FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems

Yuwei Hu[†*], Zihao Ye[§], Minjie Wang[§], Jiali Yu[§], Da Zheng[§], Mu Li[§], Zheng Zhang[§], Zhiru Zhang[†], Yida Wang[§]

[†]School of ECE, Cornell University; {yh457, zhiruz}@cornell.edu
[§]Amazon Web Services; {yeziha, minjiw, yjial, dzzhen, mli, zhaz, wangyida}@amazon.com

Abstract—**Graph neural networks (GNNs) are gaining popularity as a promising approach to machine learning on graphs. Unlike traditional graph workloads where each vertex/edge is associated with a scalar, GNNs attach a feature tensor to each vertex/edge. This additional feature dimension, along with consequently more complex vertex- and edge-wise computations, has enormous implications on locality and parallelism, which existing graph processing systems fail to exploit.**

This paper proposes FeatGraph to accelerate GNN workloads by co-optimizing graph traversal and feature dimension computation. FeatGraph provides a flexible programming interface to express diverse GNN models by composing coarse-grained sparse templates with fine-grained user-defined functions (UDFs) on each vertex/edge. FeatGraph incorporates optimizations for graph traversal into the sparse templates and allows users to specify optimizations for UDFs with a feature dimension schedule (FDS). FeatGraph speeds up end-to-end GNN training and inference by up to 32× on CPU and 7× on GPU.

I. INTRODUCTION

Graph neural networks (GNNs) are gaining popularity in recent years as a promising approach to machine learning on graphs. Because of the ability to incorporate multi-dimensional features on vertices and edges as well as graph structure information into a joint embedding for downstream tasks, GNNs have shown successful applications in social network mining [1], recommender systems [2], molecule analysis [3], combinatorial optimization [4], to name a few.

Driven by this trend, specialized software frameworks are emerging to simplify the development and processing of GNN workloads. These GNN frameworks are typically built on top of existing deep learning systems. For example, NeuGraph [5] relies on TensorFlow [6]; PyTorch geometric (PyG) [7] is built upon PyTorch [8]; DGL [9] supports multiple backends.

Unlike traditional neural network workloads that are dominated by dense operations, GNN workloads consist of both dense and sparse operations. The sparsity comes from the nature of the graph that normally each vertex only connects with a small number of other vertices. Empirically, sparse operations in a GNN model account for more than 60% of the total computation time, when both the sparse and dense operations are fully optimized. While the deep learning systems have benefited from years of development in optimizing dense operations such as convolution and matrix multiplication, they lack flexible support for sparse operations that are essential
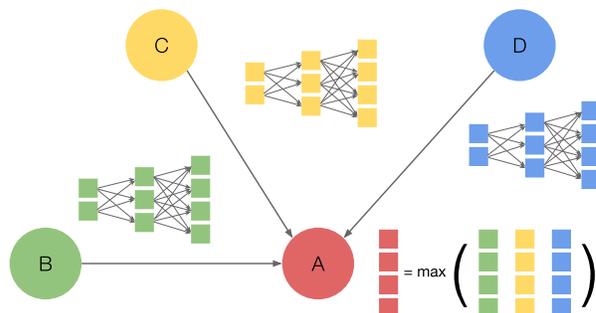
Fig. 1: Feature dimension computation makes GNNs substantially different from traditional graph workloads — Shown example is multi-layer perceptron (MLP) aggregation where each edge performs a sequence of vector matrix multiplication and non-linear activation to update the features, and the destination vertex aggregates the features (in this case, picking the maximum) as its representation.

for high-performance GNN training and inference. Specifically, the deep learning systems rely on vendor-provided sparse libraries (e.g., MKL [10], cuSPARSE [11]), which offer highly optimized implementations for only a small subset of the kernels required by diverse GNN models.

On the other hand, graph processing systems have been extensively studied in literature [12], [13], [14], [15], [16], [17], offering an alternative solution that expresses computations on graphs with a vertex- and/or edge-centric programming paradigm. As a representative attempt to circumvent the inflexibility of deep learning systems in handling sparse computations, DGL supports offloading the computation kernels in GNNs to existing graph processing systems such as Ligra [15] and Gunrock [16].

However, existing graph processing systems are not the panacea, either. They are designed for traditional graph workloads (e.g., BFS, PageRank) where each vertex is associated with a scalar, instead of a feature tensor in GNN's use case. This additional feature dimension, along with consequently more complex vertex-wise and edge-wise computations, makes kernel optimizations substantially different. For example, existing graph partitioning techniques aiming at improving cache utilization [18], [19] do not take into consideration the feature dimension; hence the entire cache could be occupied by just a

few feature tensors. Also, prior GPU graph processing systems [20], [21], [16] rarely exploit parallelism in feature dimension computation while mainly focusing on designing sophisticated load balancing methods to exploit parallelism in graph traversal.

This paper proposes *FeatGraph* to enable performant processing of GNN workloads. The key insight is that GNN workloads require co-optimizing graph traversal and feature dimension computation to achieve preferable performance. FeatGraph suits this need by the design of a two-granularity programming interface. More concretely, FeatGraph expresses the diverse variants of GNN models by composing *coarse-grained* sparse templates with *fine-grained* feature dimension computations on each vertex/edge in the form of user-defined functions (UDFs). The coarse-grained level handles traversal over the graph topology, and the fine-grained level focuses on computation over the dense feature tensor of each vertex/edge. FeatGraph provides two sparse templates: generalized *SpMM* (sparse-dense matrix multiplication) for vertex-wise computations and generalized *SDDMM* (sampled dense-dense matrix multiplication) for edge-wise computations. Here, *"generalized"* means that the templates can take different fine-grained UDFs. For example, a commonly used GNN kernel multi-layer perceptron (MLP) aggregation [22], [23], shown in Figure 1, is mapped to generalized SpMM: it calculates features by performing MLP and aggregates them by taking the max. Note that the vanilla SpMM operation corresponds to copying features and aggregating them by taking the sum. Similarly, attention calculation on edges is mapped to generalized SDDMM; the vanilla SDDMM operation corresponds to a specific attention mechanism that performs a dot product between the source vertex feature vector (i.e., 1D tensor) and the destination vertex feature vector.

By cleanly decomposing a kernel specification into sparse templates and UDFs, FeatGraph enables decoupled, two-level optimizations. At the coarse-grained level, FeatGraph incorporates optimizations for graph traversal into the sparse templates: applying graph partitioning techniques to improve cache utilization on CPU, adapting parallelization strategies for sparse patterns to fully utilize the massive compute capacity on GPU, etc. At the fine-grained level, FeatGraph allows users to specify optimizations for UDFs, e.g., how to tile or parallelize feature dimension computation, with a feature dimension schedule (FDS). FeatGraph combines sparse templates with FDS, and extends a tensor compiler, namely Apache TVM [24], to generate efficient kernels for both CPU and GPU. In addition, by decoupling these two levels of optimizations, FeatGraph significantly improves the productivity of developing new kernels for emerging GNN models.

We perform a comprehensive evaluation to verify the efficiency and flexibility of FeatGraph. Compared with traditional graph processing systems (i.e., Ligra [15] on CPU and Gunrock [16] on GPU), FeatGraph achieves significantly higher performance. Compared with vendor-provided sparse libraries (i.e., MKL [10] on CPU and cuSPARSE [11] on GPU), FeatGraph achieves competitive performance on the kernels that are supported in these libraries while being more flexible

	Platform	Flexibility	Efficiency	Open-source
MKL [10]	CPU	low	high	no
cuSPARSE [11]	GPU	low	high	no
Ligra [15]	CPU	high	low	yes
Gunrock [16]	GPU	high	low	yes
FeatGraph	CPU and GPU	high	high	yes

TABLE I: Side-by-side comparison between FeatGraph and existing works on handling GNN workloads.

to cover more kernels. We integrated FeatGraph into DGL, a popular GNN framework, to accelerate end-to-end GNN training and inference by up to $32\times$ on CPU and $7\times$ on GPU. To the best of our knowledge, FeatGraph is the first unified and generic solution that can flexibly integrate with different GNN frameworks and efficiently process GNN workloads on both CPU and GPU. We summarize the characteristics of FeatGraph and other works in Table I. FeatGraph is available in open-source format at https://github.com/dglai/FeatGraph.

Specifically, this paper makes the following contributions:

- FeatGraph provides a flexible programming interface that is able to express the diverse variants of GNN models by composing coarse-grained sparse templates with customizable fine-grained feature dimension computations on each vertex/edge.
- FeatGraph performs extensive optimizations in both graph traversal and feature dimension computation to generate efficient kernels. In addition, FeatGraph decouples these two levels of optimizations to improve the productivity of developing new kernels for emerging GNN models.
- Experiment results on representative GNN models and a wide collection of datasets show that FeatGraph is portable to existing GNN frameworks and serves as a flexible and efficient backend.

The rest of the paper is organized as follows. Section II reviews the background of GNNs and tensor compilers, and motivates FeatGraph by examining the limitations of existing graph processing systems. Section III describes the programming interface design and optimization techniques of FeatGraph. Section IV presents the system implementation, followed by evaluation in Section V. We discuss related work in Section VI and summarize in Section VII.

II. BACKGROUND AND MOTIVATION

A. Graph Neural Networks (GNNs)

In recent years, there is a rise of interest in adopting deep learning to structural data such as graphs. Unlike the dense objects (e.g., images, videos, texts) handled by traditional deep learning models, graphs represent sparsely, irregularly connected links. Essentially, graphs are defined on a non-Euclidean domain equipped with vastly different distance measurements and geometric properties, imposing the demand for new neural network architectures.

GNNs are an emerging family of neural networks capable of learning a joint representation for each vertex/edge using both features and topological data. Recent studies [3], [25] have unified different GNN models into a *message passing*

paradigm where each vertex computes a new representation by aggregating features (messages) from its neighbors. More formally, given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, we denote the input feature tensor associated with vertex v as $\mathbf{x}_v$, and that associated with the edge pointing from vertex u to v as $\mathbf{x}_{uv}$. To get the representation of a vertex and an edge, the message passing paradigm carries out the following computations:

$$\mathbf{h}_v = \bigoplus_{u \in \mathcal{N}(v)} \phi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{uv}) \qquad (1)$$

$$\mathbf{h}_{uv} = \psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{uv}) \qquad (2)$$

Here ϕ, $\bigoplus$, and ψ are customizable or parameterized functions (e.g., neural network modules) for calculating messages, aggregating messages, and updating edge representations, respectively. Similar to convolutional neural networks (CNNs), a GNN model iteratively applies Equations (1) (2) to generate vertex and edge representations for higher layers.

There is a strong connection between Equations (1) (2) and sparse matrix operations. For example, given the vertex feature matrix $\mathbf{X_V} \in \mathbb{R}^{|\mathcal{V}| \times d}$ and the adjacency matrix $\mathbf{A}$, the vertex-wise computation in the graph convolutional network (GCN) [26], which copies source vertex features as messages and aggregates messages by taking the sum, is equivalent to SpMM (sparse-dense matrix multiplication) as follows.

$$\mathbf{H_V} = \mathbf{A} \times \mathbf{X_V} \qquad (3)$$

For edge-wise computations, many GNN models [27], [28] calculate an attention weight on each edge. One popular formulation for calculating attention weight is by a dot product between the source and destination vertex features [29], that is, $\psi(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{uv}) \triangleq \mathbf{x}_u \mathbf{x}_v^T$. Its tensorized implementation corresponds to SDDMM (sampled dense-dense matrix multiplication) [30], which multiplies two dense matrices, followed by an element-wise multiplication with a sparse mask matrix, to output a sparse matrix.

$$\mathbf{H_E} = \mathbf{A} \cdot (\mathbf{X_V} \times \mathbf{X_V}^T) \qquad (4)$$

Hence, Equations (1) and (2) when implemented as tensor operations are generalized SpMM and SDDMM, respectively. They represent two distinct computation patterns in GNN workloads: reduction on each vertex and reduction on each edge. Moreover, according to the chain rule, the gradient computation of SpMM with respect to $\mathbf{A}$ requires a dot product between the gradients of source and destination vertex features, thus following the SDDMM pattern. Likewise, the gradient computation of SDDMM follows the SpMM pattern. Therefore, these two computation patterns are essential for both inference and training of GNNs. In particular, our benchmarking shows that generalized SpMM and SDDMM occupy $\sim 95\%$ of the total run time in training a 2-layer GNN model, using the existing solutions with sub-optimized sparse kernels.

B. Limitations of Existing Graph Processing Systems

Existing graph processing systems [15], [16] express computations on graphs with a vertex- and/or edge-centric programming paradigm, and they employ a scheduler to realize efficient graph traversal. For example, to ensure load balance on GPU, Gunrock [16] assigns the edges of a vertex to be processed by a thread, a warp, or a block, according to the number of the edges. Edge is the unit for scheduling—the computation on an edge is blackbox to the scheduler. The underlying assumption is that the computation on an edge is lightweight.

However, that assumption breaks in GNNs, which attach a multi-dimensional feature tensor to each vertex/edge, and consequently have more complex vertex-wise and edge-wise computations than traditional graph workloads. For example, MLP aggregation, as shown in Figure 1, performs a sequence of vector matrix multiplication and non-linear activation on each edge. Treating the computation on each edge as a blackbox, Gunrock fails to exploit the abundant parallelism in it.

To enable performant processing of GNN workloads, we need a system that: 1) makes vertex-wise and edge-wise computations whitebox to the scheduler; 2) co-optimizes graph traversal and feature dimension computation. FeatGraph achieves the goals by adopting a tensor compiler approach.

C. Tensor Compiler

Computation-intensive workloads typically operate on tensors, i.e., multi-dimensional data. For example, traditional deep learning models perform matrix multiplication and convolution over dense tensors; GNNs deal with both dense and sparse tensors (the feature tensor is dense and the adjacency matrix is sparse). Previously, people rely on vendor-specific libraries (e.g., MKL, cuBLAS) to obtain high performance of tensor computations over the vendors' own CPUs and GPUs. These libraries require heavy manual tuning by experienced engineers. As a result, they evolve slowly in contrast with the rapid emergence of new workloads.

An alternative solution is tensor compilation, which expresses the processing of tensors in its own intermediate representation (IR) [31], [24]. Tensor compilation separates the computation definition (i.e., what to compute) from the scheduling (i.e., how to compute) so as to focus on the scheduling part for performance optimization. A scheduling scheme can apply loop transformations, vectorization, thread binding, etc., to manipulate the tensor computation. Optimizing one computation kernel for different hardware architectures is essentially searching for different scheduling schemes.

FeatGraph adopts the tensor compilation approach to optimize the computation kernels in GNN workloads. However, existing tensor compilers [24], [31] mostly focus on computations over dense tensors, and there is little support for computations involving sparse tensors. FeatGraph extends TVM [24] to support the core sparse patterns in GNNs (i.e., generalized SpMM and SDDMM), and allows customizable feature dimension computations on each vertex/edge by the design of a two-granularity programming interface (Sec III-B).
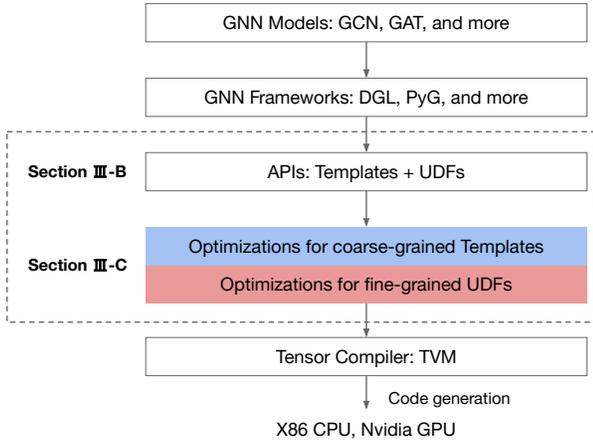
Fig. 2: System overview of FeatGraph.

III. System Design and Optimization

In this section, we first give an overview of the software stack of FeatGraph (Sec III-A). We then describe the design of the programming interface and demonstrate its expressiveness using code examples (Sec III-B). Finally, we cover the optimization techniques for generating efficient GNN kernels on CPU and GPU (Sec III-C).

A. System Overview

Figure 2 depicts the software stack of FeatGraph. At the top level, users define, train, and evaluate GNN models in specialized frameworks such as DGL and PyG, which handle dataflow programming and automatic differentiation. FeatGraph serves as a backend for these frameworks, targeting the message passing computation that is core to GNN workloads. FeatGraph provides a flexible programming interface to express the diverse variants allowed by the message passing paradigm. Specifically, FeatGraph describes feature dimension computations on each vertex/edge with user-defined functions (UDFs), and triggers UDFs by SpMM or SDDMM sparse template. FeatGraph incorporates optimizations for graph traversal into the sparse templates, and allows users to specify optimizations for UDFs with a feature dimension schedule (FDS). FeatGraph combines templates with FDS, and leverages the TVM tensor compiler [24] to generate efficient kernels for both CPU and GPU. By decoupling these two levels of optimizations, FeatGraph significantly improves the productivity of developing new kernels for emerging GNN models.

B. Programming Interface

There are two principles in the design of FeatGraph's programming interface. First, the interface should closely follow the mathematical definition of GNNs as described in Section II-A. Second, it should facilitate optimizations.

To these ends, we propose to decompose a kernel specification into two parts: UDFs written in a tensor expression language adopted from TVM to describe fine-grained feature dimension computations on each vertex/edge, and the choice of

```
1  import featgraph, tvm
2  A = featgraph.spmat(shape=(n,n), nnz=m)
3
4  # use src vertex feature as message
5  XV = tvm.placeholder(shape=(n,d))
6  def msgfunc(src, dst, eid):
7    out = tvm.compute((d,), lambda i: XV[src,i])
8    return out
9
10 # tile feature dimension for cache optimization
11 def cpu_schedule(out):
12   s = tvm.create_schedule(out)
13   # the tiling factor is tunable
14   s[out].split(out.axis[0], factor=8)
15   return s
16
17 # parallelize feature dimension
18 # by binding it to the thread index in CUDA
19 def gpu_schedule(out):
20   s = tvm.create_schedule(out)
21   s[out].bind(out.axis[0], 'thread.x')
22   return s
23
24 # use sum as the aggregation function
25 aggregation = tvm.sum
26
27 # trigger the SpMM template
28 if target = 'cpu':
29   fds = cpu_schedule
30 elif target == 'gpu':
31   fds = gpu_schedule
32 GCN = featgraph.spmm(A, msgfunc, aggregation,
33                      target, fds)
```

(a) GCN aggregation

```
1  # ReLU((src feature + dst feature) * W)
2  XV = tvm.placeholder(shape=(n,d1))
3  W = tvm.placeholder(shape=(d1,d2))
4  def msgfunc(src, dst, eid):
5    k = tvm.reduce_axis((0,d1))
6    out = tvm.compute((d2,), lambda i:
7      tvm.max(tvm.sum((XV[src,k] + XV[dst,k])
8                      * W[k,i])), 0)
9    return out
```

(b) Message function of MLP aggregation

Fig. 3: Example code of vertex-wise computations with the SpMM template — FeatGraph inlines the fine-grained FDS (in red) into the coarse-grained SpMM template (in blue) to generate a fused, optimized kernel.

coarse-grained sparse patterns. FeatGraph provides two kernel templates `featgraph.spmm` and `featgraph.sddmm` for the SpMM and SDDMM sparse patterns that directly map to the vertex-wise and edge-wise computations in the message passing paradigm, i.e., Equations (1) and (2).

More concretely, `featgraph.spmm` takes in five arguments: an adjacency matrix, a message function, an aggregation function, the target (CPU or GPU), and an FDS to specify optimizations of the message function. Figure 3a shows the code for GCN aggregation, i.e., the message aggregation in GCN model as described in Section II-A. Given the edge ID tuple `(src, dst, eid)`, the user-defined message function `msgfunc` (line 6–8) slices out the `src` row from the vertex feature matrix `XV`, which is equivalent to using the source vertex feature as the message. The aggregation function is `sum`

```
1  import featgraph, tvm
2  A = featgraph.spmat(shape=(n,n), nnz=m)
3
4  # dot product between src and dst vertex features
5  XV = tvm.placeholder(shape=(n,d))
6  def edgefunc(src, dst, eid):
7    k = tvm.reduce_axis((0,d))
8    out = tvm.compute(shape=(1,), lambda i:
9      tvm.sum(XV[src,k] * XV[dst,k]))
10   return out
11
12 # tree-based parallel reduction
13 def gpu_schedule(out):
14   s = tvm.create_schedule(out)
15   s[out].tree_reduce(out.reduce_axis[0], 'thread.x')
16   return s
17
18 # trigger the SDDMM template
19 target = 'gpu'
20 fds = gpu_schedule
21 Attention = featgraph.sddmm(A, edgefunc, target, fds)
```
(a) Dot-product attention

```
1  # multiple dot products
2  XV = tvm.placeholder(shape=(n,h,d))
3  def edgefunc(src, dst, eid):
4    k = tvm.reduce_axis((0,d))
5    out = tvm.compute(shape=(h,), lambda i:
6      tvm.sum(XV[src,i,k] * XV[dst,i,k]))
7    return out
```
(b) Edge function of multi-head dot-product attention

Fig. 4: Example code of edge-wise computations with the SDDMM template — FeatGraph inlines the fine-grained FDS (in red) into the coarse-grained SDDMM template (in blue) to generate a fused, optimized kernel.

and any commutative reducer is allowed. Figure 3b shows a more complex message function, which adds the source and destination vertex features, and then multiplies with a weight matrix, followed by a ReLU activation (i.e., taking the max with 0).

FeatGraph can easily support the commonly used message functions in GNNs—specifically, all the builtin ones provided by DGL[1], including copying vertex or edge feature tensor, element-wise operations between vertex and edge feature tensors, etc. In addition, FeatGraph can express more complex message functions such as the one in MLP aggregation.

FeatGraph inlines the message function into the SpMM template to generate a fused kernel. In contrast, existing GNN frameworks (e.g., DGL, PyG, NeuGraph) that rely on deep learning systems as backend have to materialize the messages on every edge, causing inefficiency in both performance and memory consumption.

`featgraph.sddmm` takes in four arguments: an adjacency matrix, an edge function, the target (CPU or GPU), and an FDS to specify optimizations of the edge function. Figure 4a shows the code for dot-product attention, where the user-defined edge function `edgefunc` (line 6–10) performs a dot product between the source and destination vertex feature vectors, and returns an attention weight as the new feature on the edge.

[1] https://docs.dgl.ai/api/python/function.html#message-functions

Figure 4b shows a more complex edge function, which performs multiple dot products over the feature tensors.

This two-granularity programming interface simplifies implementing new GNN kernels and, more important, facilitates optimizations. By cleanly decomposing a kernel specification into coarse-grained sparse templates and fine-grained feature dimension computations on each vertex/edge in the form of UDFs, FeatGraph enables decoupled, two-level optimizations. Specifically, FeatGraph incorporates optimizations for graph traversal into the sparse templates and allows users to specify optimizations for UDFs with an FDS. Some FDS examples, both for CPU and for GPU, are shown in Figure 3a at line 11–15 and line 19–22. It is worth noting that when the FDS is missing, FeatGraph essentially degrades to traditional graph processing systems that are designed without special handling of feature dimension computation.

C. Decoupled, Two-level Optimizations

This subsection describes the optimizations for graph traversal, which are incorporated into the sparse templates, and the optimizations for feature dimension computation, which are specified by users with an FDS. We analyze the interplay between these two levels of optimizations, and show that by combining them, FeatGraph enables performant processing of GNN workloads. Throughout this subsection, we use the sample graph shown in Figure 5 to illustrate optimization techniques.

1) Graph Partitioning and Feature Dimension Tiling: On CPU, the key factor limiting the efficiency of graph traversal is poor locality, which causes low cache utilization. Prior arts have attempted to improve locality in graph traversal by graph partitioning [18], [19]. FeatGraph proposes combining graph partitioning with feature dimension tiling to strike a balance between efficiency of graph traversal and efficiency of feature dimension computation.

Figure 6 illustrates how feature dimension tiling is combined with 1D graph partitioning [18], which partitions source vertices, to effectively optimize cache utilization in GCN aggregation, i.e., the vanilla SpMM operation. Here we assume the feature vector length is four, and the cache can hold two feature vectors. With 1D graph partitioning alone, source vertices are partitioned into four segments so that each segment fits into the cache; these segments are processed one by one to get four portions of intermediate results; in the end the
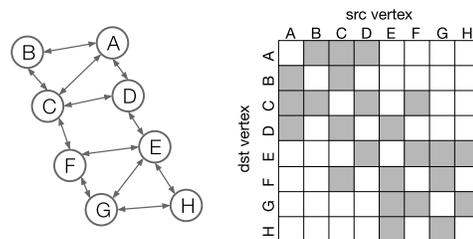

Fig. 5: A sample graph with 8 vertices and its corresponding adjacency matrix.

(a) 1D graph partitioning

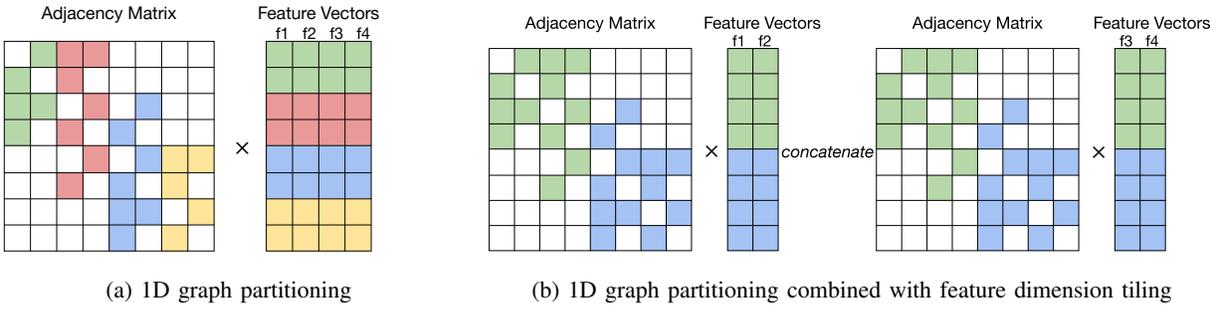(b) 1D graph partitioning combined with feature dimension tiling

Fig. 6: Feature dimension tiling and 1D graph partitioning for cache optimization in GCN aggregation — We assume the cache can hold two feature vectors. Tiling each feature vector into two sub-vectors reduces the number of graph partitions from four to two, which translates to 50% saving in merge, but at the cost of accessing the adjacency matrix twice.

(a) GCN aggregation
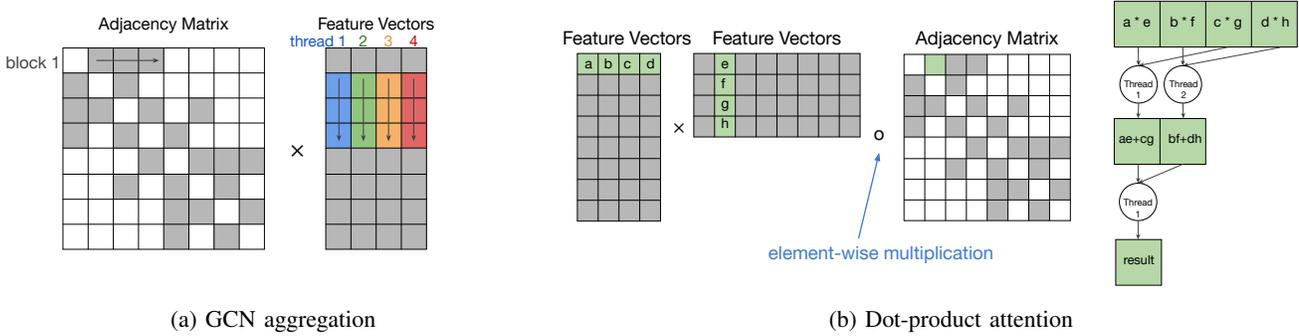
(b) Dot-product attention

Fig. 7: Parallelization strategies adapted for computation patterns.

intermediate results are merged. 1D graph partitioning improves read locality within each segment at the cost of merging intermediate results from different segments. When 1D graph partitioning is combined with feature dimension tiling, merge cost is reduced since more vertices can fit into the cache under the same capacity. As shown in Figure 6b, tiling each feature vector into two sub-vectors reduces the number of segments from four to two, which translates to 50% saving in merge cost. However, feature dimension tiling results in traversing the graph twice, which means increased accesses to graph topological data (i.e., the adjacency matrix).

Thus, feature dimension tiling introduces the trade-off between accesses to graph topological data and accesses to feature data. In GNNs, feature vectors have a typical length ranging from 32 to 1024. When the tiling factor is properly selected, the gain of improved locality in accessing feature data far outweighs the overhead of increased accesses to graph topological data.

More complex UDFs that compute on multi-dimensional feature tensors may require a multi-level tiling scheme. To efficiently support diverse UDFs, FeatGraph allows users to specify optimizations for UDFs with an FDS. Figure 8 shows the FDS for MLP aggregation—it tiles both dimensions of the weight matrix for cache optimization.

For edge-wise computations, besides feature dimension tiling, FeatGraph employs a graph traversal scheme [32] based on Hilbert curve. Edge-wise computations access both source and destination vertex features, and update edge features; Hilbert

```
1  # tile multiple dimensions
2  def cpu_schedule(out):
3      s = tvm.create_schedule(out)
4      s[out].split(out.axis[0], factor=8)
5      s[out].split(out.reduce_axis[0], factor=8)
6      return s
```

Fig. 8: FDS for MLP aggregation on CPU.

```
1  # parallelize multiple dimensions
2  def gpu_schedule(out):
3      s = tvm.create_schedule(out)
4      s[out].bind(out.axis[0], 'block.x')
5      s[out].tree_reduce(out.reduce_axis[0], 'thread.x')
6      return s
```

Fig. 9: FDS for MLP aggregation on GPU.

curve traversal exploits locality in accessing both source and destination vertices. The recursive structure of Hilbert curve enables exploiting locality across a spectrum of granularities, e.g., L1/L2/L3 caches. FeatGraph combines Hilbert curve traversal with feature dimension tiling to fully optimize edge-wise computations.

2) Adaptive Parallelization Strategies: To utilize GPU's massive parallel compute capacity, prior graph processing systems exploit parallelism in graph traversal by implementing either vertex parallelization or edge parallelization [16], [21], [20]. However, they are unable to exploit the abundant parallelism in feature dimension computation arising in GNN workloads due to treating the UDFs as a blackbox. FeatGraph enables exploiting parallelism in feature dimension computation

by opening the blackbox of UDFs so as to inform the scheduler. Specifically, FeatGraph allows users to specify a parallelization scheme for UDFs with an FDS, which can be adapted to the diverse computation patterns of UDFs to fully exploit the parallelism in feature dimension computation.

For vertex-wise computations, FeatGraph incorporates vertex parallelization into the SpMM template and allows users to specify a parallelization scheme for the message function with an FDS. For example, the FDS for GCN aggregation is shown in Figure 3a at line 19–22, which, combined with the SpMM template, defines the parallelization strategy shown in Figure 7a: each CUDA block processes a number of vertices, which correspond to several rows in the adjacency matrix, and the feature dimension is parallelized across the threads in one CUDA block. This simple parallelization strategy turns out to be highly efficient—there is no load imbalance within each CUDA block since all threads are assigned exactly the same amount of work; no control divergence; read requests into global memory from the threads within one CUDA block are contiguous and can be coalesced to realize high bandwidth utilization. This parallelization strategy is first proposed in [33] that focuses on manually optimizing the vanilla SpMM kernel; we can easily express it with the programming infrastructure of FeatGraph to optimize a broad class of generalized SpMM computations.

For edge-wise computations, FeatGraph incorporates edge parallelization into the SDDMM template and allows users to specify a parallelization scheme for the edge function with an FDS. For example, the FDS for dot-product attention is shown in Figure 4a at line 13–16, which, combined with the SDDMM template, defines the parallelization strategy shown in Figure 7b: each CUDA block processes a number of edges, which correspond to several non-zero elements in the adjacency matrix, and all the threads in one CUDA block collectively process the dot-product operations on edges using tree reduction [34]. Prior graph processing systems (e.g., Gunrock [16]), which are designed without being aware of feature dimension computation, fail to exploit this form of parallelism.

More complex UDFs that compute on multi-dimensional feature tensors require a multi-level parallelization scheme. Figure 9 shows the FDS for MLP aggregation—it parallelizes the first dimension across CUDA blocks and the second dimension across threads.

3) Hybrid Partitioning on GPU: The optimizations for graph traversal on CPU (e.g., 1D graph partitioning) are not directly applicable to GPU due to the differences between CPU and GPU memory architectures—shared memory size (configurable up to 96 KB on Tesla V100 GPU) is much smaller than LLC, which is typically tens of Mega Bytes (MBs). To make effective use of limited-capacity shared memory on GPU, we propose a hybrid partitioning method that processes high-degree vertices and low-degree vertices differently. Specifically, this method reorders the vertices into a low-degree part and a high-degree part according to a threshold; it only partitions high-degree vertices and loads them to shared memory. The intuition of hybrid partitioning is that high-degree vertices are accessed

for more times and therefore can benefit more from shared memory optimization. The key trade-off here is between read efficiency and merge cost—a smaller degree threshold leads to more partitions, which improves read efficiency but increases merge cost.

IV. System Implementation

This section describes the implementation of FeatGraph, in particular, how we extended TVM to support the core sparse patterns of GNNs (i.e., generalized SpMM and SDDMM), and how we integrated FeatGraph into DGL.

A. TVM IR Templates

We implemented the SpMM and SDDMM templates as TVM IR templates. TVM is a domain-specific language and compiler for tensor computations and has been widely adopted to accelerate deep learning workloads [35], [36]. Because TVM does not support sparse representation and computation in its tensor expression language, we implemented and optimized SpMM and SDDMM templates by directly constructing and manipulating the IR (intermediate representation) using lower-level APIs. Feature dimension computations on each vertex/edge described by UDFs are dense and therefore easily supported. FeatGraph combines scheduling parameters from the sparse templates (e.g., number of graph partitions, number of CUDA blocks) and those from the FDS (e.g., feature dimension tiling factors) to create the design space. In this work we use naïve grid search to find the optimal parameters under a given input shape, and it is an interesting future direction to try more intelligent tuners [37], [38] for faster design space exploration. After performing optimizations for both the templates and UDFs, FeatGraph inlines UDFs into the templates to generate fused kernels.

We parallelize the kernels over multiple threads on CPU using the customized thread pool [35] in TVM runtime, which is lightweight and particularly efficient in handling the kind of embarrassingly parallel workloads. To avoid LLC contention after graph partitioning, we assign multiple threads to collectively work on one graph partition at a time instead of assigning each thread to a different partition.

B. DGL Integration

In order to evaluate the performance of FeatGraph in end-to-end GNN training and inference, we integrated FeatGraph into DGL, a popular open-source GNN framework. DGL implemented a minimal Gunrock-like graph kernel interface named Minigun [39]. With Minigun, DGL provided a set of builtin message functions and edge functions to support common GNN workloads. For each of these builtin functions, we implemented a corresponding one with the programming infrastructure of FeatGraph, such as GCN aggregation and dot-product attention. To handle more complex cases such as MLP aggregation, the current solution in DGL is to calculate and materialize the messages on every edge using deep learning systems as backend. In contrast, FeatGraph generates fused kernels, thus both saving memory and improving efficiency. FeatGraph generates kernel

codes for a specific graph topology (i.e., the adjacency matrix); since GNN training typically involves hundreds of epochs, the compilation cost is amortized and negligible.

The integration requires a small amount of effort (only ~ 300 lines of Python code) because both FeatGraph and DGL follow the message passing paradigm in their programming interface design. The integration with DGL demonstrates that it is straightforward to have FeatGraph be the backend to accelerate GNN frameworks in general, including PyG, NeuGraph, etc.

V. EVALUATION

This section seeks to answer the following questions:
1) What is the performance gain of GNN kernels on both CPU and GPU?
2) What is the implication of each of our proposed optimization techniques for both templates and UDFs?
3) Is the kernel performance sensitive to scheduling parameters and graph sparsity?
4) What is the speedup of end-to-end GNN training and inference brought by FeatGraph without affecting the accuracy of the models?

A. Experiment Setup

Environment. For CPU evaluation, we conduct experiments on Amazon EC2 c5.9xlarge instance, which is a one-socket 18-core 3.0 GHz Intel Xeon Platinum 8124M machine with 25 MB LLC and 68 GB DRAM. For GPU evaluation, we conduct experiments on p3.2xlarge instance, which has a Tesla V100 GPU with 80 SMs; each SM has shared memory configurable up to 96 KB (the default size is 48 KB).

Datasets. Table II lists the datasets used for evaluation: `ogbn-proteins` represents proteins and their biological associations with vertices and edges—this dataset is from Open Graph Benchmark[2], a realistic benchmark suite for GNNs; `reddit` [40] is constructed from the Reddit online forum wherein vertices represent posts and edges are established if two posts are commented by a same use—this dataset is commonly used in GNN research for evaluating the accuracy of new models; `rand-100K` is a synthetic graph wherein 20K vertices have an average degree of 2000 and the remaining 80K vertices have an average degree of 100—this dataset is specifically aimed at studying the effect of hybrid partitioning on GPU performance.

Baselines. We compare FeatGraph with state-of-the-art graph processing systems, specifically *Ligra* on CPU and *Gunrock* on GPU. We also compare with vendor-provided sparse libraries, specifically *MKL* (2019.5) on CPU and *cuSPARSE* (10.1) on GPU whenever possible, as only a subset of GNN kernels are supported in these libraries. In all the experiments, we first do a warm-up run and then take the average time of 10 runs as the measurement.

B. Performance Gain of GNN Kernels

We evaluate the performance gain of FeatGraph on three kernels: GCN aggregation, MLP aggregation, and dot-product

[2]https://ogb.stanford.edu/

Graph dataset	$|\mathcal{V}|$	$|\mathcal{E}|$	Average degree
ogbn-proteins	132.5K	79.1M	597
reddit	233.0K	114.8M	493
rand-100K	100.0K	48.0M	480

TABLE II: Graph datasets (K: thousand, M: million).

Unit: sec		Feature length				
		32	64	128	256	512
ogbn-proteins	Ligra	1.47	2.05	3.10	6.01	12.30
	MKL	0.60	**0.96**	2.17	5.34	14.71
	FeatGraph	**0.50**	0.99	**1.97**	**3.94**	**8.02**
reddit	Ligra	4.10	7.20	13.10	20.40	34.90
	MKL	1.50	3.01	7.87	17.79	40.06
	FeatGraph	**1.02**	**2.13**	**4.09**	**8.16**	**16.71**
rand-100K	Ligra	0.64	0.86	1.49	2.58	4.91
	MKL	0.43	0.77	2.26	5.45	15.51
	FeatGraph	**0.22**	**0.43**	**0.87**	**1.74**	**3.52**

(a) GCN aggregation

Unit: sec		Feature length				
		32	64	128	256	512
ogbn-proteins	Ligra	12.90	24.70	47.70	94.00	187.00
	FeatGraph	**2.48**	**4.84**	**9.68**	**19.55**	**38.70**
reddit	Ligra	20.70	37.90	71.50	139.00	273.00
	FeatGraph	**4.03**	**8.20**	**15.33**	**30.80**	**62.07**
rand-100K	Ligra	7.81	14.80	28.80	56.90	113.00
	FeatGraph	**1.42**	**2.74**	**5.48**	**10.96**	**21.97**

(b) MLP aggregation

Unit: sec		Feature length				
		32	64	128	256	512
ogbn-proteins	Ligra	9.81	22.30	47.50	97.70	198.00
	FeatGraph	**2.21**	**4.39**	**8.67**	**16.46**	**32.97**
reddit	Ligra	17.20	37.30	77.20	152.00	297.00
	FeatGraph	**3.71**	**7.34**	**14.11**	**27.13**	**54.51**
rand-100K	Ligra	5.57	12.90	28.20	58.30	119.00
	FeatGraph	**1.28**	**2.51**	**5.37**	**10.76**	**21.47**

(c) Dot-product attention

TABLE III: Single-threaded CPU performance. Best result is marked in **bold**.

attention. The kernels are performed on the full graph. We do the evaluation across a spectrum of feature lengths. For MLP aggregation, the feature length refers to `d2` that is shown in Figure 3b; `d1` is fixed as 8.

Single-threaded CPU Performance. Table III shows that across all the evaluated datasets under different feature lengths, FeatGraph achieves $1.4\times$–$4.0\times$ speedup over Ligra on GCN aggregation, $4.4\times$–$5.5\times$ speedup on MLP aggregation, and $4.3\times$–$6.0\times$ speedup on dot-product attention, using a single thread. Compared against MKL on GCN aggregation, FeatGraph is faster in 14 out of 15 cases and achieves higher speedup with a larger feature length. Specifically, when the feature length is 512, FeatGraph is $1.8\times$ faster on `ogbn-proteins`, $2.4\times$ faster on `reddit`, and $4.4\times$ faster on `rand-100K`. MKL does not support MLP aggregation and dot-product attention.

Multi-threaded CPU Performance. Figure 10 shows that with 16 threads, for GCN aggregation on `reddit`, FeatGraph achieves $12.6\times$ speedup over its single-threaded execution, which is slightly higher than Ligra ($9.5\times$) and MKL
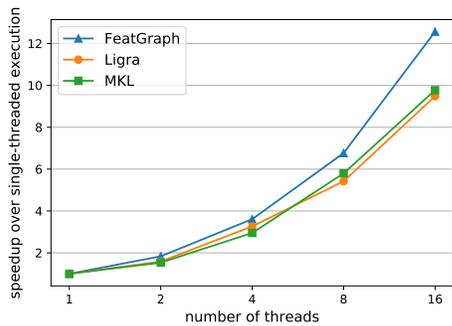
Fig. 10: Scalability comparison of FeatGraph with Ligra and MKL. Evaluated on GCN aggregation. Tested on `reddit` with feature length 512.

Unit: ms		Feature length				
		32	64	128	256	512
ogbn-proteins	Gunrock	114.2	276.7	1322.3	4640.3	12423.9
	cuSPARSE	**4.1**	8.1	16.2	32.1	64.2
	FeatGraph	4.6	**7.8**	**15.4**	**30.8**	**61.9**
reddit	Gunrock	616.9	2026.4	5141.2	11715.3	24749.8
	cuSPARSE	**12.2**	**25.1**	**51.6**	**104.7**	**209.6**
	FeatGraph	14.3	28.6	57.8	116.9	232.0
rand-100K	Gunrock	72.7	175.5	1006.2	3303.7	8236.5
	cuSPARSE	3.6	5.9	10.6	21.9	44.4
	FeatGraph	**2.8**	**4.9**	**10.2**	**20.3**	**39.9**

(a) GCN aggregation

Unit: ms		Feature length				
		32	64	128	256	512
ogbn-proteins	Gunrock	591.6	833.4	2067.7	5603.5	13687.4
	FeatGraph	**26.9**	**46.7**	**87.4**	**168.9**	**332.9**
reddit	Gunrock	1285.6	2697.5	5886.4	12285.0	25442.3
	FeatGraph	**33.2**	**76.7**	**142.9**	**277.1**	**547.9**
rand-100K	Gunrock	447.2	648.1	1556.1	3848.5	8624.6
	FeatGraph	**8.9**	**14.9**	**26.0**	**46.6**	**89.6**

(b) MLP aggregation

Unit: ms		Feature length				
		32	64	128	256	512
ogbn-proteins	Gunrock	30.9	58.8	120.2	251.3	645.1
	FeatGraph	**24.4**	**37.9**	**69.3**	**143.3**	**333.7**
reddit	Gunrock	44.8	99.3	278.5	648.2	1388.7
	FeatGraph	**35.9**	**56.6**	**103.7**	**212.0**	**483.2**
rand-100K	Gunrock	19.3	37.3	75.5	174.3	441.6
	FeatGraph	**14.9**	**23.2**	**42.3**	**87.8**	**201.5**

(c) Dot-product attention

TABLE IV: GPU performance. Best result is marked in **bold**.

$(9.8\times)$. Similar observation applies to other datasets and kernels. As a result, FeatGraph outperforms the others consistently in multi-threaded environment. FeatGraph scales well due to two factors: 1) its parallelization method avoids LLC contention by assigning multiple threads to collectively work on one graph partition at a time; 2) the thread pool in TVM runtime is lightweight [35].

GPU Performance. Table IV shows that FeatGraph is $24\times$–$206\times$ faster than Gunrock on GCN aggregation, $18\times$–$96\times$ faster on MLP aggregation, and $1.2\times$–$3.1\times$ faster on dot-product attention. The extreme slowness of Gunrock on GCN aggregation and MLP aggregation is caused by two reasons: 1)

Gunrock's edge parallelization execution incurs huge overhead of atomic operations for vertex-wise reductions such as GCN aggregation and MLP aggregation; 2) Gunrock fails to exploit parallelism in feature dimension computation. FeatGraph is on par with cuSPARSE on GCN aggregation, being 10%–20% faster on `ogbn-proteins` and `rand-100K` while 10% slower on `reddit`. Notably, cuSPARSE does not support MLP aggregation and dot-product attention.[3]

C. Optimization Implications

This subsection investigates the performance boost of each individual optimization technique described in Section III. For the sake of space, in each ablation analysis we only pick one dataset to show the optimization effects. Other datasets share similar observations.

Graph Partitioning and Feature Dimension Tiling. Figure 11 shows that feature dimension tiling combined with graph partitioning effectively boosts the performance of GCN aggregation on CPU. Specifically, when the feature length is 512, feature dimension tiling alone and graph partitioning alone bring $1.2\times$ speedup and $1.7\times$ speedup, respectively; combining two achieves $2.2\times$ speedup.

Adaptive Parallelization Strategies on GPU. Figure 12 shows that tree reduction boosts the performance of dot-product attention by up to $2\times$. The naïve parallelization strategy in Gunrock that assigns the entire dot product operation on each edge to one CUDA thread is less efficient in handling large feature lengths due to consuming too many registers per thread.

Hybrid Partitioning on GPU. Figure 13 shows the effect of hybrid partitioning on GCN aggregation tested on `rand-100`. FeatGraph gets 10%–20% performance boost by hybrid partitioning, and consequently outperforms cuSPARSE.

D. Sensitivity Analysis

Sensitivity to Partitioning Factors. Figure 14 shows that the performance of FeatGraph is sensitive to partitioning factors for GCN aggregation on CPU. Specifically, on `reddit`, when the feature length is 128, the best performance is achieved with 16 graph partitions and 4 feature partitions. On the same graph, as the feature length increases, the optimal number of feature partitions increases proportionately, while the optimal number of graph partitions stays constant. Transferable tuning across graphs, i.e., using the optimal partitioning factors tuned on one graph to predict the optimal partitioning factors for a new graph, is more challenging and worth further study.

Sensitivity to GPU Parameters. Figure 15 shows that FeatGraph performs better with a larger number of CUDA blocks for GCN aggregation on GPU, because a larger number of CUDA blocks can better utilize the massive parallel compute capacity of GPU. In the evaluation, we set the number of CUDA blocks to the number of rows of the adjacency matrix.

Sensitivity to Graph Sparsity. Table V shows that Feat-Graph achieves higher speedup over MKL as the graph sparsity decreases for GCN aggregation on CPU. This trend is because

---

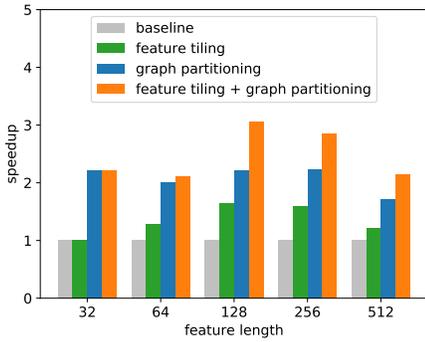[3]The latest cuSPARSE supports dot-product attention via ConstrainedGeMM.

Fig. 11: Effect of graph partitioning and feature tiling on the CPU performance of GCN aggregation. Tested on `reddit`.
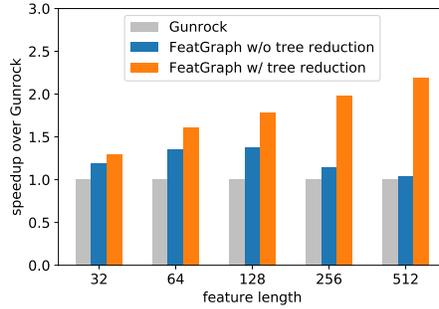

Fig. 12: Effect of tree reduction on the GPU performance of dot-product attention. Tested on `rand-100K`.
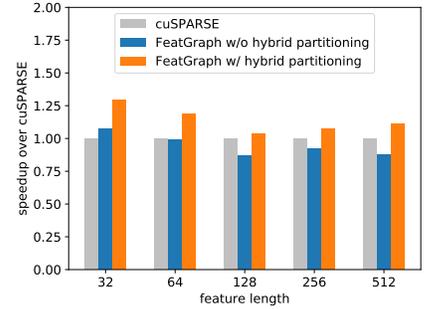

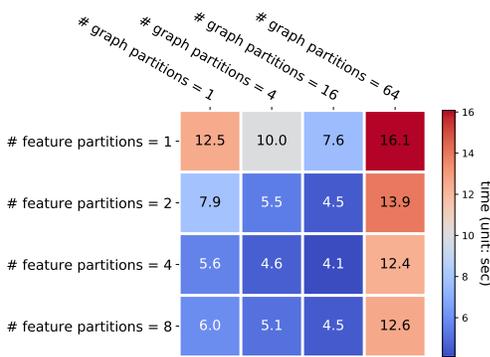Fig. 13: Effect of hybrid partitioning on the GPU performance of GCN aggregation. Tested on `rand-100K`.

Fig. 14: Sensitivity of FeatGraph performance to partitioning factors for GCN aggregation on CPU. The dataset is `reddit`. The feature length is 128.

Graph sparsity	MKL (unit: sec)	FeatGraph (unit: sec)	Speedup
99.95%	0.34	0.31	1.10×
99.5%	3.58	1.95	1.84×
95%	37.22	12.78	2.91×

TABLE V: Sensitivity of FeatGraph performance to graph sparsity for GCN aggregation on CPU. The dataset is a synthetic uniform graph with 100K vertices. The feature length is 128.

		DGL w/o FeatGraph (unit: sec)	DGL w/ FeatGraph (unit: sec)	Speedup
CPU training	GCN	2447.1	114.5	21.4×
	GraphSage	1269.6	57.8	21.9×
	GAT	5763.9	179.3	32.2×
CPU inference	GCN	1176.9	55.3	21.3×
	GraphSage	602.4	29.8	20.2×
	GAT	1580.9	71.5	22.1×
GPU training	GCN	6.3	2.2	2.9×
	GraphSage	3.1	1.5	2.1×
	GAT	*N/A	1.64	*N/A
GPU inference	GCN	3.1	1.5	2.1×
	GraphSage	1.5	1.1	1.4×
	GAT	8.1	1.1	7.1×

TABLE VI: Speedup of end-to-end GNN training and inference brought by FeatGraph. Tested on `reddit`. Time is for one epoch. (*GAT training in DGL w/o FeatGraph runs out of GPU memory.)
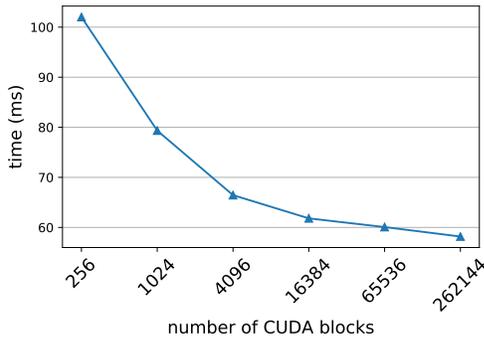

Fig. 15: Sensitivity of FeatGraph performance to the number of CUDA blocks for GCN aggregation on GPU. The dataset is `reddit`. The feature length is 128.

a denser graph has more data reuse, which FeatGraph is able to exploit by graph partitioning and feature dimension tiling.

E. End-To-End GNN Training and Inference

We integrated FeatGraph into DGL and evaluated the performance of FeatGraph in end-to-end GNN training and inference on three models: a 2-layer graph convolutional network (GCN) [26] of hidden size 512, a 2-layer GraphSage [40] of hidden size 256, and a 2-layer graph attention network (GAT) [27] of hidden size 256. GCN uses sum aggregation and requires generalized SpMM computations in both forward and backward propagation; GraphSage follows a similar architecture as GCN but allows more flexible aggregation functions (e.g., max); GAT uses dot-product attention, thus requiring both generalized SpMM and SDDMM computations.

Accuracy. FeatGraph as a backend is for performance optimization without changing the semantics of GNN models. As a sanity check, we evaluate the accuracy of the three models on the task of vertex classification. The 233K vertices of the `reddit` dataset are split into 153K, 24K, and 56K for training, validation, and testing, respectively. We train the models for 200

epochs. The testing accuracy obtained by DGL using FeatGraph matches that obtained by DGL using its original backend Minigun—93.7% for GCN and 93.1% for GraphSage. The training of GAT does not converge due to gradient explosion, with either FeatGraph backend or Minigun backend.

Speedup. Table VI reports the training and inference time of one epoch for the three GNN models. The time of tuning partitioning factors is excluded, because it is amortized over multiple epochs—it is less than 1% of the time of training GCN for 200 epochs. Furthermore, the partitioning factors tuned on GCN are directly applied to GraphSage and GAT— the number of graph partitions is kept the same and the number of feature partitions is adjusted to the feature length. The results show that on CPU, FeatGraph speeds up both training and inference by more than $20\times$ on all the three models; on GPU, FeatGraph speeds up training by more than $2\times$, and inference by $1.4\times$–$7.1\times$. The highest speedup is achieved on GAT, which has a more complex architecture than GCN and GraphSage.

VI. Related Work

Recent years have seen an emergence of specialized frameworks that attempt to make the processing of GNN workloads easier and faster. For example, DGL [9] and PyG [7] wrap deep learning systems with a message-passing programming interface for GNNs. NeuGraph [5] addresses the challenge of large-scale GNN training by partitioning the dataflow over multiple GPUs and employing a chain-based streaming schedule. FeatGraph focuses on optimizing graph-specific kernels, and can be integrated into these GNN frameworks to serve as an efficient backend on both CPU and GPU.

Systems for processing traditional graph workloads (e.g., BFS, PageRank) have been extensively studied in literature [12], [13], [14], [15], [16], [17]. These systems allow users to flexibly express graph algorithms by defining computation on each vertex/edge. Among them, Ligra [15] achieves superior performance on CPU by dynamically switching the message propagation direction (i.e., push or pull) based on the size of the frontier (active vertices) at each iteration, and Gunrock [16] achieves superior GPU performance by sophisticated scheduling methods to ensure load balance in its edge parallelization execution. However, Ligra is not exploiting cache optimization, and its push-pull optimization is no longer critical in GNN workloads since typically all vertices are active at each layer of a GNN model. Gunrock fails to achieve good performance for GNN workloads because it is unable to exploit parallelism in feature dimension computation, let alone adapt parallelization strategies for computation patterns.

There is another series of works that focus on formulating graph algorithms as sparse linear algebra operations [41], [42], [43], [44]. For example, BFS is formulated as a sequence of sparse matrix sparse vector multiplication (SpMSpV); PageRank is formulated as a sequence of sparse matrix dense vector multiplication (SpMV). FeatGraph borrows from these works the general idea of mapping graph computations to sparse kernels. FeatGraph differs from these works in two major aspects: 1) FeatGraph can express more complex user-defined

functions (UDFs) to support the diverse variants of GNN models; 2) FeatGraph pays a special attention to optimizations of feature dimension computation, which are unexploited in previous efforts.

Vendor-specific libraries (e.g., MKL [10] and cuSPARSE [11]) provide highly optimized implementations for sparse kernels that are identified important to a broad range of applications. Compared with these libraries, FeatGraph is more comprehensive at kernel coverage for GNN's use case. Besides, by adopting a tensor compiler approach in contrast to the manual optimization approach of these libraries, FeatGraph is able to search for the best scheduling schemes on both CPU and GPU.

TACO [45] is a compiler targeting general sparse tensor computations by the design of a flexible sparse tensor representation system. However, TACO does not allow scheduling as TVM, and it lacks support for generating high-quality GPU code. Instead of targeting general sparse computations, FeatGraph targets the core sparse patterns of GNNs, namely, generalized SpMM for vertex-wise computations and generalized SDDMM for edge-wise computations. This design choice enables FeatGraph to fully exploit the optimization opportunities specific to GNN workloads.

VII. Conclusion

We propose FeatGraph to enable performant processing of graph neural network (GNN) workloads. FeatGraph provides a flexible programming interface that is able to express the diverse variants of GNN workloads by composing sparse templates with customizable feature dimension computations on each vertex/edge. FeatGraph extensively explores optimization opportunities in both graph traversal and feature dimension computation. Moreover, it decouples these two levels of optimizations to improve the productivity of developing new kernels for emerging GNN models. FeatGraph is portable to existing GNN frameworks as a high-performance backend. Our evaluation verifies that FeatGraph is comprehensive at kernel coverage and outperforms the state-of-the-art solutions. Future work remains to utilize more intelligent tuners [38] to further improve the performance, and to integrate FeatGraph into large-scale GNN training systems such as NeuGraph to accelerate multi-GPU training.

REFERENCES

[1] Q. Tan, N. Liu, and X. Hu, "Deep representation learning for social network analysis," *arXiv preprint arXiv:1904.08547*, 2019.

[2] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," *Int'l Conf. on Knowledge Discovery and Data Mining (KDD)*, 2018.

[3] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *Int'l Conf. on Machine Learning (ICML)*, 2017.

[4] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," *Conf. on Neural Information Processing Systems (NIPS)*, 2018.

[5] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: Parallel deep neural network computation on large graphs," *USENIX Annual Technical Conf. (ATC)*, 2019.

[6] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2016.

[7] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Conf. on Neural Information Processing Systems (NeurIPS)*, 2019.

[9] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma *et al.*, "Deep graph library: Towards efficient and scalable deep learning on graphs," *arXiv preprint arXiv:1909.01315*, 2019.

[10] Intel, "Intel math kernel library," https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html.

[11] Nvidia, "Cusparse library," https://developer.nvidia.com/cusparse.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," *Int'l Conf. on Management of Data (SIGMOD)*, 2010.

[13] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," *ACM Symp. on Operating Systems Principles (SOSP)*, 2013.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2012.

[15] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM SIGPLAN Notices*, 2013.

[16] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," *ACM SIGPLAN Notices*, 2016.

[17] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," *USENIX Conf. on File and Storage Technologies (FAST)*, 2015.

[18] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," *IEEE Int'l Conf. on Big Data*, 2017.

[19] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," *USENIX Annual Technical Conf. (ATC)*, 2015.

[20] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable simd-efficient graph processing on gpus," *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[21] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: Vertex-centric graph processing on gpus," *Int'l Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, 2014.

[22] A. Santoro, D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia, and T. Lillicrap, "A simple neural network module for relational reasoning," *Conf. on Neural Information Processing Systems (NIPS)*, 2017.

[23] R. Palm, U. Paquet, and O. Winther, "Recurrent relational networks," *Conf. on Neural Information Processing Systems (NIPS)*, 2018.

[24] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," *USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2018.

[25] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

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

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

[28] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, "Attention-based graph neural network for semi-supervised learning," *arXiv preprint arXiv:1803.03735*, 2018.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Conf. on Neural Information Processing Systems (NIPS)*, 2017.

[30] H. Zhao, "High performance machine learning through codesign and rooflining," Ph.D. dissertation, UC Berkeley, 2014.

[31] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, 2013.

[32] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[33] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu," *European Conf. on Parallel Processing (Euro-Par)*, 2018.

[34] M. Harris *et al.*, "Optimizing parallel reduction in cuda," http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, 2012.

[35] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing cnn model inference on cpus," *USENIX Annual Technical Conf. (ATC)*, 2019.

[36] L. Wang, Z. Chen, Y. Liu, Y. Wang, L. Zheng, M. Li, and Y. Wang, "A unified optimization approach for cnn model inference on integrated gpus," *Int'l Conf. on Parallel Processing (ICPP)*, 2019.

[37] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[38] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," *Conf. on Neural Information Processing Systems (NIPS)*, 2018.

[39] "Minigun: Light-weight gpu kernel interface for graph operations," https://github.com/dglai/minigun, 2019.

[40] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Conf. on Neural Information Processing Systems (NIPS)*, 2017.

[41] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the graphblas," *IEEE High Performance Extreme Computing Conf. (HPEC)*, 2016.

[42] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Int'l Conf. on Very Large Data Bases (VLDB)*, 2015.

[43] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011.

[44] D. Zheng, D. Mhembere, V. Lyzinski, J. T. Vogelstein, C. E. Priebe, and R. Burns, "Semi-external memory sparse matrix multiplication for billion-node graphs," *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 2016.

[45] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2017.