# Accelerating Tensor-train Decomposition on Graph Neural Networks

Shenghao Qiu
*University of Leeds, U. K.*
sc19sq@leeds.ac.uk

Chunwei Xia
*University of Leeds, U. K.*
c.xia@leeds.ac.uk

Zheng Wang
*University of Leeds, U. K.*
z.wang5@leeds.ac.uk

***Abstract*—Memory footprint is a major concern when training graph neural networks (GNNs) on large graph data. Tensor-train decomposition (TTD) offers a potential solution by representing high-dimensional tensors with a set of smaller tensors, reducing memory overhead. However, existing TTD-based solutions for GNNs fail to reuse intermediate computation results and minimize memory data transfers to improve GNN performance. We introduce FALCON, a software framework to accelerate TTD-based GNN training. FALCON leverages the observation that a small subset of graph nodes with high edge degrees are frequently accessed, enabling the caching of intermediate results to reduce redundant computation and data transfers. Additionally, it incorporates multi-level graph partitioning and kernel optimization techniques to boost computational efficiency. We evaluated FALCON using three real-world datasets on three GPU platforms—NVIDIA 3090, 4090, and A100. Experimental results show that FALCON outperforms previous TTD-based frameworks, delivering a $1.3$ to $8.17\times$ improvement in throughput while maintaining comparable or better efficiencies in memory footprint and model accuracy.**

***Index Terms*—graph neural networks, code optimization, tensor-train decomposition**

## I. INTRODUCTION

While Transformer-based deep neural networks (DNNs) have achieved remarkable success in natural language and image processing [1], [2], graph neural networks (GNNs) remain a powerful tool for processing graph-structured data. Indeed, GNNs have a wide range of applications in tasks like drug discovery [3]–[6], physics simulations [7], fake news detection [8], traffic prediction [9] and recommendation systems [10], [11].

A key task of GNNs is to learn, propagate, and aggregate the vector representation (or *embeddings*) of graph nodes. The *node embeddings* capture the structure and relationships of graph nodes, enabling downstream tasks such as node classification [12]–[14], link prediction [6], and graph clustering [15]. GNNs update node embeddings based on information from neighbouring nodes in the graph. This is typically performed through multiple iterations of message passing, where each node aggregates information from its neighbours and updates its embedding accordingly. As real-life graphs often consist of millions or billions of nodes [16]–[18] and a single node embedding can have hundreds of elements, learning embeddings for individual graph nodes incurs a significant memory footprint. This is a particular problem for running GNNs on GPUs with limited memory, restricting the scale of data a GNN can effectively process.

Various attempts have been made to reduce the GPU memory overhead of GNNs during training. These include offloading some parameters and data from the GPU to CPU memory [19], [20] and sampling a subset of nodes or edges [21]. However, these methods have limitations. CPU-GPU offloading introduces significant performance overhead due to frequent data movement between host memory and GPU's global memory, while accuracy requirements can constrain the memory saving of graph sampling alone.

More recent studies show that tensor-train decomposition (TTD) can further reduce GNN memory usage by approximating a high-dimensional tensor with low-dimensional core tensors [22]. The core tensors, connected by tensor-train-rank (TT-rank) matrices, compress the original tensor while capturing its multidimensional interactions. TTD can be combined with graph sampling to effectively reduce the memory footprint for storing node embeddings. Our work leverages the recent development in TTD to reduce the memory footprint of GNN training.

While TTD offers the potential to reduce the memory footprint of GNNs, applying TTD to GNN training poses two challenges. Firstly, compressing the original large tensor into smaller core tensors risks information loss, where even minor inaccuracies in node embeddings can accumulate, leading to a significant decline in model accuracy. Secondly, the TTD computational overhead is large due to the need for dynamic reconstruction of the original tensor from its core tensors. This process, although manageable in systems like recommendation engines [23], where the embedding table is reconstructed once per batch, becomes a substantial burden in GNN training. The frequent updates and recomputations of the node embedding table in GNNs during iterative message-passing cycles significantly increase computational demands, which in turn limits the scalability of TTD.

We present FALCON, an open-source framework to accelerate TTD-based GNN training. FALCON leverages TTD to reduce the memory footprint of the graph node embedding table while minimizing the computational overhead and accuracy degradation introduced by TTD. FALCON finds the TTD weights using multiple initialization methods to improve training convergence. It implemented auto-tuning to determine the core tensor dimensions and their permutation according to the underlying GNN architecture and the input graphs. This preprocessing step is conducted *offline* and incurs only modest overhead, taking place in the initial 3 to 4 training iterations.

To accelerate TTD computation, we exploit data reuse and optimize TTD kernel invocations. Specifically, we cache embeddings of the most frequently accessed graph nodes so that access to them does not require reconstructing the

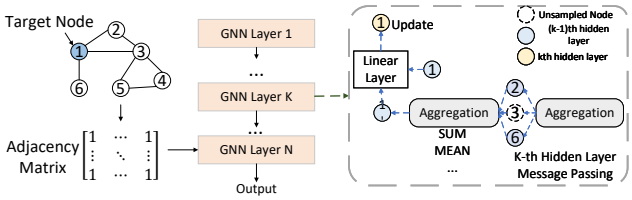
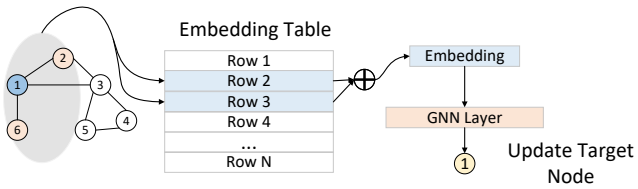
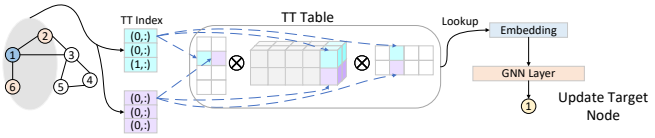



Fig. 1: General computation flow of GNNs.




(a) Lookup with standard embedding tables




(b) Lookup with TTD-based embedding tables

Fig. 2: The GNN node lookup operation with (a) standard and (b) TTD-based embedding tables.

original embedding to reduce computation overhead. Kernel optimization is applied based on the node embedding size and lookup vector size[1]. For smaller lookup vectors, we buffer the tensors to avoid additional kernel launch overhead. We also combine the *matrix multiplication (GEMM)* and *tensor core updates* to eliminate intermediate global memory writes, thereby minimizing the number of kernel launches and enhancing overall efficiency. Our new kernel maintains good embedded lookup performance while leveraging TTD to reduce the memory footprint of graph processing.

We evaluated FALCON by applying it to three node classification tasks using the Open Graph Benchmark (OGB) datasets [18]. We tested FALCON on three GPU platforms: NVIDIA GeForce RTX 3090, RTX 4090, and A100 GPUs. We compared FALCON against the state-of-the-art TTD GNN implementation in Nimble [22], [23]. We applied FALCON to three representative GNN architectures provided in the deep graph library (DGL) [19]: the graph convolution network (GCN) [24], the graph attention network (GAT) [25] and GraphSAGE [21]. Compared to Nimble, FALCON achieves an average speedup of $2.3\times$ across GNN architectures and datasets (up to $8.17\times$) without compromising the model accuracy while reducing the memory footprint of graph processing by orders of magnitude (up to $5,474\times$) over native non-TTD GNN implementation.

This paper makes the following contributions:

- It presents new optimizations to accelerate TTD-based GNN computation through graph partitioning (Sec. IV-A) and kernel optimizations (Sec. IV-B);
- It exploits node embedding caching and TTD kernel optimizations to reduce TTD computation overhead for GNN processing (Sec. IV-C);
- It showcases how auto-tuning techniques can be employed to search for TTD parameters to accelerate TTD computation in GNNs (Sec. IV-D).

**Online materials.** FALCON is publicly available at: https://github.com/JoshuaQSH/FALCON-TTDforGNNs.

## II. BACKGROUND

### A. Graph Neural Networks

Figure 1 depicts the standard GNN computation flow. The connectivity of graph edges is encoded within an adjacency matrix, where a value of 1 indicates a connection between two graph nodes. A GNN then propagates information through the edge connections to learn features and patterns.

When learning with GNNs, each node of the input graph is initially assigned a fixed-length vector (embedding) as its feature representation. These node embeddings are stored in an embedding matrix, where each row corresponds to the embedding of a particular node. Each node embedding is iteratively updated by aggregating information from its neighbouring nodes. This is done through performing *message passing* through multiple GNN layers, where each node first aggregates information from its neighbours and updates its embedding accordingly based on the current features of neighbouring nodes. Upon completion of this iterative process, the updated node embedding can be pooled or aggregated across the graph to obtain a global representation. Alternatively, they can be utilized directly for graph classification or link prediction tasks. When processing large-scale graphs, the node embedding matrix can incur significant memory overhead, limiting the scale of data a GNN can effectively operate on. As we will show later in the paper, even with state-of-the-art graph sampling methods, processing a graph with over 1 billion nodes would require over 20GB of GPU memory.

### B. Tensor-train Decomposition

TTD offers a solution to reduce the memory footprint of node embedding matrices [22], [26]. TTD represents a high-dimensional tensor (e.g., a node embedding) as a series of 3-dimensional *core tensors* linked in a chain-like structure [27], akin to the train carriages (hence the term "tensor-train").

**Core tensors and TT ranks.** Given a $d$-dimensional tensor $\mathcal{A}$ with dimensions $\mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \times \mathcal{I}_d$, the goal of TTD is to express $\mathcal{A}$ as a sequence of 3-dimensional *tensor core* $\{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_d\}$[2], where each tensor core $\mathcal{G}_k$ is of size

[1]A lookup vector is a numerical vector from the node embedding table.

[2]In general, $\mathcal{G}_d$ is a 4-dimensional tensor as $\mathcal{I}_d$ should satisfy $\mathcal{I}_d \in \mathbb{R}^{p_i \times q_i}$, such that $p_i$ and $q_i$ are two factorizations of the TT embedding dimensions. In practice, we often combine the third and fourth dimension of $\mathcal{G}$ and reduce $\mathcal{G}$ into a three dimensional matrix for simplicity.

**Algorithm 1:** TTD in GNN node embeddings

**Input:** Embedding Tensor $\mathcal{W} \in \mathbb{R}^{M \times N}$
**Output:** Tensor train cores $\mathcal{G}_1, \mathcal{G}_2, \ldots \mathcal{G}_d$ described in Eq.II-B

```
// Initialization
```
1 $\mathcal{W}$ = reshape($[p_1, p_2, \ldots, p_d, q_1, q_2, \ldots, q_k, q_d]$)
2 temp = transpose($\mathcal{W}$, $[p_1, q_1, p_2, q_2, \ldots, p_d, q_d]$)
3 **for** *(i=0; i<d; i++)* **do**
4  temp = reshape($[r_i * p_i * q_i, :]$)
  // A TT SVD follows: temp $= U\Sigma V^T + E$
5  temp = TT_SVD()
6  $\mathcal{G}_d$ = reshape(U, $[r_i, p_i, q_i, p_{i+1}]$)
7 **end**
8 $\mathcal{G}_d$ = reshape(temp, [1, $p_d, \prod_d^{i=1}(r_d) * q_d$])
9 return $(\mathcal{G}_1, \mathcal{G}_2, \ldots \mathcal{G}_d)$

$\mathcal{R}_{k-1} \times \mathcal{I}_k \times \mathcal{R}_k$. Here, $\mathcal{R}_0$ and $\mathcal{R}_d$ are usually set to 1, and the dimensions $\mathcal{R}_k$ for $k = 1, \ldots, d-1$ are known as the *TT ranks*. Each tensor core $\mathcal{G}_k$ captures the interactions along a particular mode (dimension) of the original tensor, with the TT ranks $\mathcal{R}_k$ controlling the level of detail retained from the original tensor. Because the aggregated size of the core tensors is smaller than the original tensor, TTD can reduce the memory footprint of the original tensors through approximation. Essentially, the TT ranks balance decomposition accuracy with storage space savings. Choosing a high TT rank enhances the precision of TTD approximation but decreases storage efficiency and also slows down the model runtime (see also Section V-A).

**Tensor reconstruction.** With TTD, the original tensor $\mathcal{A}$ needs to be reconstructed, during DNN training and inference, as a product of matrices from the core tensors:

$$x_{i_1,i_2,\ldots,i_d} = \mathcal{G}_1[:, i_1, :] \cdot \mathcal{G}_2[:, i_2, :] \cdot \ldots \cdot \mathcal{G}_d[:, i_d, :] \quad (1)$$

where $x_{i_1,i_2,\ldots,i_d}$ is an element of the targeted tensor $\mathcal{A}$ and $\mathcal{G}_1[:, i_k, :]$ denotes the matrix obtained by fixing the second index of the $k$-th core tensor to $i_k$.

### C. Node Embedding Table Lookup

Recall that training a GNN involves updating each node's embedding based on the embeddings of its neighbors (Sec. II-A). This process includes retrieving the embeddings of neighboring nodes from the embedding table and applying an aggregation function (e.g., summation or mean) to compute a new representation for the target node.

Figure 2 compares node lookup for updating node 1's embedding with embeddings of nodes 2 and 3, using standard Non-TTD and TTD-based embedding tables. In the Non-TTD (Figure 2a), the node index is used to retrieve the corresponding embeddings (rows 2 and 3), which are then aggregated with the target node's embedding (node 1).

For a TTD-based embedding table (Figure 2b), the node index is converted into TT indices to retrieve the core tensors $\mathcal{G}_k$, which are used to reconstruct the node embedding vector. Algorithm 1 details this process, starting with an embedding table of size $M \times N$. Each TT core is derived from the tensor's singular value decomposition (SVD) [27], then reshaped into a 4-dimensional format (lines 4-6), with customizable dimensions (line 1). Finally, $\mathcal{G}_d$ is reshaped into a 3-dimensional tensor (line 8) to form the GNN embedding layer.

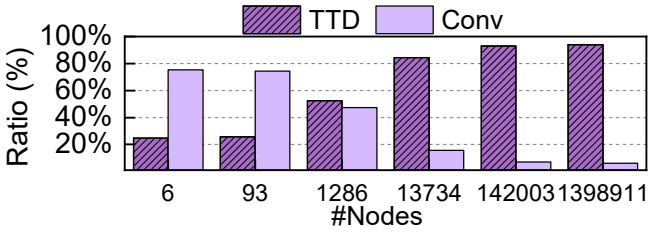



Fig. 3: TTD kernel computation w.r.t. the end-to-end GNN computation time as the number of graph nodes increases.

### D. Graph Sampling

A common practice to reduce the memory footprint of GNNs is to sample only a subset of edges and nodes of the input graph [1], [21], [28]–[32]. TTD is orthogonal to these sampling techniques. In this work, we demonstrate that our approach can be integrated with sampling-based GNNs, using GraphSAGE as a case study. Specifically, GraphSAGE implements a neighbourhood sampling method to use a subset of neighbouring nodes to update the target node's embedding. During backward propagation, GraphSAGE groups the sampled graph nodes into multiple *minibatches* before computing each minibatch's gradients, where each minibatch contains a set of nodes to update the node feature. This strategy reduces computational overhead while maintaining the local neighbourhood structure essential for learning node representations.

## III. Motivation

TTD trades computation overhead for memory saving because the original tensor needs to be dynamically reconstructed. It can introduce considerable computation overhead during graph embedding lookup.

**TTD computation overhead.** Figure 3 shows the time spent by the TTD kernel to reconstruct the node embedding tensor during the forward and backward process of GNN training, compared to a typical graph convolution layer. In this example, we consider a one-layer GraphSAGE, where the main GNN kernel is shown as `Conv`, and the TTD kernels are represented as `TTD`. We consider a fixed-sized graph embedding table (32GB) by varying the number of input graph nodes to simulate the embedding table lookup process. The experiment was performed on an NVIDIA 3090 GPU using a highly optimized, standard TTD implementation from the *Meta FBTT* library [23]. In this experiment, we represent a node embedding using two core tensors with $r_1$ and $r_2$ set to 16. As can be seen from the diagram, the TTD kernel can account for 25% to 93% of the end-to-end GNN training time, which grows as the input graph becomes larger, highlighting the need to accelerate TTD to process large graphs.

**Graph edge distribution.** FALCON leverages the observation that in real-world graphs, a small proportion of nodes exhibit high connectivity and are frequently accessed during node

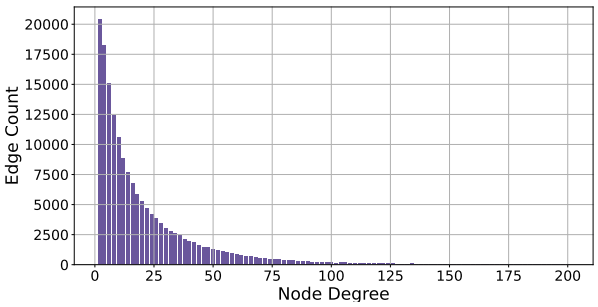



Fig. 4: Node degree distribution with *ogbn-arxiv* dataset.

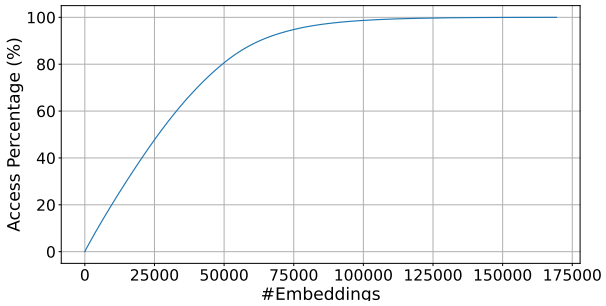



Fig. 5: *ogbn-arxiv* dataset cumulative row accessing percentage while embedding lookup.

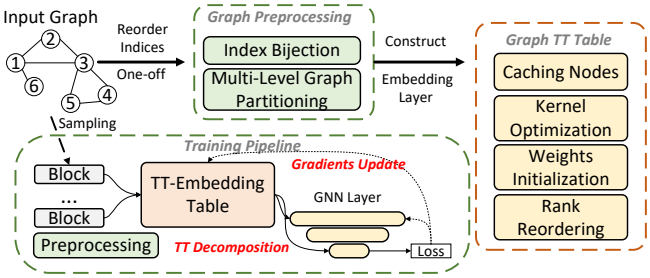



Fig. 6: FALCON overview and training workflow.

```
1  import torch.nn as nn
2  import dgl.nn.pytorch as dglnn
3  +import falcon
4
5  class GNNModel(nn.Module):
6      def __init__(self, *(parameters)):
7          super().__init__()
8          # GNN model layer, GraphSAGE as an example
9          self.layer = dglnn.SAGEConv(in_feat, n_hidden
               )
10         ...
11 +        self.emb_layer = falcon.EmbeddingBag(
12 +            num_embeddings = num_nodes,
13 +            embedding_dim = in_feat,
14 +            num_rank = 2, # Num. tt_cores
15 +            cached = True, # caching the nodes
16 +            fusion = True, # kernel fusion
17 +            ...
18 +        )
```

Listing 1: GNN model definition with FALCON.

embedding lookups. As a concrete example, Figures 4 and 5 report the node degree distribution and the cumulative row access percentage during the embedding lookup of GNN training, respectively, for the ogbn-arxiv dataset [18]. This dataset collects the citation network among Computer Science (CS) arXiv papers. From Figure 4, we observe a skewed node edge connectivity distribution, where less than 1.2% of nodes have a node degree greater than 100. Consequently, as shown in Figure 5, 40% of the nodes account for 90% of the accesses to the embedding table during GNN training. By caching the embeddings of these frequently accessed nodes, we can reduce the TTD overhead for reconstructing node embeddings, lowering the overall TTD computation overhead.

**Excessive TTD kernel invocations.** In previous TTD implementations [22], [23], [26], [33], each TTD kernel handles a fixed number of graph nodes. This fix-sized computation kernel strategy leads to low GPU utilization and poor scalability. Our profiling results show that this strategy is ill-suited for GNN training, where the number of neighbor nodes sampled at each layer can vary. As a result, this approach requires multiple kernel invocations for different sizes of sampled nodes, increasing overhead and reducing computational efficiency. FALCON addresses this issue by implementing TTD kernels that can process a dynamic number of embedding queries and kernel fusions for better TTD computation.

## IV. OUR APPROACH

Figure 6 shows how FALCON integrates with the GNN training pipeline during offline preprocessing and online training. In the *one-off* preprocessing step, FALCON partitions the input graph to enhance data locality using index mapping and multi-level graph partitioning (Sec. IV-A). The node embeddings are sorted into an ordered table and then mapped to the TTD table.

FALCON integrates a range of optimizations to accelerate TTD computation for GNN training. In the forward phase, we buffer and batch kernel inputs to reduce kernel launching overhead (Sec. IV-B) and cache embedding tensors of frequently accessed nodes (Sec. IV-C) to accelerate TTD computation. In the backward phase, we introduce an optimized *TTD updates kernel* (Sec. IV-B) to improve memory utilization. Finally, we employ auto-tuning to choose the TT rank order and the optimal tensor core weight initialization (Sec. IV-D).

**Implementation.** We have packaged all our optimizations into Python packages and provide APIs similar to the PyTorch `torch.nn.EmbeddingBag()` interface. Listings 1 depicts how FALCON methods can be utilized in the GNN model definition. For general GNN training, the user needs to invoke FALCON in the model settings (lines 14 in Listing 1) before model training. The `self.emb_layer()` (Line 11) is just the same as what the user creates with `torch.nn.EmbeddingBag()` and can be

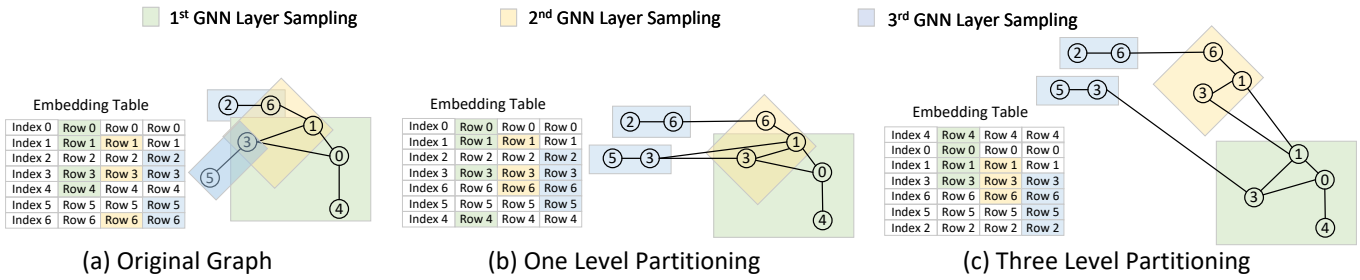



Fig. 7: An input graph partitioning example with graph sampling method. (a) is the original input graph, (b) shows the one-level METIS partitioning method, and (c) shows the three-level customized partitioning approach. The shared nodes (e.g., *node 6* and *node 3*) are boundary nodes and are referenced to nodes in other partitions.

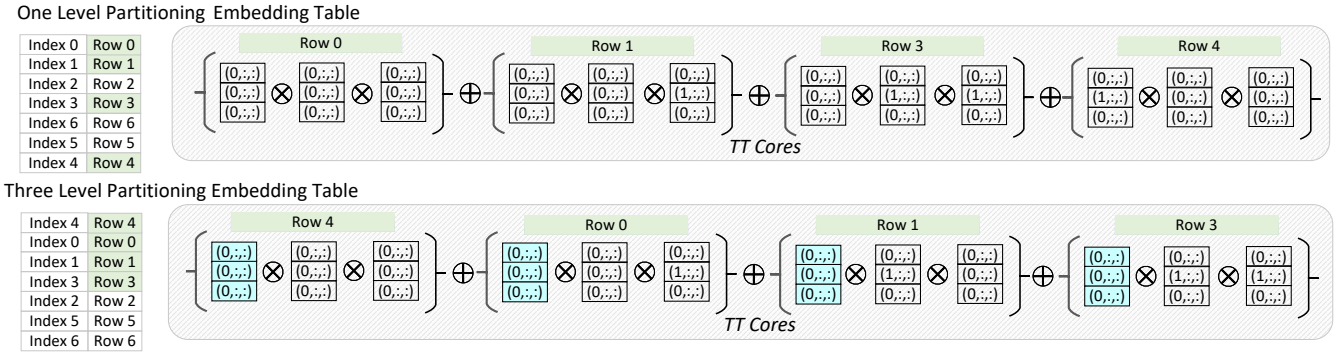
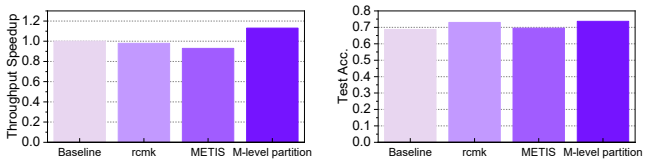



Fig. 8: An example of updating node 0 from Figure 7. We use the $[2, 2, 2]$ index system to store all the TT cores index. E.g., node 0 from Figure 7 represents *Row 0* from the embedding table and will be further mapped as $\mathcal{G}_1[0, :, :] \times \mathcal{G}_2[0, :, :] \times \mathcal{G}_3[0, :, :]$.




(a) End2End Performance (b) Test Accuracy

Fig. 9: GraphSAGE performance on the *ogbn-products* dataset with different partitioning methods. The baseline is GraphSAGE without graph partitioning.

further called before passing to GNN layers. FALCON will then automatically manage the rest of the GNN training pipeline.

### A. *Offline Graph Partitioning*

For an input graph, FALCON provides an API to partition the graph, improving data locality and aligning graph topology with TTD. The aim is to create disjoint subgraphs that maximize internal connections and minimize external ones. During the GNN's *Aggregation* process, node states within a graph tend to show similarity, especially with high-degree nodes requiring frequent access. Proper graph partitioning enhances data locality and improves the embedding table lookup performance during GNN training.

We implement an *edge-cut*[3] strategy, which assigns nodes to specific partitions and stores the result in a JSON file with a small overhead[4]. GNNs update node embeddings by traversing edges to aggregate neighboring embeddings. To improve GPU memory locality, nodes and their neighbors are placed in consecutive locations in the embedding table after partitioning the graph. A bijective mapping array is used to link node indices to embedding table indices. Although this introduces an indirect memory reference, subsequent accesses to larger tensor cores improve consecutive memory access and cache locality, enhancing performance.

Figures 7 and 8 illustrate the differences between our graph partitioning method and conventional approaches. Figure 7 shows a three-layer graph sampling method and the resulting node embedding structure, where four nodes are sampled in the first layer, three in the second, and two in the final layer. To update node 0 according to the sampling rules, the graph

[3]Each node is assigned to a specific partition (subgraph), which shares a specific number of edges between them.

[4]The overhead depends on the size of the dataset. As for the largest dataset used in this paper (*ogbn-paper100M*), it can take around 10 training epochs.

*Aggregations* will proceed as follows:

$$N_0 \leftarrow Aggregate([N_0, N_1, N_3, N_4]) \quad (2)$$
$$N_1 \leftarrow Aggregate([N_1, N_3, N_6]) \quad (3)$$
$$N_3 \leftarrow Aggregate([N_3, N_5]) \quad (4)$$
$$N_6 \leftarrow Aggregate([N_6, N_2]) \quad (5)$$

In this setup, using a standard graph partitioning algorithm like METIS (Figure 7b) improves data locality for the third layer, where embedding vectors of sampled nodes at the 3rd layer are stored in consecutive order in the embedding table. However, the embedding tables for the first and second layers are not optimized for memory access. Standard graph partitioning algorithms are designed to minimize edge cuts and balance partitions but do not optimize memory access patterns across multiple GNN layers with different sampling patterns. FALCON performs multi-level partitioning (Figure 7c), aligning embedding vector memory layouts across all layers and improving the data locality across GNN layers for full-graph processing.

Figure 8 illustrates the process of mapping the embedding table to the TT core index system. Using a $[2, 2, 2]$ indexing scheme for all nodes, this structure aligns with three tensor cores that update node embeddings. The mapping is represented as:

$$N_i = \mathcal{G}_1[:, i_1, :] \times \mathcal{G}_2[:, i_2, :] \times \mathcal{G}_3[:, i_3, :] \quad (6)$$

where $i_{th}$ is a node's index decomposed as $[i_1, i_2, i_3]$, corresponding to indices in the tensor cores. Figure 8 highlights how the first layer of the embedding vector is organized into TT cores. If the TT cores are optimally structured, shared TT indices (e.g., $\mathcal{G}_1[0, :, :]$) can be effectively cached across embedding vectors.

Figure 9 presents an end-to-end GNN performance on the *ogbn-products* dataset using different graph partitioning methods. In Figure 9a, we compare the TTD training times for two representative one-level partitioning methods: METIS and Recursive Coordinate Metis-Kway (rcmk) [34]. We observe that effective graph partitioning should 1) follow multi-level partitioning based on GNN layers and 2) align with the TTD factorization $p_i$ (see Figures 7c and 8). The multi-level partitioning strategy employed by FALCON, shown in Figure 9, is tailored to the graph's factorization, yielding an average $1.15\times$ speedup and improved test accuracy due to better convergence. We note that while Figure 9 shows how our approach can be applied to a 3-layer GNN using a 3-level partitioning, our optimization can be a GNN with an arbitrary number of layers and support multi-level partitioning.

We stress that graph partitioning takes place *offline* during preprocessing before training. Its overhead is comparable to 2 to 5 training epochs and hence is negligible in typical GNN training that requires hundreds or thousands of training epochs. Given the structure of a GNN, *Sampling* and *Aggregation* occur at each layer, so an effective partitioning method should ensure $P = L$, where $P$ represents the partitioning levels and $L$ corresponds to the number of GNN layer.

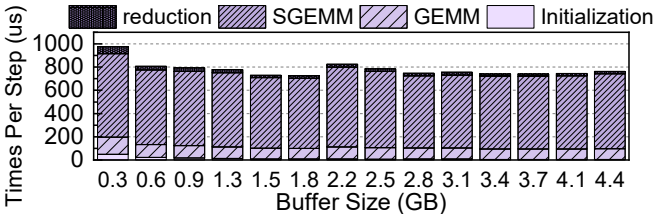



Fig. 10: TTD breakdowns with forward optimization.

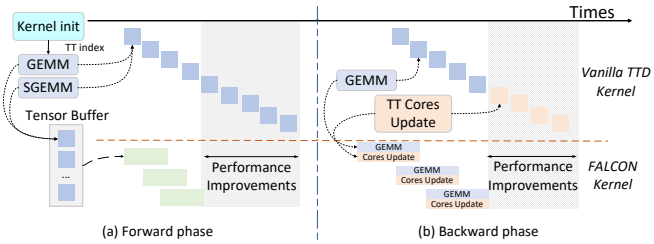



Fig. 11: Overview of our kernel optimization. The forward (a) and backward(b) phases show a one-step walk of a GNN+TTD training iteration.

### B. TTD Kernel Optimization

As discussed in Sec. III, the current TTD kernel implementation assumes a fixed input size, leading to poor GPU utilization during GNN embedding table lookups. FALCON addresses this by batching input processing in the TTD forward phase and introducing kernel fusion in the backward phase. As shown in Figure 10, a TTD kernel has a four-stage pipeline: initialization, general matrix multiplication (GEMM), sparse GEMM (SGEMM), and reduction. Each GNN layer updates node embeddings using neighbor sampling from the previous layer, forming a chain of TTD kernel calls. However, standard TTD implementations like the Nimble TTD kernels are not dynamically resized for GEMM and SGEMM, resulting in redundant kernel calls. With graph sampling, each layer processes a subgraph by retrieving varying numbers of embedding vectors. Using a fixed kernel size is inefficient for this computing pattern, resulting in computational redundancy and synchronization delays.

**Forward optimization.** FALCON reduces GEMM and SGEMM kernel invocations by first buffering the tensor data and then launching the kernels to process the buffered data. Figure 11a outlines our forward optimization. We create a *buffer* in the GPU memory to adapt to various input node sizes. Figure 11 depicts how this improves performance, with GEMM and SGEMM only triggered when the tensor size reaches the *buffer* threshold. We record the first few steps of the forward running (similar to a warmup stage) and choose the optimal buffer threshold. This happens for each GNN layer forward pass. Figure 10 provides a profiling example of forward kernel optimization. In a typical GNN, most of the computational cost during the forward phase comes from GEMM and SGEMM, depending on the number of nodes

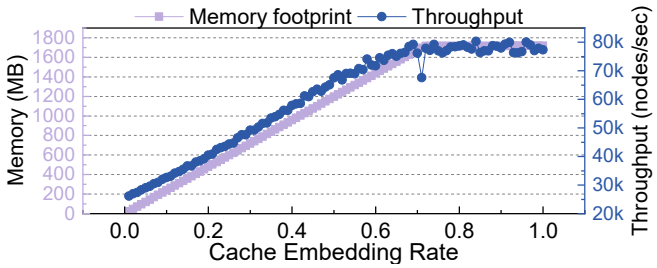



Fig. 12: GraphSAGE performance on the *ogbn-products* dataset with different caching size. The x-axis shows the cache size (with up to 100% of the embedding table for training a subgraph). The left y-axis shows the memory footprint of our caching scheme and the right-axis gives the graph processing throughput.

accessed. For instance, profiling with a 32GB embedding table and 10,000 node accesses per step results in 11 GEMM and SGEMM computations. As the number of kernel invocations increases, the computation overhead grows.

Our method uses a temporary buffer to store TTD core tensors, deferring kernel launches until the buffer is full. This strategy significantly reduces kernel launch times (e.g., from 11 to 1 per step for a 32GB embedding table with 10,000 node accesses). While it increases per-kernel execution time, it minimizes initialization and launch overhead by reducing the total number of kernel invocations. Notably, kernel launching accounts for 24% of the end-to-end execution time of GEMM, and by lowering the number of kernel invocations from 11 to 1 for GEMM computation, our optimization achieves a 1.63× speedup over the standard TTD implementation for GEMM on an NVIDIA 3090 GPU. Furthermore, as shown in Figure 10, larger buffers do not always improve performance; there is a tradeoff between average kernel cost and overall efficiency, depending on the input node size. In our setup, we allocate 1-2% of the node size for buffering (e.g., a 600MB buffer for a 32GB embedding table).

**TT backward optimization.** Using TTD for node embeddings, the backward pass of GNN training consists of multiple stages of GEMM and TT core updates. This process is shown in Figure 11b. Here, the unoptimized approach (Nimble and others) handles the backward pass separately: first, GEMM operations compute the gradient-related values, followed by a subsequent kernel that updates the tensor cores (TT cores). To address this, we propose a new *TTD update kernel*, where the GEMM and TT core update kernels are fused within the backward propagation loop. In the previous Nimble approach, the intermediate results from GEMM were often written to global memory and then read again for the TT core updates. By fusing the GEMM and TT core update operations into a single CUDA kernel, we eliminate intermediate memory writes and redundant kernel launches, improving efficiency.

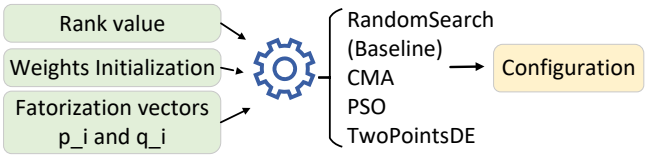



Fig. 13: Auto-tuning methods and parameters used by FALCON to choose the TTD parameters.

TABLE I: Tunable parameters considered by FALCON

| Tuning Parameters | Range | Total Num. |
|---|---|---|
| Rank | [1, 1] - [256, 256] | 65,536 |
| Factorization $p$ | [1,1,1] - [256, 256, 256] | 16,777,216 |
| Factorization $q$ | [1,1,1] - [256, 256, 256] | 16,777,216 |
| Weights | ['Gaussian', 'Eigen Decomposition', 'Orthogonal Decomposition'] | 3 |

### C. Caching Graph Nodes

TTD introduces computational overhead by recomputing intermediate results during both forward and backpropagation. To reduce recomputations, FALCON stores forward pass intermediates of frequently accessed, high-degree nodes. Based on insights from Figures 4 and 5, which show unbalanced graph degree distributions and sparse embeddings, we cache embeddings for high-degree nodes as non-TTD tensors in GPU memory. The cache is a dedicated, dynamically adjusted GPU buffer, fine-tuned throughout training to balance memory usage and computation cost to maximize memory efficiency.

To initialize the software cache, we traverse the graph (or a sampled subgraph) during the first two training epochs to cache the top 10% of nodes by degree. As shown in Figure 12, caching more nodes improves performance but increases memory usage. To accommodate diverse graph structures and usage patterns, FALCON provides a customizable interface to adjust cache size, optimizing GPU memory usage. To minimize caching overhead, we use up to 5% of GPU memory for caching. For GraphSAGE with sampled subgraphs, we cache around 50% of the nodes, achieving a throughput improvement of 2.65×. For full-neighbor global training, we cache 5–10% of the nodes for the same improvement. Users can increase the cache size for up to 3× performance improvement.

We manage the changing access patterns, especially in deeper GNN layers, using a hash table with a Least Frequently Used (LFU) algorithm to track access frequencies. This table stores embeddings, access counts, and GPU caching states. During forward passes, cache hits retrieve embeddings directly, while misses trigger TTD computation. Embedding lookups are faster than TTD, so the reduction in computation outweighs cache management overhead. This cache is automatically managed by the FALCON runtime and is transparent to the user.

### D. TTD Parameter Tuning

FALCON searches for the optimal parameters using the *nevergrad* library [35]. Specifically, it explores combinations of the tensor-train weight initialization method, rank, and factorization values $p_i$ and $q_i$ to enhance performance. As shown in Figure 13, FALCON leverages multiple algorithms like CMA and PSO to find the best configurations for TTD, aiming to reduce running times without sacrificing accuracy. Our objective function is defined as:

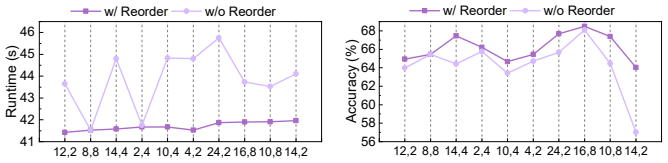
$$Obj = min\left((1-\lambda)\frac{Loss}{Loss_{max}} + \lambda\frac{T}{T_{max}}\right) \quad (7)$$

where $\lambda$ controls the trade-off between accuracy and runtime. Increasing $\lambda$ shifts the focus towards minimizing runtime while still aiming to achieve high accuracy. We empirically set $\lambda$ to 0.2 in this work.

Table I summarizes the tunable parameters. Our search strategy considers four default optimization algorithms: RandomSearch, Covariance Matrix Adaptation (CMA) [36], Particle Swarm Optimization (PSO) [37], and the Differential Evolution algorithm (TwoPointsDE) [38], which are shown to be useful in prior performance tuning works. Additional customized algorithms can be incorporated based on specific needs. These algorithms are combined in a hybrid manner, with strategies switching during the optimization process. Nevergrad allocates the computational budget (in terms of the number of configurations to be evaluated) across different algorithms, balancing exploration and exploitation, and dynamically adjusts which algorithm to prioritize based on intermediate results throughout the optimization.

**TTD weights initialization.** Previous research shows that initializing TTD weights using orthogonal initialization (i.e., ensuring the shaped matrices of TTD core tensors have orthogonal columns or rows) instead of the more commonly used Gaussian initialization can speed up convergence and reduce the reliance on network width for effective training [39]. This is because orthogonal initialization helps preserve the magnitude of gradients, avoiding issues like vanishing or exploding gradients during training. To ensure these benefits, we initialize the core tensors (TT-cores) in a way that guarantees the orthogonality of the product embedding matrix $\mathcal{W}$. Our system also offers tunable initialization methods: (1) Gaussian distribution, (2) eigen decomposition, and (3) orthogonal decomposition, providing flexibility to choose the best method for each GNN task.

**TTD rank reordering.** Due to the characteristics of the input graph, the rank order of the tensors will also affect the final performance after the small tensors are divided by TTD. As shown in Figure 14, a reordered TTD rank will affect both the running time and model performance. The data was collected by applying GraphSAGE to the *ogbn-product* dataset. For example, choosing $[12, 2]$ and $[2, 12]$ may affect the computation complexity and lead to up to 20% drop in runtime performance and test accuracy.




(a) Runtime Performance (b) Test Accuracy

Fig. 14: Top 10 performance with different reordering and rank numbers by applying GraphSAGE to the *ogbn-products* dataset. (a) indicates the runtime per epoch for each rank pair with (and without) reordering method, respectively, while (b) provides the test accuracy.

TABLE II: Hardware platforms used in evaluation

| GPU | CUDA ver. | GPU mem. (GB) | Mem. bandwidth (GB/s) |
|---|---|---|---|
| NVIDIA 3090 | V12 | 24 | 935.8 |
| NVIDIA 4090 | V12 | 24 | 1008 |
| NVIDIA A100 | V12 | 80 | 2039 |

## V. EXPERIMENTAL SETUP

**Evaluation platforms.** As listed in Table II, we evaluate FALCON on three NVIDIA GPU platforms. We use Nvidia CUDA Toolkit version 12.0 with driver version 525.60.13. All platforms have 2x 20-core Intel Xeon Gold 5218R CPUs @ 2.10GHz and 128 GB of CPU RAM.

**GNN models.** We consider three representative GNN architectures: GCN, GAT and sampling-based GraphSAGE, following the OGB leaderboard configurations [40]. Each model has three GNN layers with a hidden dimension of 256. For GraphSAGE, we use the default sampling rate, selecting 3, 5, and 15 neighbors at the first, second, and third layers, respectively. All implementations are sourced from DGL [19].

**Datasets.** We use three graph datasets from the OGB suite [18] to train and test GNN models for node classification, with dataset statistics shown in Table III. GCN and GAT are applied to the *ogbn-arxiv* dataset using full-batch training, while GraphSAGE is used with sampling techniques for the larger *ogbn-products* and *ogbn-papers100M* datasets.

**Evaluation baselines and metrics.** We compare FALCON against the closely related work, Nimble GNN [22], which employs the standard TTD implementation or GNN. Since our work aims to accelerate TTD-based GNNs, we evaluate FALCON and the baseline on model accuracy, training throughput (measured as the number of graph nodes processed per second) and running time under a similar embedding table compression rate. For the *ogbn-arxiv* dataset, we can use the full-batch training (without TTD) as the upper-bound accuracy.

### A. Overall Results

Table IV compares the performance of FALCON with the standard node embedding method without TTD (Non-TTD)

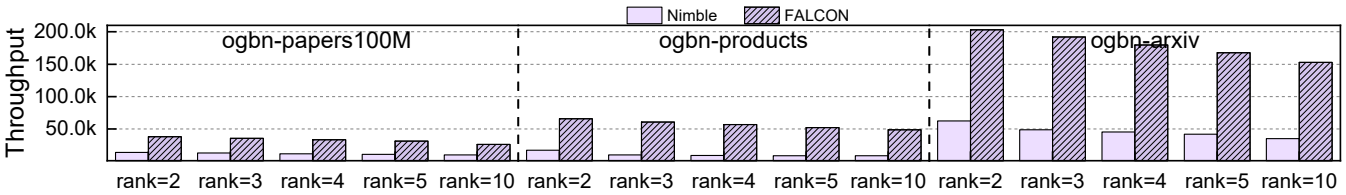



Fig. 15: GraphSAGE end-to-end performance on NVIDIA 3090, the higher the throughput the better.

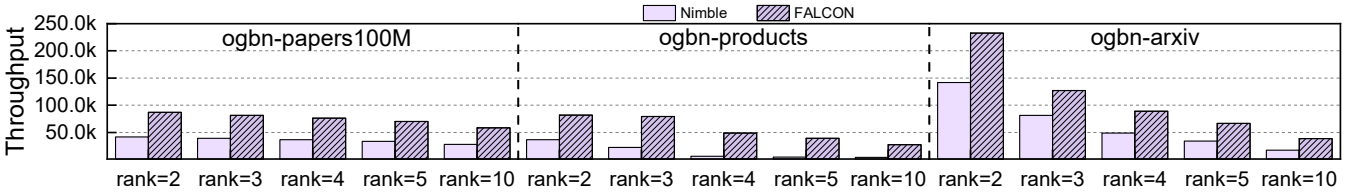



Fig. 16: GraphSAGE end-to-end performance on NVIDIA 4090, the higher the throughput the better.

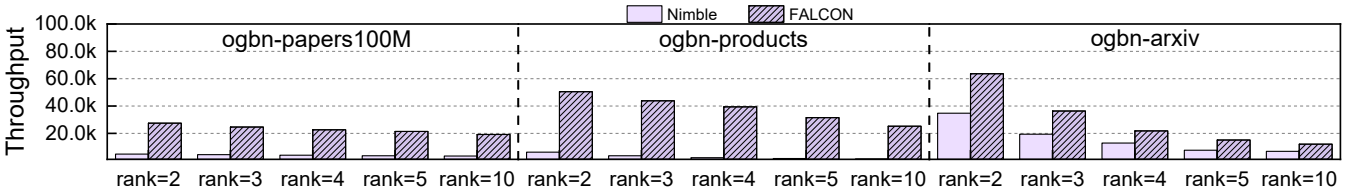



Fig. 17: GraphSAGE end-to-end performance on Nvidia A100, the higher the throughput the better.

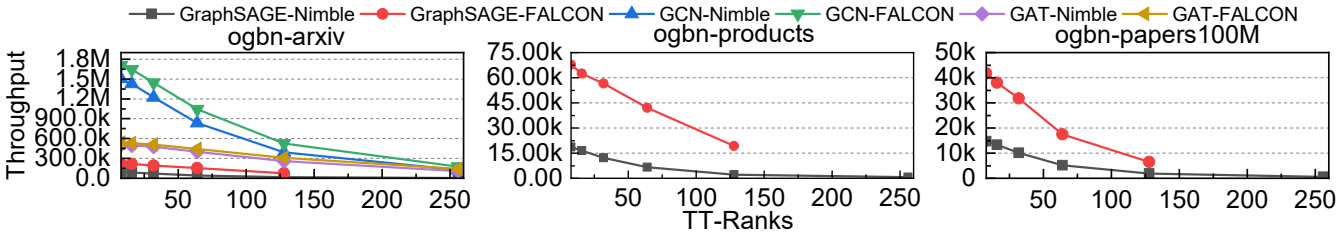



Fig. 18: Performance of GNN models with different TT ranks on NVIDIA 3090. We exclude training using a high TT rank such as $[256, 256]$ on large datasets like *ogbn-papers100M* as this leads to an out-of-memory error.

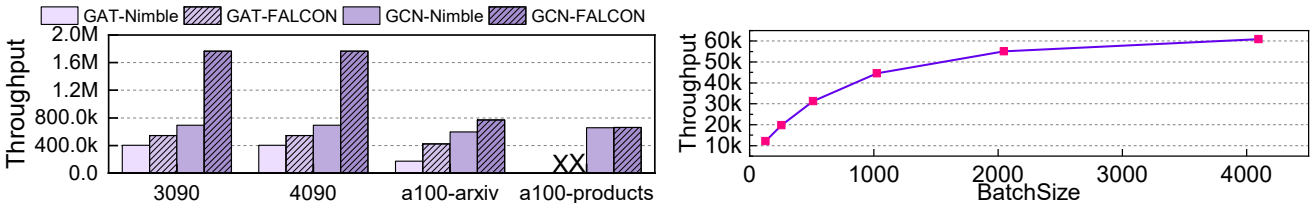
TABLE III: Graph datasets used in the evaluation.

| Dataset | #Node | #Edge |
|---|---|---|
| ogbn-arxiv | 169,343 | 1,166,243 |
| ogbn-products | 2,449,029 | 61,859,140 |
| ogbn-papers100m | 111M | 1.615B |

TABLE IV: Overall GNN model performance.

| | Approach | Acc. (%) | Compress. Ratio |
|---|---|---|---|
| GCN on ogbn-arxiv | Non-TTD | **66.2** | 1 |
| | Nimble GNN | 64.8 | 22.9× |
| | FALCON | 65.7 | 21.75× |
| GraphSAGE on ogbn-products | Non-TTD | 74.5 | 1 |
| | Nimble GNN | 69.3 | 5,762× |
| | FALCON | **75.2** | 5,473.9× |
| GraphSage on ogbn-papers100m | Non-TTD | N/A | 1 |
| | Nimble GNN | 59.5 | 424× |
| | FALCON | **63.2** | 402.8× |

and Nimble GNN (that uses a standard TTD for the node embedding table) across various graph datasets. All GNN models are trained with the same number of iterations. For the smallest dataset (*ogbn-arxiv*), we apply GCN for full graph training, while GraphSAGE is used for the other two larger datasets. Later, we extend our evaluation to GAT. We measure the memory footprint reduction over the Non-TTD version, which we refer to as *the compression ratio*. FALCON delivers better test accuracy than Nimble due to improved convergence, with accuracy comparable to or sometimes better than the Non-TTD version. For *ogbn-products*, FALCON reduces the memory footprint of graph processing by $5,473.9\times$ while achieving the highest accuracy, outperforming Nimble's TTD implementation with only a little memory loss (around 5%).




(a) GCN and GAT performance

(b) GraphSAGE Scalability

Fig. 19: (a) GCN and GAT end2end performance with *ogbn-arxiv/products* dataset and (b) GraphSAGE scalability with *ogbn-products* dataset. The X marks the out-of-memory due to the high memory footprint of GAT.

### B. GNN Training Performance

We compare FALCON with Nimble GNN that uses the standard TTD implmentation in Figures 15 - 17. Varying TT rank configurations (by adjusting the number of TT core tensors) shows that higher TT ranks preserve accuracy but increase computational overhead, reducing throughput by about 7%, consistent with previous studies [23], [26]. For a three-layer GNN with billions of nodes and edges, a TT rank of 2 is sufficient for good accuracy. Across all TT rank settings, FALCON achieves a $1.3 \sim 8.17\times$ throughput improvement over Nimble, with smaller graphs like *ogbn-arxiv* showing greater speedups, while larger graphs like *ogbn-papers100M* show smaller gains. This is because our optimized kernel is more effective when the input node size fits within the kernel buffer. Larger GPU memory and higher bandwidth further improve performance for larger graphs, as seen with *ogbn-products* on the A100 GPU (Figure 17). We also apply FALCON to GCN and GAT in Figure 19(a), where GCN shows a $2.54\times$ throughput improvement, and GAT achieves $1.35\times$ on the *ogbn-arxiv* dataset. Both GCN and GAT, trained in full batches without sampling, benefit less from our TTD optimizations since their hidden layers impact TTD computation less than GraphSAGE. The advantages of FALCON mainly come from two key strategies: 1) caching key nodes to reduce TTD recomputation, and 2) kernel optimization, which maximizes GPU utilization.

### C. Portability and Scalability

We assess the impact of TT rank settings by varying ranks from $[8, 8]$ to $[256, 256]$ (Figure 18) using GNN models on the *ogbn-arxiv* dataset. FALCON consistently outperforms Nimble across all TT ranks, with even greater advantages on larger graphs. Notably, performance gains are significant on *ogbn-products* and *ogbn-papers100M* with GraphSAGE. While higher TT ranks increase memory and computation costs, FALCON sustains good performance. Additionally, we evaluate its scalability with GraphSAGE under varying batch sizes (Figure 19b), where FALCON exhibits good scalability.

### D. Perofmrance Breakdown

We now examine the impact of individual optimizations in FALCON using the *ogbn-products* dataset to quantify their benefits. While different graph datasets may yield varying performance improvements, the relative contribution of each technique remains similar. As shown in Figure 20, our approach achieves an overall speedup of $6.2\times$. Techniques like rank reordering and weight initialization contribute approximately $1.1\times$ speedup, while graph partitioning and backward kernel fusion provide around $1.2\times$ improvement. The most significant gains come from caching and kernel optimization, each delivering about $1.5\times$ speedup. Additionally, we ensure that caching and kernel optimizations incur minimal memory overhead to maintain a compression rate comparable to SOTA methods. In our experiments, kernel optimization introduces only around 2% overhead in overall GPU usage, and caching takes around 5%. Furthermore, increasing the cache size offers the potential for further performance gains, achieving up to a $3\times$ speedup (as shown in Figure 12).

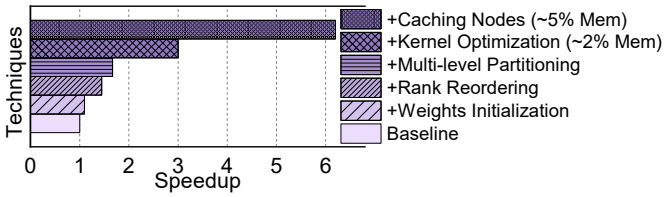



Fig. 20: Speedup breakdown and memory usage for GraphSAGE in *ogbn-products* on Nvidia 3090.

## VI. Related Work

Our work builds upon the following past foundations.

**TTD for GNNs.** TTD was introduced in [27] and later employed in [41] to compress and calculate the weight tensors within each linear layer of a neural network. Subsequently, Nimble GNN Embedding [22], derived from FBTT-Embedding, attempted to optimize GNNs using TTD. It introduced two algorithms: one for TT matrix decomposition and another for orthogonal initialization of TT cores. TT-GNN [26] targets hardware-level optimization, using TTD to compress the graph embedding matrix to enable model training fully within on-chip memory. However, these methods overlook the unique characteristics of graph data for memory efficiency optimizations and fail to optimize the TTD kernel. Our approach aims to fill this gap by leveraging graph connectivity to improve memory access efficiency to the node embedding table and optimizing GNN TTD kernels for performance.

**TTD optimizations.** The work presented in [42] shows that TTD-based DNN model inference has redundant computations. ETTE [33] proposes a new tensor core construction and computation ordering mechanism to simultaneously reduce stage-wise computation and memory overhead. TTD has also been applied in NLP [43], [44]. Recent work [39] shows that proper weight initialization can improve the training convergence when using TTD with DNNs. Additionally, TTD is used to reduce the memory footprint of recommendation systems [23], [45]. Although FALCON is tailored for graph applications, it is theoretically orthogonal to the above methods at the system level. Our kernel optimization approach could be further extended to DLRM and NLP applications while dealing with large embedding tables.

**TTD parameters tuning.** TT-RALS [46] demonstrates how to choose optimal TT ranks. Some implementations by *tensorly*, such as [47], focus on different compression methods (CP, TT, Tucker, etc.) in convolutional and fully connected layers. [39] provides TTD convergence guarantees and assists with TTD initialization. Although TTD weights initialization is crucial for maintaining DNN training performance, it is compatible with our approach as discussed in Sec. IV-D. Additionally, since our approach results in minimal accuracy degradation, we consider the above parameter selection and weight initialization to be enhancements rather than necessities.

## VII. CONCLUSION

We have presented FALCON, an open-source software framework to accelerate TTD computation for graph node embedding lookup and update in GNN training. FALCON enhances data reuse by caching frequently accessed graph nodes and TTD kernel pre-fusion. It automatically searches offline for optimal TTD parameters to improve training convergence and implements kernel fusion for backward propagation to enhance TTD computation efficiency. We evaluated FALCON on representative graph datasets across three NVIDIA GPU platforms and three GNN architectures. Experimental results show that FALCON delivers, on average, a $2.3\times$ speedup (up to $8\times$) over the standard TTD-based GNN method. It can reduce the memory footprint of the node embedding tables by up to $5,473\times$ without compromising the accuracy.

## ACKNOWLEDGMENTS

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X018202/1 and EP/X037304/1. For any correspondence regarding this work, please contact Chunwei Xia (Email: c.xia@leeds.ac.uk) and Zheng Wang (Email: z.wang5@leeds.ac.uk).

## REFERENCES

[1] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 257–266.

[2] K. Han, Y. Wang, J. Guo, Y. Tang, and E. Wu, "Vision gnn: An image is worth graph of nodes," *Advances in neural information processing systems*, vol. 35, pp. 8291–8303, 2022.

[3] D. Jiang, Z. Wu, C.-Y. Hsieh, G. Chen, B. Liao, Z. Wang, C. Shen, D. Cao, J. Wu, and T. Hou, "Could graph neural networks learn better molecular representation for drug discovery? a comparison study of descriptor-based and graph-based models," *Journal of cheminformatics*, vol. 13, pp. 1–23, 2021.

[4] Z. Zhang, L. Chen, F. Zhong, D. Wang, J. Jiang, S. Zhang, H. Jiang, M. Zheng, and X. Li, "Graph neural network approaches for drug-target interactions," *Current Opinion in Structural Biology*, vol. 73, p. 102327, 2022.

[5] K. Shao, Y. Zhang, Y. Wen, Z. Zhang, S. He, and X. Bo, "Dti-heta: prediction of drug–target interactions based on gcn and gat on heterogeneous graph," *Briefings in Bioinformatics*, vol. 23, no. 3, p. bbac109, 2022.

[6] D. Morselli Gysi, Í. Do Valle, M. Zitnik, A. Ameli, X. Gan, O. Varol, S. D. Ghiassian, J. Patten, R. A. Davey, J. Loscalzo *et al.*, "Network medicine framework for identifying drug-repurposing opportunities for covid-19," *Proceedings of the National Academy of Sciences*, vol. 118, no. 19, p. e2025581118, 2021.

[7] P. Reiser, M. Neubert, A. Eberhard, L. Torresi, C. Zhou, C. Shao, H. Metni, C. van Hoesel, H. Schopmans, T. Sommer *et al.*, "Graph neural networks for materials science and chemistry," *Communications Materials*, vol. 3, no. 1, p. 93, 2022.

[8] H. T. Phan, N. T. Nguyen, and D. Hwang, "Fake news detection: A survey of graph neural network methods," *Applied Soft Computing*, p. 110235, 2023.

[9] A. Derrow-Pinion, J. She, D. Wong, O. Lange, T. Hester, L. Perez, M. Nunkesser, S. Lee, X. Guo, B. Wiltshire *et al.*, "Eta prediction with graph neural networks in google maps," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 3767–3776.

[10] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–37, 2022.

[11] K. Tsolaki, T. Vafeiadis, A. Nizamis, D. Ioannidis, and D. Tzovaras, "Utilizing machine learning on freight transportation and logistics applications: A review," *ICT Express*, 2022.

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

[13] Y. Wang and T. Derr, "Tree decomposed graph neural network," in *Proceedings of the 30th ACM international conference on information & knowledge management*, 2021, pp. 2040–2049.

[14] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[15] A. Tsitsulin, J. Palowitch, B. Perozzi, and E. Müller, "Graph clustering with graph neural networks," *Journal of Machine Learning Research*, vol. 24, no. 127, pp. 1–21, 2023.

[16] Y. Lee, J. Chung, and M. Rhu, "Smartsage: training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 932–945.

[17] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *arXiv preprint arXiv:1902.08730*, 2019.

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

[19] M. Wang, L. Yu, Q. G. Da Zheng, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang *et al.*, "Deep graph library: towards efficient and scalable deep learning on graphs. corr abs/1909.01315 (2019)," *arXiv preprint arXiv:1909.01315*, 2019.

[20] Y.-C. Lin, G. Deng, and V. Prasanna, "A unified cpu-gpu protocol for gnn training," *arXiv preprint arXiv:2403.17092*, 2024.

[21] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[22] C. Yin, D. Zheng, I. Nisa, C. Faloutos, G. Karypis, and R. Vuduc, "Nimble gnn embedding with tensor-train decomposition," 2022.

[23] C. Yin, B. Acun, X. Liu, and C.-J. Wu, "Tt-rec: Tensor train compression for deep learning recommendation models," 2021.

[24] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2017.

[25] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[26] Z. Qu, D. Niu, S. Li, H. Zheng, and Y. Xie, "Tt-gnn: Efficient on-chip graph neural network training via embedding reformation and hardware optimization," ser. MICRO '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 452–464. [Online]. Available: https://doi.org/10.1145/3613424.3614305

[27] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[28] X. Liu, M. Yan, L. Deng, G. Li, X. Ye, and D. Fan, "Sampling methods for efficient training of graph convolutional networks: A survey," *IEEE/CAA Journal of Automatica Sinica*, vol. 9, no. 2, pp. 205–234, 2021.

[29] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.

[30] G. Alain, A. Lamb, C. Sankar, A. Courville, and Y. Bengio, "Variance reduction in sgd by distributed importance sampling," *arXiv preprint arXiv:1511.06481*, 2015.

[31] J. Chen, T. Ma, and C. Xiao, "Fastgcn: fast learning with graph convolutional networks via importance sampling," *arXiv preprint arXiv:1801.10247*, 2018.

[32] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu, "Layer-dependent importance sampling for training deep and large graph convolutional networks," *Advances in neural information processing systems*, vol. 32, 2019.

[33] Y. Gong, M. Yin, L. Huang, J. Xiao, Y. Sui, C. Deng, and B. Yuan, "Ette: Efficient tensor-train-based computing engine for deep neural networks," ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589103

[34] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[35] P. Bennet, C. Doerr, A. Moreau, J. Rapin, F. Teytaud, and O. Teytaud, "Nevergrad: black-box optimization platform," *ACM SIGEVOlution*, vol. 14, no. 1, pp. 8–15, 2021.

[36] C. Igel, N. Hansen, and S. Roth, "Covariance matrix adaptation for multi-objective optimization," *Evolutionary computation*, vol. 15, no. 1, pp. 1–28, 2007.

[37] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4. ieee, 1995, pp. 1942–1948.

[38] K. V. Price, R. M. Storn, and J. A. Lampinen, "The differential evolution algorithm," *Differential evolution: a practical approach to global optimization*, pp. 37–134, 2005.

[39] Z. Qin, M. B. Wakin, and Z. Zhu, "Guaranteed nonconvex factorization approach for tensor train recovery," 2024.

[40] S. University. (2024) Ogb node property prediction leaderboard. Accessed: 2024-06-14. [Online]. Available: https://ogb.stanford.edu/docs/leader_nodeprop/

[41] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/6855456e2fe46a9d49d3d3af4f57443d-Paper.pdf

[42] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "Tie: energy-efficient tensor train-based inference engine for deep neural network," ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 264–278. [Online]. Available: https://doi.org/10.1145/3307650.3322258

[43] Y. Ren, B. Wang, L. Shang, X. Jiang, and Q. Liu, "Exploring extreme parameter compression for pre-trained language models," 2022.

[44] V. Chekalina, G. Novikov, J. Gusak, I. Oseledets, and A. Panchenko, "Efficient gpt model pre-training using tensor train matrix representation," 2023.

[45] Z. Wang, Y. Wang, B. Feng, D. Mudigere, B. Muthiah, and Y. Ding, "El-rec: Efficient large-scale recommendation model training via tensor-train embedding table," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–14.

[46] M. Imaizumi, T. Maehara, and K. Hayashi, "On tensor train rank minimization: Statistical efficiency and scalable algorithm," 2017.

[47] X.-Y. Liu, Y. Fang, L. Yang, Z. Li, and A. Walid, "Chapter 9 - high-performance tensor decompositions for compressing and accelerating deep neural networks," in *Tensors for Data Processing*, Y. Liu, Ed. Academic Press, 2022, pp. 293–340. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128244470000157