introduce GALA, a DSL and a compiler that optimizes both GNN inference and training. GALA compiler composes optimizations at both intra- and inter-operator levels by separately tracking the transformations using its compute and data IRs. This allows GALA to exploit synergies between optimizations that happen at different levels, leading to improved runtime performance. Further, GALA's IRs maintain a global view of the GNN computation, allowing it to perform novel training-specific optimizations. Our evaluations show that GALA outperforms state-of-the-art GNN inference and training systems across widely used GNN models and graphs.

Data-Availability Statement

Our artifact [32], evaluated and freely available, is a C++ code repository that implements a DSL compiler that lowers GNN models written in the GALA language to CUDA and LibTorch. When compiled, this repository would generate the compiler as an executable. The datasets used for the artifact are well-known in the GNN domain and are publicly available at https://ogb.stanford.edu/docs/nodeprop/ and https://www.dgl.ai/dgl_docs/api/python/dgl.data.html.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and our shepherd for their guidance. We would also like to thank Sameh Gobriel for all the guidance given, Vimarsh Sathia for the support in the early stages of the project, as well as Wanyu Zhao, Stefanos Baziotis, and Jai Arora for their feedback on drafts of this paper. 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] Abien Fred Agarap. 2018. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375* (2018).

[2] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 41–54. doi:10.1145/3579990.3580020

[3] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. doi:10.1145/3519939.3523442

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[5] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. 2023. Mosaic: An Interoperable Compiler for Tensor Algebra. *Proceedings of the ACM on Programming Languages* 7 (June 2023). Issue PLDI.

[6] Filippo Maria Bianchi, Daniele Grattarola, and Cesare Alippi. 2020. Spectral Clustering with Graph Neural Networks for Graph Pooling. arXiv:1907.00481 [cs.LG] https://arxiv.org/abs/1907.00481

[7] Pasquale Bove, Alessio Micheli, Paolo Milazzo, Marco Podda, et al. 2020. Prediction of Dynamical Properties of Biochemical Pathways with Graph Neural Networks.. In *Bioinformatics*. 32–43.

[8] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks? arXiv:2105.14491 [cs.LG]

[9] Hongzheng Chen, Cody Hao Yu, Shuai Zheng, Zhen Zhang, Zhiru Zhang, and Yida Wang. 2024. Slapo: A schedule language for progressive optimization of large deep learning model training. In *ASPLOS 2024*. https://www.amazon.science/publications/slapo-a-schedule-language-for-progressive-optimization-of-large-deep-learning-model-training

[10] Kazem Cheshmi, Zachary Cetinic, and Maryam Mehri Dehnavi. 2022. Vectorizing sparse matrix computations with partially-strided codelets. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 32, 15 pages.

[11] Kazem Cheshmi, Shoaib Kamil, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2017. Sympiler: transforming sparse matrix codes by decoupling symbolic analysis. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 13 pages. doi:10.1145/3126908.3126936

[12] Kazem Cheshmi, Michelle Strout, and Maryam Mehri Dehnavi. 2023. Runtime Composition of Iterations for Fusing Loop-carried Sparse Dependence. In *Proceedings of the International Conference for High Performance Computing,*

Networking, Storage and Analysis (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 89, 15 pages. doi:10.1145/3581784.3607097

[13] Adhitha Dias, Logan Anderson, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. 2024. SparseAuto: An Auto-Scheduler for Sparse Tensor Computations Using Recursive Loop Nest Restructuring. arXiv:2311.09549 [cs.PL] https://arxiv.org/abs/2311.09549

[14] Changxu Dong, Dengdi Sun, Zhenda Yu, and Bin Luo. 2025. Multi-view brain network classification based on Adaptive Graph Isomorphic Information Bottleneck Mamba. *Expert Systems with Applications* 267 (2025), 126170. doi:10.1016/j.eswa.2024.126170

[15] Manfred Droste, Werner Kuich, and Heiko Vogler. 2009. *Handbook of Weighted Automata* (1st ed.). Springer Publishing Company, Incorporated.

[16] Jian Du, Shanghang Zhang, Guanhang Wu, Jose M. F. Moura, and Soummya Kar. 2018. Topology Adaptive Graph Convolutional Networks. arXiv:1710.10370 [cs.LG]

[17] Farida Farsian, Federico Marulli, Lauro Moscardini, and Carlo Giocoli. 2023. New Applications of Graph Neural Networks in Cosmology. In *Machine Learning for Astrophysics*, Filomena Bufano, Simone Riggi, Eva Sciacca, and Francesco Schilliro (Eds.). Springer International Publishing, Cham, 35–38.

[18] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[19] Zhangxiaowen Gong, Houxiang Ji, Yao Yao, Christopher W. Fletcher, Christopher J. Hughes, and Josep Torrellas. 2022. Graphite: Optimizing Graph Neural Networks on CPUs through Cooperative Software-Hardware Techniques. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 916–931. doi:10.1145/3470496.3527403

[20] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.

[21] Md Abul Hasnat, Somayeh Asadi, and Negin Alemazkoor. 2025. A graph attention network framework for generalized-horizon multi-plant solar power generation forecasting using heterogeneous data. *Renewable Energy* 243 (2025), 122520. doi:10.1016/j.renene.2025.122520

[22] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. 2019. Adaptive Sparse Tiling for Sparse Matrix Multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 300–314. doi:10.1145/3293883.3295712

[23] Yang Hu, Wenxi Wang, Sarfraz Khurshid, Kenneth L. McMillan, and Mohit Tiwari. 2024. 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* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 104–115. doi:10.1109/ASE56229.2023.00167

[24] Kezhao Huang, Jidong Zhai, Liyan Zheng, Haojie Wang, Yuyang Jin, Qihao Zhang, Runqing Zhang, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2024. WiseGraph: Optimizing GNN with Joint Workload Partition of Graph and Operations. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3627703.3650063

[25] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. 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* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 119–132. doi:10.1145/3437801.3441585

[26] Yue Jin, Chengying Huan, Heng Zhang, Yongchao Liu, Shuaiwen Leon Song, Rui Zhao, Yao Zhang, Changhua He, and Wenguang Chen. 2023. G-Sparse: Compiler-Driven Acceleration for Generalized Sparse Computation for Graph Neural Networks on Modern GPUs. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 137–149. doi:10.1109/PACT58117.2023.00020

[27] Raghavendra Kanakagiri and Edgar Solomonik. 2024. 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)*. ACM, 169–181. doi:10.1145/3626183.3659985

[28] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 387–400. https://proceedings.mlsys.org/paper/2021/file/85d8ce590ad8981ca2c8286f79f59954-Paper.pdf

[29] Thomas N. Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *CoRR* abs/1609.02907 (2016). arXiv:1609.02907 http://arxiv.org/abs/1609.02907

[30] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. doi:10.1145/3133901

[31] Avery Laird, Bangtian Liu, Nikolaj Bjørner, and Maryam Mehri Dehnavi. 2024. SpEQ: Translation of Sparse Codes using Equivalences. *Proc. ACM Program. Lang.* 8, PLDI, Article 215 (June 2024), 24 pages. doi:10.1145/3656445

[32] Damitha Lenadora, Nikhil Jayakumar, Chamika Sudusinghe, and Charith Mendis. 2025. *Artifact for OOPSLA 2025 Paper: GALA: A High Performance Graph Neural NetworkAcceleration LAnguage and Compiler.* doi:10.5281/zenodo.16923829

[33] John Levine and Levine John. 2009. *Flex & Bison* (1st ed.). O'Reilly Media, Inc.

[34] Guannan Li, Le Zhang, Lingzhi Yang, Hang Hu, Chengliang Xu, Liangzhen Jiao, Donghua Liu, Chenglong Xiong, and Jiahui Deng. 2025. Model interpretation and interpretability performance evaluation of graph convolutional network fault diagnosis for Air Handling Units. *Journal of Building Engineering* 103 (2025), 112048. doi:10.1016/j.jobe.2025.112048

[35] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.* 37, 4, Article 139 (July 2018), 13 pages. doi:10.1145/3197517.3201383

[36] Zhuoran Li, Lianshan Yan, Hua Li, and Yu Wang. 2025. Environmental factors-aware two-stream GCN for skeleton-based behavior recognition. *Mach. Vision Appl.* 36, 2 (Jan. 2025), 12 pages. doi:10.1007/s00138-024-01656-7

[37] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*, Vol. 12.

[38] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[39] Shenghao Qiu, Liang You, and Zheng Wang. 2022. Optimizing Sparse Matrix Multiplications for Graph Neural Networks. In *Languages and Compilers for Parallel Computing*, Xiaoming Li and Sunita Chandrasekaran (Eds.). Springer International Publishing, Cham, 101–117.

[40] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. doi:10.1145/2499370.2462176

[41] Md Rahman, Majedul Haque Sujon, Ariful Azad, et al. 2021. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks. In *35th Proceedings of IEEE IPDPS.*

[42] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

[43] Zixing Song, Yuji Zhang, and Irwin King. 2023. Towards fair financial services for all: A temporal GNN approach for individual fairness on transaction networks. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management.* 2331–2341.

[44] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations.* https://openreview.net/forum?id=rJXMpikCZ

[45] 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. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. arXiv:1909.01315 [cs.LG]

[46] Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth McMillan, and Risto Miikkulainen. 2024. NeuroBack: Improving CDCL SAT Solving using Graph Neural Networks. arXiv:2110.14053 [cs.AI] https://arxiv.org/abs/2110.14053

[47] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21).* USENIX Association, 515–531. https://www.usenix.org/conference/osdi21/presentation/wang-yuke

[48] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr. au2, Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. 2019. Simplifying Graph Convolutional Networks. arXiv:1902.07153 [cs.LG]

[49] Kun Wu, Mert Hidayetoğlu, Xiang Song, Sitao Huang, Da Zheng, Israt Nisa, and Wen-Mei Hwu. 2024. Hector: An Efficient Programming and Compilation Framework for Implementing Relational Graph Neural Networks in GPU Architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24).* Association for Computing Machinery, New York, NY, USA, 528–544. doi:10.1145/3620666.3651322

[50] 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* (Online Event, United Kingdom) *(EuroSys '21).* Association for Computing Machinery, New York, NY, USA, 359–375. doi:10.1145/3447786.3456247

[51] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 515–528. https://proceedings.mlsys.org/paper/2022/file/a87ff679a2f3e71d9181a67b7542122c-Paper.pdf

[52] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[53] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Characterizing and Understanding GCNs on GPU. *IEEE Computer Architecture Letters* 19 (2020), 22–25.

[54] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. doi:10.1145/3582016.3582047

[55] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. 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* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 329–341. doi:10.1145/3572848.3577506

[56] 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. 4 (2022), 467–484. https://proceedings.mlsys.org/paper/2022/file/9a1158154dfa42caddbd0694a4e9bdc8-Paper.pdf

[57] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*. 5165–5175.

[58] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[59] Kaixiong Zhou, Qingquan Song, Xiao Huang, and Xia Hu. 2019. Auto-GNN: Neural Architecture Search of Graph Neural Networks. arXiv:1909.03184 [cs.LG] https://arxiv.org/abs/1909.03184

[60] 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. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 878–891. doi:10.1145/3575693.3575723

Received 2025-03-25; accepted 2025-08-12