TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU

Qiang Fu
charlesfoo@gwu.edu
George Washington University
Washington, DC, USA

Yuede Ji
yuede.ji@unt.edu
University of North Texas
Denton, TX, USA

H. Howie Huang
howie@gwu.edu
George Washington University
Washington, DC, USA

ABSTRACT

Graph Neural Networks (GNNs) are an emerging class of deep learning models on graphs, with many successful applications, such as, recommendation systems, drug discovery, and social network analysis. The GNN computation includes both regular neural network operations and general graph convolution operations, which take the majority of the total computation time. Though several recent works have been proposed to accelerate the computation for GNNs, they face the limitations of heavy pre-processing, low efficient atomic operations, and unnecessary kernel launches. In this paper, we design TLPGNN, a lightweight two-level parallelism paradigm for GNN computation. First, we conduct a systematic analysis on the hardware resource usage of GNN workloads to deeply understand the specialties of GNN workloads. With the insightful observations, we then divide the GNN computation into two levels, i.e., *vertex parallelism* for the first level and *feature parallelism* for the second. Next, we employ a novel hybrid dynamic workload assignment to address the imbalanced workload distribution. Furthermore, we fuse the kernels to reduce the number of kernel launches and cache the frequently accessed data into registers to avoid unnecessary memory traffics. Together, TLPGNN is able to significantly outperform existing GNN computation systems, such as DGL, GNNAdvisor, and FeatGraph, by 5.6×, 7.7×, and 3.3×, respectively, on the average.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → **Massively parallel algorithms**; • **Computer systems organization** → *Neural networks.*

KEYWORDS

Graph Neural Networks, GPU, Performance

ACM Reference Format:
Qiang Fu, Yuede Ji, and H. Howie Huang. 2022. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531467

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06…$15.00
https://doi.org/10.1145/3502181.3531467

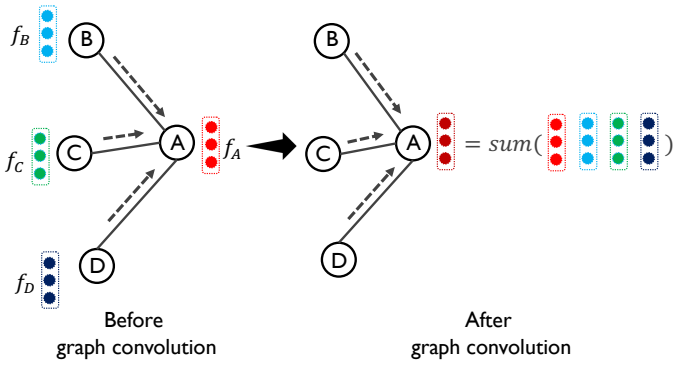

Figure 1: An example of the graph convolution operation in GCN. The sum operation is weighted based on degree.

1 INTRODUCTION

Thanks to the ability to capture the relationships (*edges*) between different entities (*vertices*), the *graph* data structure [48] has long been utilized to model a myriad of real applications, such as social networks [33], molecular graph structures [49], and biological-protein networks [47]. In recent years, there has been a great surge of research interest on graph neural networks (GNNs), in which each vertex recursively aggregates feature vectors of all its neighbors to compute its new feature vector. GNNs can accurately represent the high-dimensional features of vertices and edges as well as the structure information into a low-dimensional embedding [22], which can be further utilized for various tasks, e.g., node classification [12], edge classification [25], and link prediction [53]. GNNs have achieved promising results in various domains, including chemistry [13], social science [9], knowledge graph [2, 14], recommendation system [52], and neuroscience [3].

Motivation. GNN operations can be mainly classified into two categories: classical dense neural network operations and general graph convolution. Dense neural network operations, e.g., matrix multiplication and activation function (*ReLU*), are applied to the matrices denoting the features of all the vertices (or edges). The general graph convolution operations in each layer accumulate the features of each vertex's neighbors, conduct certain operations (e.g., *sum*) on them, merge with its own features, and generate the new features for the next layer. Figure 1 shows an example of the graph convolution operation in a representative GNN, i.e.,

GCN [26]. Specifically, for vertex A, it accumulates the features of its neighbors B, C, D, denoted as f_B, f_C, and f_D. Then, it merges with its own feature f_A, applies weighted sum operation $sum(\cdot)$, and generates the new feature f'_A. The graph convolution is reported to take about 60% of the total GNN computation time [18]. Because of that, there are high demands for better computation methods for graph convolution operations.

Limitation of state-of-art approaches. A major difference between GNNs and traditional graph algorithms is the feature vector on the vertex or edge. The dimension of the feature vector can be dramatically large (e.g., the input feature size of the Cora dataset is 1,433 [31]). With such a newly added workload, the runtime characteristics of GNNs on the GPU are drastically different from traditional graph algorithms. Though several existing works have observed this and designed various optimization techniques [18, 19, 46], they fail to provide a deep understanding of the impact of such optimizations on hardware (e.g., GPU) resource usage, e.g., atomic operations, coalesced memory access, and kernel launches. For example, GNNAdvisor [46] simply claims that the huge feature size will make the impact of workload imbalance severer, while the impact is not clearly quantified. Therefore, we believe an in-depth understanding of hardware resource usage is the foundations for designing more efficient computation techniques.

In addition, existing works [10, 18, 19, 44, 46] still face several limitations to unleash the full potential of GPU. 1) *Heavy pre-processing.* Existing works observe GNN computations can cause low data locality [42]. To address that, they pre-process the graph by reordering the vertices to make the ones sharing more common neighbors closer [18, 19, 46]. However, the overhead of the pre-processing is non-trivial, which can make the total computation time more than the conventional computation methods. 2) *Low efficient atomic operations.* To address the workload imbalance issue when distributing the workload based on vertex, existing works design different edge-centric techniques [44, 46]. However, to get the correct result, they have to rely on atomic operations, which turn the parallel operations into serial. This can actually bring more overhead than the benefit gained from a more balanced workload. For example, GNNAdvisor [46] balances the workload by partitioning each vertex's neighbors into several fix-sized groups, while they have to atomically aggregate the messages from all the groups. 3) *Unnecessary kernel launches.* Existing works launch many kernels to implement a GNN, causing unnecessary read, store, and waiting onto global memory [10, 44]. For example, DGL [44] uses 18 kernels to implement the graph attention network (GAT) as it relies on the fine-grained sparse matrix kernels provided by NVIDIA cuSPARSE [36].

Key insights and contributions. Motivated by these, we design TLPGNN, a lightweight two-level parallelism paradigm specifically designed for GNN computation on GPU. First, we conduct a systematic analysis to deeply understand the performance of GNN workloads. We demonstrate our observations with extensive profiling data which, to the best of our knowledge, has never been studied in previous works. In summary, we observe three interesting insights. 1) The atomic operations, especially atomic writing, can drastically hurt the performance of GNN computation. 2) Coalesced memory access is critically important for GNN workloads. 3) Using fewer kernels can lead to better performance in general.

Second, we propose a lightweight warp-centric two-level parallelism paradigm for GNN computation on GPU. In the first level, we apply vertex parallelism by mapping each vertex to one warp, i.e., warp-vertex mapping. The benefits of vertex parallelism are two-fold. On one hand, it can avoid atomic operations as the workload of each vertex is independent of others. On the other hand, it can eliminate the branch divergence in each warp, that happens when the threads in one warp come into different control paths. In the second level, we apply feature parallelism by mapping each dimension of the feature vectors to the threads within a warp. As the threads within one warp are accessing consecutive global memory, this can make the memory access coalesced where the memory addresses requested by all the threads in one warp fall into just a few cache lines.

Third, as vertex parallelism can cause imbalanced workload distribution, we design a hybrid dynamic workload distribution technique. It can switch between hardware- and software-implemented dynamic workload distribution strategies based on the properties of the input graph. In addition, we use kernel fusion to reduce the number of launched kernels. We also use register caching to cache both the frequently accessed graph index boundary and the intermediate aggregation result.

Experimental methodology We evaluate our system by comparing with three state-of-the-art GNN computation frameworks, including Deep Graph Library (DGL) [44], GNNAdvisor [46], and FeatGraph [18]. We tested four widely used GNN models, i.e., Graph Convolutional Network (GCN) [26], GraphSage [15], Graph Isomorphism Network (GIN) [51], and Graph Attention Network (GAT) [43]. Our evaluation shows that TLPGNN is able to significantly improve the performance on average by 5.6×, 7.7×, and 3.3×over DGL [44], GNNAdvisor [46], and FeatGraph [18], respectively. More importantly, TLPGNN does not rely on any pre-processing techniques.

Limitations of the proposed approach. We find two major limitations in the current stage of our work. 1) Our method is designed for GNN models on homogeneous graphs, that only include one type of vertices and edges. However, our designs for the kernel is generic and should be also applicable to the GNN models on heterogeneous graphs [28] with reasonable modifications. We will explore this in future. 2) Although TLPGNN is evaluated on one GPU, we believe our techniques can also be deployed on a multi-GPU setting with the help of graph partition techniques, e.g., METIS [23], which we leave for future works.

2 BACKGROUND

2.1 Graph Neural Network

Mathematically, in the l-th layer of a GNN, we denote h_u^l as the feature vector of vertex u, e_{vu}^l as the feature of edge from vertex v to u. The computation in each layer of GNNs can be expressed as follows:

$$a_u^l = \bigoplus_{v \in \mathcal{N}(u)} \psi(h_u^l, e_{vu}^l, h_v^l), \tag{1}$$

$$h_u^{l+1} = \sigma(a_u^l), \tag{2}$$

where ψ is a customizable function applied on the features of the edge, source, and destination vertex, $\bigoplus$ is a reduce function that

aggregates the features from all the edges of u, and σ is the function applied on the result of the reduce function. In the implementation of GNN's one layer, the operations applied on the features of vertices (edges) mainly follow a three-phase pattern chronically. First, the features are processed by some regular neural operations, e.g.,, *Dropout*, and *Matmul*. Next, all the features are fed into a phase called "Graph Convolution", in which each vertex gathers the features of all its neighbors and the associated edges, applies neural operations, combines the results with its own features, and uses a reduce operation (e.g. *max*, *mean*) to produce a new feature vector. At last, before passing to the next layer, the features are usually applied by operations such as activation function, batch normalization, and softmax, similar to traditional DNNs. The input graph structure is only involved in the graph convolution phase. As a result, the computation that taken place during graph convolution has a distinctive difference from the other two phases, that consist of regular dense neural operations. Graph convolution is naturally sparse due to the sparsity of the input graph. In this work, we mainly focus on the performance of graph convolution as it takes majority of the computation time.

2.2 GPU Architecture

In the CUDA programming model, a kernel is a function executed in a GPU[1] device for parallel tasks. It consists of a grid of thread blocks and a grid can have multiple thread blocks (compute thread arrays), e.g., 256. Figure 2 shows a brief overview of the mapping between the CUDA programming model and the underlying GPU hardware. After the kernel has been launched, each block is assigned to a *streaming multiprocessor* (SM). Then, a certain amount of hardware resources including registers and shared memory are allocated to each block. When a block completes the computation, the SM releases the occupied hardware resources and reallocate them to the next block. This process repeats till the end.

Each block can have up to 1,024 threads and every 32 continuous threads are grouped as a *warp* [35]. For example, NVIDIA Volta V100 consists of 80 SMs, each of which has 64 KB register file and supports up to 64 warps. The threads in one warp run in parallel following single instruction multiple data (SIMD). If there are branches, e.g., `if-else` statements, first the threads satisfying the `if` condition become active, but the remaining threads turn into idle. After the `if` branch completes, the previously idle threads become active for the `else` branch, while the other threads are idle. This is known as *branch divergence*, which can reduce SM utilization thus greatly lower the performance. During the execution of a kernel function, when data in global memory are needed, each warp sends a memory request containing the required data from all the threads in that warp. The memory controller receives the request and locates the cache line of the corresponding data. Then, it issues one or more transactions to the global memory to retrieve the data. In this process, if the required data belongs to a few shared cache lines, e.g., all the threads in one warp are accessing continuous memory addresses, then the issued memory transactions are kept to a low level, resulting in high bus and cache utilization. This is known as *coalesced memory access*, which can lead to better performance. However, if all the required data belong to completely different cache lines, the memory controller has to issue a large number of memory transactions (up to 32), leading to low bus and cache utilization. This is known as *uncoalesced memory access*.

[1]We use Nvidia GPUs as representatives.

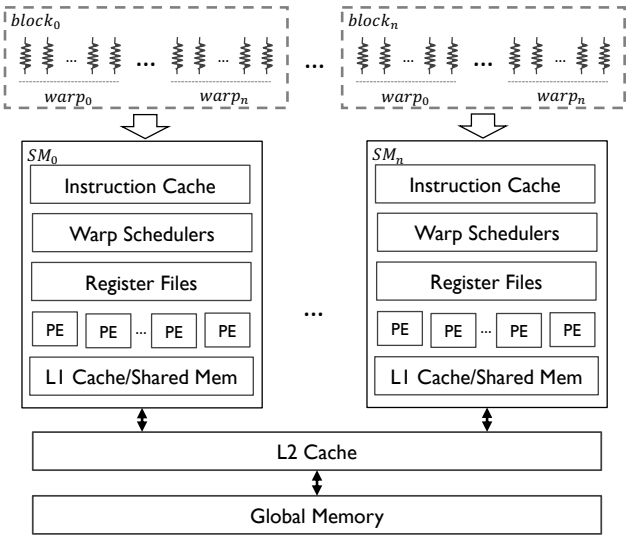

Figure 2: An overview of the mapping between CUDA programming model and GPU architecture. PE refers to *processing element*, which could be FP32 core, FP64 core, INT core, or Tensor core for different computing tasks.

2.3 GPU Profiling

GPU profiling collects runtime statistics of the CUDA program. In this work, we use NVIDIA Nsight Compute, an interactive kernel profiler for CUDA applications. It provides detailed performance metrics via a user interface and command-line tool [37]. We use two well-known metrics, including *cache hit*, and *global memory traffic*. Besides, we also use the following four metrics [19, 46].

- *Streaming multiprocessor (SM) utilization* rate measures the utilization of the computing resources on a GPU, such as, pipelining, and issue queue. A low value usually indicates that the program does not well utilize the corresponding resources.
- *Achieved occupancy* is the ratio of the number of active warps per SM over the maximum number of possibly active warps. A higher value denotes a more balanced workload distribution among different warps.
- *Sector per request* is the average number of transactions issued by the memory controller for each global memory request. A lower value indicates more coalesced memory access.
- *Stall for long scoreboard* is the average cycles spent on waiting for memory operations of all resident warps. A higher value means many cycles are waiting for the data from the memory, which is low efficient.

Table 1: Profiling result of *Push, Edge-centric, GNNAdvisor* and *Pull* implementation for GCN over ovcar_8h dataset with 128 feature size.

Metrics	Push	Edge	GnnA.	Pull
Runtime (ms)	3.3	2.8	10.4	**1.8**
Mem load traffics (GB)	**1.5**	1.6	2.6	1.7
Mem atomic store traffics (GB)	1.3	1.3	2.9	**0.0**
Stall for long scoreboard (cycle)	36.7	80.1	45.1	**26.5**
SM utilization	14.3%	17.4%	23.5%	**41.1%**

3 UNDERSTANDING PERFORMANCE OF GNN

In this section, we present a systematical analysis and profiling to understand the runtime characteristics of GNN workloads. The testing and profiling are on a NVIDIA Volta V100 GPU with 32GB DRAM and the data are collected with NVIDIA Nsight Compute. The impact can be summarized from three major perspectives, i.e., *atomic operation*, *coalesced memory access*, and *kernel launches*.

3.1 Atomic Operation

In a CUDA program, atomic operations help to avoid race conditions when multiple threads are writing to the same memory address concurrently. A number of implementations for the graph convolution of GNNs have utilized atomic operations. In the implementation of using push updating policy [4], every vertex writes the message to all of its neighbors along the out-going edges concurrently. Therefore, the atomic operations are used to avoid race conditions as different vertices are updating the same neighbors. In edge-centric processing, each edge passes the message and writes to the destination vertex concurrently, using the atomic operation to guarantee the accuracy of the result when multiple edges are writing to the same vertex [40]. GNNAdvisor divides each vertex's neighbors into groups with a fixed size, and uses atomic operations to combine the intermediate results from different neighbor groups into a single one. However, atomic operations will introduce extra overhead.

To show the impact of atomic operations on GNNs, we tested four different implementations of the graph convolution operations used in GCN and many other GNNs. They are push, edge-centric, pull updating policy, and the implementation from GNNAdvisor [46]. Push, edge-centric, and GNNAdvisor all use atomic operations to update the feature vectors of each vertex, while pull is atomic free. Table 1 shows the performance and related profiling metrics, where *edge* refers to edge-centric and *GnnA.* refers to GNNAdvisor. First, one can see that pull achieves 1.8×, 1.6×, and 5.8×speedups over push, edge-centric, and GNNAdvisor. From the profiling metrics, we notice that push, edge-centric, and GNNAdvisor have a large amount of memory atomic store traffics, while pull does not. The reason is the former three methods need to send an atomic write request to memory for each edge, while pull does not. Higher memory store traffics can make each warp spend more cycles waiting for memory operations (stall for long scoreboard), which in turn decreases the utilization of streaming multiprocessors.

Observation I: Optimizations with atomic writing can drastically lower the performance because of the huge extra overhead. The larger feature size of GNN will enlarge the extra overhead from the atomic operations, leading to low performance despite of the performance benefits from the optimizations techniques.

Table 2: Comparison between the implementations using one thread and half warp for one vertex.

Metrics	One Thread	Half Warp
Runtime (ms)	434.9	**15.9**
Sector per request	9.2	**2.1**
L1 cache hit	**42.3%**	34.4%
Long scoreboard (cycle)	251.8	**75.2**

3.2 Coalesced Memory Access

Coalesced memory access refers to the threads within the same warp accessing the continuous memory addresses, as illustrated in Section 2.2. In traditional graph processing, coalesced access is hard to achieve because each vertex is associated with a scalar value for feature, and the memory access is random due to the irregular graph structure. Thus, the memory address accessed by each thread in the same warp could fall into different cache lines. For each cache line, the memory controller has to issue one request to retrieve it from the global memory, leading to uncoalesced memory access.

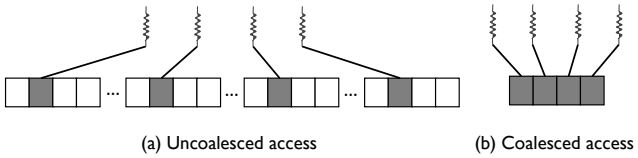

Figure 3: Memory access pattern

Nevertheless, GNN's new characteristic (i.e., large feature size) provides an opportunity to achieve a memory access pattern that is more coalesced. Since the feature vector of each vertex is stored in consecutive memory addresses, the access to one vertex's feature could be coalesced if adjacent threads in the warp are accessing the adjacent indices of the feature vector. To demonstrate this, we show the comparison between two different implementations of GCN's graph convolution operation. The first one uses a single thread to process one vertex, so each thread of a warp needs to access the same position of different vertices' features at the same time, which is demonstrated in Figure 3(a), resulting in memory requests for many different cache lines. The second implementation uses half warp (16 threads) to process one vertex. The threads in the same half warp accesses 16 consecutive positions of a vertex's feature vector, which is more coalesced than the previous implementation, as shown in Figure 3(b).

Table 2 shows the comparison of runtime and related profiling metrics. First, one can see the implementation using half warp for one vertex outperforms the runtime of *one thread* by 27.3×. From the profiling metrics, we notice that *one thread* has an extremely higher value of sector per request than the *half warp* implementation, i.e., 9.2 vs 2.1. This demonstrates that each warp in the first implementation issues many more transactions to the global memory than the second implementation. This can lead to huge waste on the memory bus and caches because too many unnecessary data

Table 3: Profiling result of DGL, three-kernel, and one-kernel implementation for GAT's graph convolution on the reddit dataset with feature size 32.

Metrics	DGL	Three-Kernel	One-Kernel
GPU Kernel launch	18	3	**1**
Runtime (ms)	122	74.6	**16.1**
GPU time (ms)	102	70.9	**15.6**
Runtime - GPU time (ms)	20	3.69	**0.5**
Global mem usage (GB)	10	2.8	**1.5**
Global mem traffics (GB)	35.9	19.5	**4.8**
Stall for long scoreboard (cycle)	241.4	241.1	**18.0**
Average SM utilization	13.8%	6.7%	**51.7%**

is loaded from DRAM. As a result, each warp in the first implementation waits for more cycles than the second one. In conclusion, we get our second observation.

Observation II: Coalesced memory access can lead to tremendous performance improvement if more threads within one warp are used to process one vertex for GNN workloads.

3.3 Kernel Launches

A kernel function is the minimum executing unit of a CUDA program. The graph convolution of a GNN could be implemented as different numbers of kernels, and the launched number of kernels has a major impact on the performance. Usually, more kernels mean higher kernel launch overhead (i.e., it takes time to launch a GPU kernel) and more memory traffics because more intermediate results need to be stored into the memory. To show the impact of different number of kernels, we implement Graph Attention Network (GAT)'s graph convolution phase as one kernel and three kernels. We also compare with an implementation from a popular GNN framework DGL, which launches 18 GPU kernels for the graph convolution of GAT.

Table 3 summarizes the performance and profiling results. One-kernel implementation outperforms DGL and three-kernel implementation by 7.5×and 4.6×, respectively. We got three observations from the profiling results. First, too many kernel launches will bring extra overhead, which comes from the CUDA runtime to execute the kernel on GPU. We measure such overhead with the difference between runtime and GPU time (denoted as "Runtime - GPU time (ms)" in Table 3). The runtime is the execution time conducting all the computation and the GPU time is the total time spent on GPU. One can see, DGL takes 20 ms, three-kernel implementation takes 3.69 ms, while one-kernel implementation only takes 0.5ms. Second, using more GPU kernels will consume more global memory because the intermediate data needs to be stored in the global memory for subsequent kernels. We can see that DGL uses 10 GB, three-kernel implementation uses 2.8 GB, while one-kernel implementation only uses 1.5 GB. Consequently, this will increase the global memory traffic. For example, the total memory traffic of DGL is 35.9 GB, three-kernel implementation is 19.5 GB, while one-kernel implementation is only 4.8 GB.

To conclude, implementations with more kernels will generate more memory traffic, causing the increase of memory stalls. That eventually decreases SM utilization, and hurts the overall performance. What is worse, the large feature size of GNN can further exacerbate this limitation. In short, we get the following observation.

Observation III: The graph convolution of GNNs should be implemented with as few kernels as possible for better performance.

4 TWO-LEVEL PARALLELISM

In this section, we present our novel warp-centric two-level parallelism computation method.

4.1 Overview

The CUDA programming model provides a thread hierarchy with different levels (i.e., thread, warp, and block) to parallelize the computing tasks running on GPUs. And there are plenty of parallelism opportunities in the computation of GNN's graph convolution. To unleash the full power of GPU, we have to find the best mapping relation between the workload and GPU thread hierarchy. Since the CUDA programming model is organized as a multi-level hierarchy, we also divide the computation in the graph convolution of GNNs into two levels. In the first level, the granularity of computation is measured by the entities in the graph, i.e., vertex and edge. Each vertex or edge has a certain amount of computing tasks associated with it, and it can be assigned to one level of the CUDA thread hierarchy. In the second level, we focus on the concrete computation task of individual vertex or edge, that is, the specific computation task for this vertex. In each level of the computation, we have different choices of parallelism strategies and mapping relations between CUDA thread hierarchy, which we will illustrate in the following subsections.

4.2 First Level: Vertex Parallelism

In the first level, the parallelism opportunity comes from the fact that the workload associated with each vertex or edge can be processed independently from others. Therefore, there are two different types of parallelism: vertex parallelism and edge parallelism. Vertex parallelism means processing multiple vertices in parallel, in contrast to edge parallelism, which processes multiple edges in parallel. Although edge parallelism can avoid workload imbalance caused by uneven edge distributions in vertex parallelism [40], the huge overhead brought by atomic operations will dramatically hurt the performance based on our Observation I. As a result, we choose to use vertex parallelism in our design.

The next step is to decide how to map the workloads associated with vertices in the graph to the underlying GPU's execution units in its thread hierarchy. With the vertex-centric processing pattern, we can assign the workload of each vertex to one thread, warp, or CTA (thread block). Here we argue that mapping each vertex to a warp (as shown in Figure 4(a)) is superior to the other two methods for better performance of graph convolution of GNNs.

First, using one CUDA thread for each vertex in the graph will lead to *divergent execution* within each warp (as discussed in Section 2.2), due to the uneven edge distribution of vertices. To be more specific, within each warp, threads processing vertices with the small number of neighbors would become idle after the jobs are done. Its memory access pattern is also not optimal for the GPU, which will cause too much useless data load from global memory to caches.

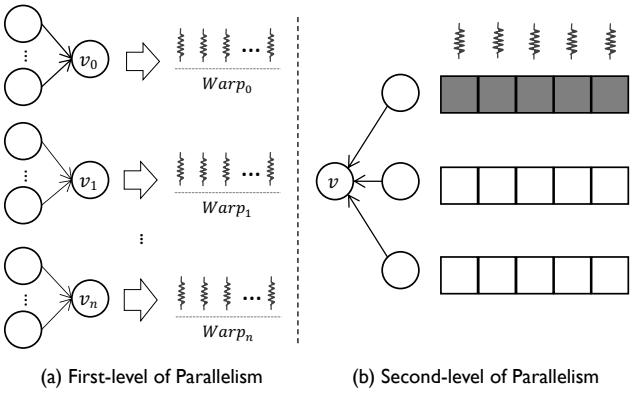

Figure 4: Two-level Parallelism. (a) First-level: The computation of graph convolution of each vertex in the input graph is assigned to single warp of GPU for execution. (b) Second-level: Each element of feature vectors is assigned to an individual CUDA thread.

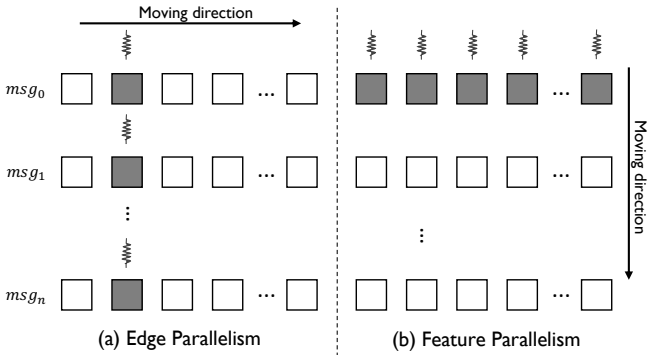

Figure 5: Two different looping schemes for each warp to process a vertex. $msg_0, \cdots, msg_n$ **are the messages generated by the connecting edges of the vertex being processed.**

On the other hand, mapping a vertex to a whole CTA introduces synchronization overhead into the kernels. Specifically, to avoid race conditions and ensure the correctness of results, coordinating the warps in the same CTA to accomplish the computation of a single vertex requires extra sync operations, and atomic operations are needed to update the resulting feature of the vertex.

Warp is the actual minimum processing unit of GPU hardware, executing SIMD instruction over multiple data with a 32-wide lane. Assigning each vertex to a single warp means the workload running by each individual warp won't interfere with each other, which means we can avoid any possible synchronization overhead. Since each CUDA threads in the warp are working the same vertex, they will follow the same control path, eliminating the *branch divergence* caused by uneven edge distribution of the input graph. Furthermore, combined with the techniques in following subsection, we can achieve coalesced memory access when loading features of neighbors for graph convolution.

4.3 Second Level: Feature Parallelism

The reason why we need a second level of parallelism is two-fold. First, the computation associated with each vertex could be parallelized internally. Second, since we use one warp to process each vertex, there is a parallel opportunity within one warp, given that each warp has 32 threads. The computation within a vertex can be expressed as a nested loop with two options for the looping order: *edge-then-feature* or *feature-then-edge*. The former looping scheme first iterates on each edge of the vertex being processed, and then for each index of the feature dimension axis, calculating the value in the index of the message generated by the edge and updating the corresponding index of the vertex's resulting feature. The latter one uses reversed order, instead.

Since each warp has 32 threads, the inner loop of the nested iteration could be partially parallelized. As shown in Figure 5(a), in the *feature-then-edge* order, each warp can process the elements at the same dimension of up to 32 messages generated by edges in one step; after one dimension is finished, it moves to the next dimension. In this parallel strategy, multiple edges are processed at the same time, and feature dimensions are handled sequentially. So we also refer to this as *edge parallelism*. While in the *edge-then-feature* order (Figure 5(b)), each warp processes consecutive 32 elements of the same message generated by one edge in each step; after one edge is finished, it moves to the next edge. And one can notice that in this design, edge parallelism is ignored because all edges are processed sequentially and multiple feature dimensions are processed concurrently. So we call it *feature parallelism*.

In our kernel design for graph convolution, a *feature parallelism* looping scheme is adopted for two major reasons. First, in the *edge parallelism* scheme, all threads in one warp work on the same dimension of messages, then the same element of the resulting feature needs to be updated according to those values. So each thread has to read and write the same memory address, introducing atomic operation overhead to avoid race conditions. In contrast, threads of one warp in the *feature parallelism* scheme process different dimensions at all times, which means they need to update different elements of the resulting feature. So we can avoid the synchronization overhead brought by the atomic operations. Second, to calculate the element values at the same dimension of messages from different edges, threads need to access the data distributed in scattered addresses of global memory, leading to uncoalesced memory access. Nevertheless, the memory access pattern in *feature parallelism* is perfectly coalesced, because all threads process a continuous segment of a single message. As a result, we choose to make use of *feature parallelism* and ignore the *edge parallelism* for the free from atomic overhead and coalesced memory access, according to our Observation I and II.

5 HYBRID WORKLOAD BALANCING

As vertex parallelism may cause imbalanced workload distribution, we design a hybrid workload balancing technique with both software-based and hardware-based dynamic workload distribution.

Hardware-based assignment. As we mentioned in Section 2.2, GPU dynamically assigns blocks to streaming multiprocessors for execution. This characteristic can be utilized to implement our

Algorithm 1: Software-Based Dynamic Workload Assignment

Parameters: Number of warps: $nwarp$;
Total number of vertices in graph: $nvertex$;
Number of vertices to process at each iteration: $step$;

```
1 begin
2     Initialize global integer variable G ← 0 ; // G is the
        starting index of unfinished vertices
3     for warp ∈ Kernel do in parallel
4         sindex ← AtomicAdd(G, step) ; // Add step to G
            atomically and return the old value of G.
5         while sindex < nvertex do
              /* Processing vertices with consecutive
                 indices.                                  */
6             for v ∈ [sindex, min(sindex + step, nvertex)) do
7                 Processing vertex v;
8             sindex ← AtomicAdd(G, step) ; // Request
                workload again.
```

workload assignment technique. Specifically, we use the same number of warps as the number of vertices of the input graph, and each of these warps processes only one vertex. Then, GPU hardware will schedule the execution of these warps in a dynamic mode, in which hardware resources will be released after one block is done and then be allocated to the next new one. Since the minimum scheduling unit is block, there will be workload imbalance within one block if it has more than one warps. As a result, the number of warps in each block becomes a tunable parameter for this implementation. Fewer warps mean a more balanced workload but higher hardware scheduling overhead, and more warps mean a more imbalanced workload but lower hardware scheduling overhead.

Software-based assignment. Software-based dynamic workload assignment can be understood as the task pool model. All the vertices needed to be processed are put into a task pool. Each warp takes a fixed number of tasks from the pool at a time and executes the computation on these vertices. After finishing the current workload, it checks if there are remaining tasks in the pool. If so, then this warp will continue to pull the tasks from the pool until it is empty. Specifically, as shown in Algorithm 1, we maintain a global variable initialized as zero in the global memory to record the starting of unfinished vertices, then each warp reads it and then adds a pre-defined integer to the variable atomically. The returned value indicates the starting index of a vertex to process at this iteration for this warp. After finishing the workload of consecutive vertices of the pre-defined number, each warp repeats the same action in the next loop until the global variable exceeds the total number of vertices. This implementation uses a fixed number of warps for all inputs at the beginning, which normally is the maximum warps all SM can run in parallel. All the hardware resources are allocated once, so there is no hardware scheduling overhead.

Heuristic hybrid assignment. Based on our benchmarking, we obtain two observations for the question of which workload assignment method to choose. First, when the total number of vertices in the input graph is relatively high, the software-based method is better than the hardware-based one. This is because the hardware-based method needs to allocate too many blocks, leading to high hardware schedule overhead. Second, the software-based method outperforms hardware-based if the average degree of vertices is high. The high average degree means the workload defined in each vertex is heavy, then the overhead from the atomic operation will become negligible compared with computation on the vertex. Therefore, we utilize a heuristic-based discriminant to determine which method to use. Specifically, we use software-based dynamic workload assignment when the number of vertices is over 1M or the average degree is over 50, otherwise, we use the hardware-based method.

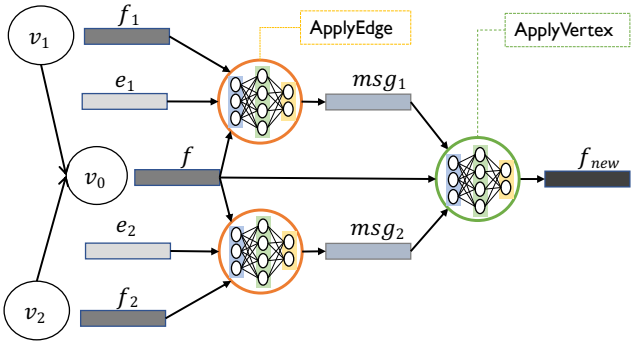

Figure 6: Two types of building-block kernels for the implementation of GNN's graph convolution, *ApplyEdge*, and *ApplyVertex*.

6 KERNEL FUSION AND REGISTER CACHING

Kernel fusion. As we concluded in Observation III, using fewer kernels for graph convolution can decrease memory usage and traffics. While most existing works, such as DGL [44], FeatGraph [18], use multiple kernels for the implementation. In general, there are two types of basic kernels that could be considered as the building block for the computation in graph convolution: *ApplyEdge*, and *ApplyVertex*. As shown in Figure 6. *ApplyEdge* is responsible for calculating the message generated by each edge, taking as input the features of source and destination vertex, and the edge. *ApplyVertex* is used to aggregate the message of each edge and generate the new feature for each vertex.

Most GNN models' graph convolution can be expressed as a combination of these two types of kernels. Graph Attention Network (GAT), which is more complicated, calculates an attention coefficient for each edge and then aggregates all neighbors' features weighted by the softmax over all attentions; so one *ApplyEdge* to obtain the attentions, two *ApplyVertex* for the softmax and aggregation can be used to express the computation in its graph convolution [44].

However, using separate kernels means that the messages generated by the *ApplyEdge* kernel for all edges need to be written to the global memory. This could lead to significant computational and memory bottlenecks, especially when each edge generates

```
int start = indptr[des_v]; // caching the index boundary
int end = indptr[des_v+1]; // caching the index boundary
float ret = 0.0;   // caching the intermediate reduction result

for (int i = start; i < end; ++i) {  // iterating the edge list
     ret += features[indices[i]*blockDim.x+threadIdx.x];
}
result[threadIdx.x] = ret;
```

(a) With register caching

```
result[threadIdx.x] = 0.0;
for (int i = indptr[des_v]; i < indptr[des_v+1]; ++i) {
     result[threadIdx.x] += features[indices[i]*blockDim.x+threadIdx.x];
}
```

(b) Without register caching

Figure 7: The CUDA codes for each thread processing one vertex in GCN model with/without register caching the index boundary and intermediate reduction result. The nvcc compiler automatically use registers for the local temporal variables defined within a kernel.

high-dimensional messages [39]. To address this problem, the computation in the graph convolution of GNN models is fused into a single kernel in our design. The fused kernel generates and aggregates the message of each edge collectively without explicitly storing the message. By doing so, first, we can avoid redundant DRAM usage, which could be extremely huge if the input graph has a large number of edges or the model uses a long message size. Second, the total memory traffics could be decreased by a large extent, which can greatly benefit the overall performance, given that global memory operations in GPU are highly expensive.

Register caching. Each SM has a large number of registers, e.g., 65,536 32-bit for each V100 SM. Each thread can use up to 255 registers and perform four register access for each clock cycle. In our design, we find that using registers for two types of data objects can greatly improve the overall performance: index boundary and intermediate reduction result.

As shown in Figure 7, before iterating the edge list of the vertex, each thread in the warp needs to read the starting index of the edge list, and after each iteration, it also needs to check if the end is reached by reading the end index. Since the index boundary is accessed frequently, using registers to cache those two values will reduce unnecessary global memory requests to a large extent. Furthermore, GNN models use a reduction operation to aggregate the messages from all edges into a single resulting feature. Updating the resulting feature into the global memory in each iteration will cause too many read and store transactions. As a result, caching the intermediate reduction result in registers can eliminate most of them; only one write operation is needed after the iteration over the edge list.

7 EXPERIMENT AND EVALUATION

7.1 Experiment Setup

GNN models. We test four representative GNN models, including Graph Convolutional Network (GCN) [26], GraphSage [15], Graph Isomorphism Network (GIN) [51], Graph Attention Network (GAT) [43]. They are widely used in previous works and have a remarkable diversity in terms of the computation [8]. GCN has been used in many semi-supervised or unsupervised tasks, such as node embedding, graph classification. it applies a weighted sum operation for the neighbor's features for each vertex in graph convolution. GraphSage and GIN differ from GCN as to how they aggregate messages from neighbors during graph convolution. GAT introduces the attention mechanism [16] to learn the different weight coefficients of different neighbors.

Table 4: Graph benchmarks sorted by edge count (K: thousand, M: million).

Dataset (Abbr.)	vertex #	edge #	avg. degree
Citeseer (CS)	3.3K	9.2K	2.7
Cora (CR)	2.7K	10.5K	3.8
Pubmed (PD)	19.7K	88.6K	4.5
Ogbn-arxiv (OA)	169K	1.1M	6.5
PPI (PI)	56K	1.6M	28.5
DD (DD)	334K	1.6M	4.9
Ovcar-8h (OH)	1.8M	3.9M	2.2
Collab (CL)	372K	24.9M	66.9
Ogbn-protein (ON)	132K	79M	607
Reddit (RD)	232K	114M	491
Ogbn-product (OT)	2.4M	123.7M	51.5

Compared works. We compare TLPGNN with three recent works on GNN computation. 1) Deep Graph Library (DGL) [44] is a widely used python framework aiming for easy-to-use and high-performance GNN computation. It can take various tensor-based deep learning frameworks as back ends, such as TensorFlow [1], PyTorch [38], and MxNet [5]. To accelerate the computation, DGL uses the sparse-dense matrix multiplication (spmm) kernels from cuSPARSE [36]. 2) GNNAdvisor [46] is one of the state-of-the-art GNN computation systems. It first improves the data locality by reordering the graph. Then, it uses two-dimensional workload management to improve GPU utilization for GNN workloads. 3) FeatGraph [18] is a tensor compiler-based system to generate efficient GPU kernels with the TVM framework [6]. A user can customize different GNN kernels with user-defined functions.

Datasets. Table 4 lists the real-world datasets for evaluation. They are commonly used to evaluate GNN models for various tasks, such as node classification, link prediction, and graph classification [7, 26, 46]. We also include three datasets (i.e., OA, ON, OT) from Open Graph Benchmark [17], a realistic benchmark suite for GNNs. We add additional data, e.g. vertex features, edge features, and weight parameters, which are initialized to random 32-bits floating numbers, following previous works [46].

Setting. We implement our kernel design using C++ & CUDA C language, and its PyTorch front end using Python Language. The total lines of code is roughly 3,000. We evaluate TLPGNN and compared works on a server with 24-Core Intel(R) Xeon(R) Gold 6126 2.6GHz CPU and a Nvidia Tesla V100 GPU (32GB DRAM). The server installs Centos 8 operating system, CUDA 11.1, GCC/G++

Table 5: Execution times (in ms) of TLPGNN, DGL, GNNAdvisor (GNNA.) and FeatGraph (FeatG.) with feature size of 32. The speedup refers to the performance improvement of TLPGNN over the best compared work for the corresponding dataset.

Model	Data	DGL	GNNA.	FeatG.	TLPGNN	Speedup
GCN	CS	0.4	0.35	0.05	**0.026**	1.9×
	CR	0.39	0.37	0.1	**0.028**	3.6×
	PD	0.35	0.39	0.17	**0.033**	5.2×
	OA	**0.69**	1.1	5.1	1.2	0.6×
	PI	0.69	0.63	1.1	**0.14**	4.5×
	DD	2.2	1.2	0.45	**0.3**	1.5×
	OH	6.6	3.6	1.4	**1.2**	1.2×
	CL	4.2	-	5.3	**1.1**	3.8×
	ON	13.1	-	27.2	**4.2**	3.1×
	RD	25.4	-	42.3	**6.9**	3.7×
	OT	41.3	-	37.4	**16.1**	2.3×
GIN	CS	0.25	0.38	0.04	**0.035**	1.14×
	CR	0.27	0.42	0.05	**0.026**	1.9×
	PD	0.25	0.45	0.07	**0.032**	2.2×
	OA	**0.77**	1.1	1.4	0.97	0.79×
	PI	0.55	0.8	0.5	**0.13**	3.8×
	DD	1.1	1.2	0.37	**0.24**	1.5×
	OH	3.9	3.4	1.6	**1.1**	1.5×
	CL	4.5	-	2.8	**0.98**	2.9×
	ON	13.1	-	18.9	**4.2**	3.1×
	RD	19.5	-	17.3	**7.3**	2.4×
	OT	31.5	-	23.4	**16.4**	1.4×
Sage	CS	0.47	-	0.07	**0.068**	1.03×
	CR	0.44	-	0.11	**0.034**	3.2×
	PD	0.38	-	0.21	**0.039**	5.4×
	OA	**0.86**	-	5.3	1.02	0.84×
	PI	0.58	-	1.1	**0.16**	3.6×
	DD	1.2	-	0.72	**0.31**	2.3×
	OH	4.6	-	2.9	**1.3**	2.2×
	CL	4.6	-	5.6	**1.0**	4.6×
	ON	13.6	-	27.3	**4.1**	3.3×
	RD	21.4	-	41.9	**7.3**	2.9×
	OT	32.6	-	38.9	**17.2**	1.9×
GAT	CS	0.69	-	0.07	**0.044**	1.6×
	CR	0.71	-	0.09	**0.059**	1.5×
	PD	0.68	-	0.19	**0.061**	3.1×
	OA	4.8	-	5.8	**3.7**	1.3×
	PI	1.3	-	1.5	**0.34**	3.8×
	DD	1.3	-	0.61	**0.41**	1.5×
	OH	3.4	-	1.9	**1.6**	1.2×
	CL	10.3	-	7.7	**2.4**	3.2×
	ON	62.3	-	42.3	**9.1**	4.6×
	RD	102.2	-	56.6	**14.6**	3.9×
	OT	112.1	-	43.3	**28.4**	1.5×

8.5.0, and PyTorch 1.8.1. We report the results on the averages of ten runs.

7.2 Performance Comparison

This section compares the performance of TLPGNN with three recent GNN computation methods. We measure the execution time of the single operation of graph convolution for each GNN model, which is the focus of this study. We compare with GNNAdvisor for GCN and GIN models as other models are not implemented. In addition, GNNAdvisor faced illegal CUDA memory access for the four largest graphs. For DGL and FeatGraph, we compared with them for the four GNN models on all the graphs. The feature size is set to 32.

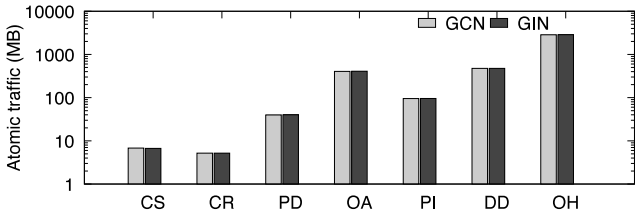

Figure 8: The memory traffic (MB) of the atomic writes in GNNAdvisor for GCN and GIN models over 7 datasets.

Table 5 shows the execution times of TLPGNN and compared works. The table also shows the speedups of TLPGNN over the best baseline for the same dataset. On average, TLPGNN achieves 7.7× (up to 13.5×) speedup over GNNAdvisor, 5.6× (up to 13.9×) over DGL, and 3.3× (up to 7.5×) over FeatGraph. For the four graph datasets that have larger edge numbers and average degrees, TLPGNN achieves an average of 3.7× speedup compared with DGL, and and average of 4.1× speedup compared with Featgraph. For the first three models (i.e., GCN, GIN, and GraphSage) whose graph convolution operations are relatively simply constructed, TLPGNN achieves 5.8× for GCN, 4.6× for GIN, and 4.7× for GraphSage. And for the most complicated model GAT, TLPGNN outperforms the other two baselines by 6.5×and 2.6×. Because of its complexity, DGL uses 18 GPU kernels and FeatGraph uses 3, while TLPGNN keeps the kernel number to one using our kernel fusion technique.

DGL's implementations rely on general sparse-dense matrix multiplication (SpMM) kernels provided by the vendor-shipped library cuSPARSE [36]. First, DGL needs extra kernels to manipulate the data formats of input in order to invoke the SpMM kernels [44]. Second, for complex GNN models such as GAT, a single SpMM kernel is not enough to express the computation of the model's graph convolution, so DGL uses more kernels to compose it. These two factors lead to the high number of kernel launches of DGL. Specifically, DGL launches 6, 8, 10, and 18 GPU kernels for GCN, GIN, GraphSage, and GAT, respectively. The unnecessary kernel launches lead to sub-optimal performance of DGL. It is worth noting that DGL outperforms TLPGNN for GCN, GIN, and GraphSage

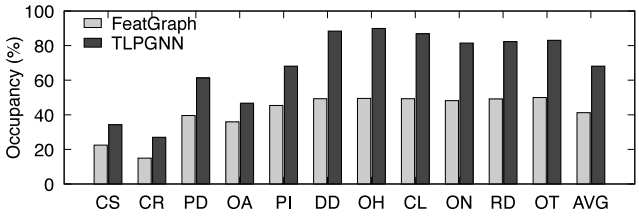

Figure 9: The achieved occupancy of GCN implementation of FeatGraph and TLPGNN over all graph dataset (with average value).

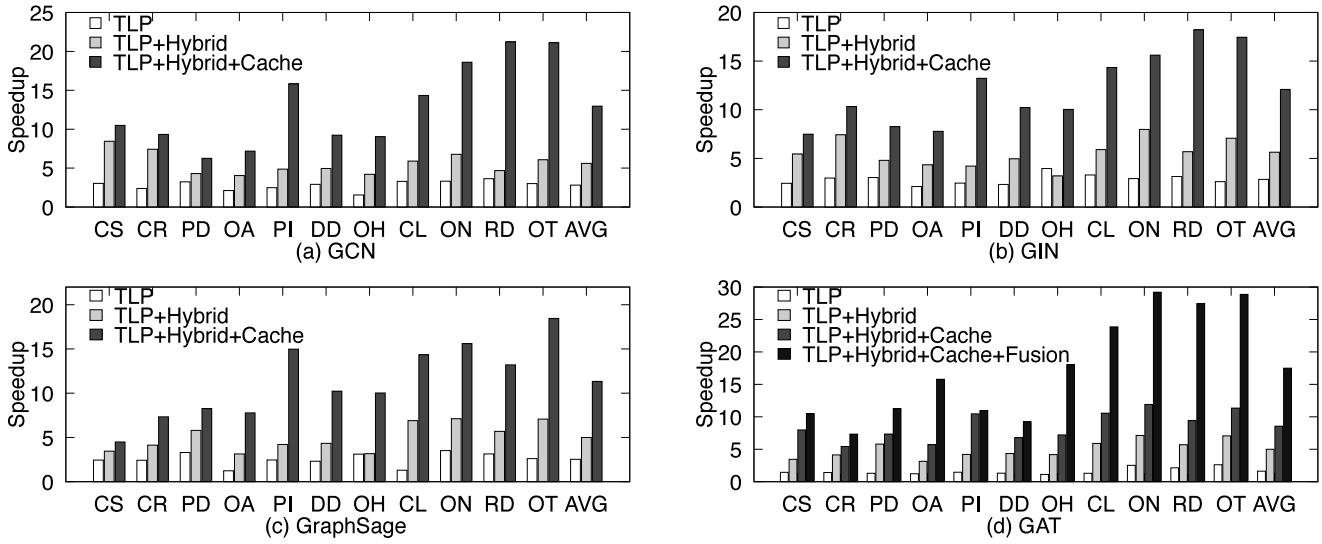

Figure 10: Benefits of the newly designed techniques for (a) GCN, (b) GIN, (c) GraphSage, and (d) GAT.

models over the OA dataset, due to the excellent performance of the SpMM kernel on the dataset according to our profiling.

GNNAdvisor partitions each vertex's neighbor list into multiple fix-sized groups and assign each group to one warp for processing. As a result, each warp uses atomic operations to update the intermediate result into the final result of the vertex. Figure 8 shows the memory traffics of atomic writes for GNNAdivsor. As a contrast, the two-level parallelism design allows TLPGNN to avoid any atomic operations, which brings the performance improvement over GNNAdvisor. In addition, GNNAdvisor is suffering from two types of pre-processing overhead: vertex reordering and partition building [46]. Unlike many existing works [46, 50], TLPGNN doesn't require any data pre-process.

FeatGraph makes use of TVM [6], a deep learning compiler framework, to generate efficient GPU kernels for GNN operations. One can notice it outperforms the other two baselines for most experiment settings. However, TVM's Tensor Expression API is not flexible to manage the workload mapping between vertices and CUDA threads, leading to low hardware utilization and occupancy. As shown in Figure 9, the average achieved occupancy of FeatGraph's GCN implementation for all datasets is 41.2%, in comparison with TLPGNN's average value of 68.2%.

7.3 Technique Benefits

Figure 10 presents the speedups of using our design and optimization techniques over the baseline method, which is an edge-centric process implementation. For GCN, GIN and GraphSage, we measure the impact of our two-level parallelism (TLP), hybrid dynamic workload assignment (Hybrid), and register caching (Cache). And we also show the impact of kernel fusion (Fusion) for GAT model. Our two-level parallelism design achieves 2.82×, 2.84×, 2.53×, and 1.63× speedups for the four GNN models, respectively. Only with the two-level parallelism, the implementation is still suffering from uneven workload distribution and unnecessary memory traffic. But we can still see considerable improvement due to the exemption from atomic operations. The hybrid dynamic workload assignment brings 1.99×, 1.98×, 1.97×, and 3.1× speedups on the basis of the previous setting. The speedups from this technique over large graphs are more significant than small ones because the issue of workload imbalance is more severe in large graphs.

Register Caching is able to improve the performance by 2.3×, 2.15×, 2.26×, and 1.71× for GCN, GIN, GraphSage, and GAT. Coupled with the previous two techniques, we can achieve speedups of 12.9×, 12.1×, 11.3×, and 8.6× averagely for all different GNN models. One can notice that this technique performs much better on graphs with large average degree numbers (i.e., PI, CL, ON, RD, and OT), which is reasonable because a large degree means more unnecessary global memory write without register caching. In addition, the kernel fusion also brings speedup of 2.01× to GAT model. To sum up, each individual technique plays an indispensable role in our design to address a specific performance issue. When combined together, they give us significant performance improvement.

7.4 Scalability

In this section, we demonstrate the scalability of TLPGNN from two perspectives: thread count and feature size.

Scalability against thread count. To show the scalability against the increasing of thread count, we increase the number of blocks from 1 to 128 by launching 512 threads per block. Figure 11 presents the speedups over the single block. We can observe that TLPGNN achieves a stable linear scalability when increasing the number of computation threads. Compared with the runtime of 512 threads, TLPGNN with 128∗512 threads is able to get 67.5×, 62.5×, 67.2×, and 45.3× speedup for GCN, GIN, GraphSage, and GAT, respectively, on the average.

Scalability against feature size. Further, we evaluate the runtime of TLPGNN when increasing the input feature size from 16 to

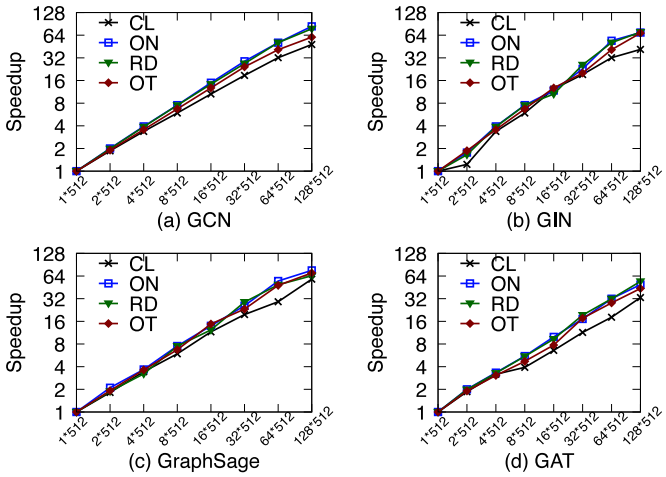

Figure 11: The scalability of TLPGNN against the number of threads (shown in x-axis) for the four largest graphs (CL, ON, RD, OT). (a) GCN, (b) GIN, (c) GraphSage, and (d) GAT.

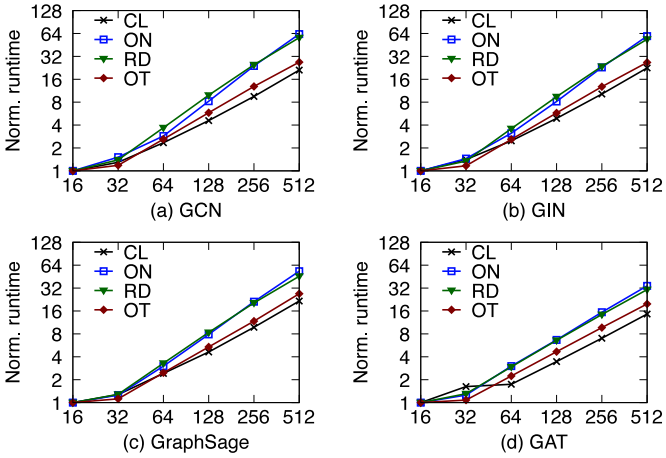

Figure 12: The scalability (normalized runtime) of TLPGNN against feature sizes (shown in x-axis) for the four largest graphs (CL, ON, RD, OT). (a) GCN, (b) GIN, (c) GraphSage, and (d) GAT.

512. Figure 12 presents the normalized runtime by dividing the runtime of different feature size over feature size 16. We also observe that TLPGNN achieves a linear correlation between the runtime and the feature sizes. When processing a feature size of 512 (32 times over the smallest one, i.e., 16), TLPGNN is 41.6×, 40.4×, 36.7×, and 27.3× slower than the feature size of 16. In addition, we find the runtime of for the implementation with size 16 are 1.4×, 1.4×, 1.3×, and 1.4× faster than with size 32 for the four different models, respectively. This actually demonstrates that even with half of the threads in a warp idle, the performance of TLPGNN is not affected much.

8 RELATED WORKS

In this section, we discuss the related works of TLPGNN in terms of GNN systems, and traditional graph processing systems.

GNN systems. The last few years have seen an increased research interest in building systems attempting to make GNN execution more efficient in various hardware platforms [7, 11, 18, 29, 42, 44, 46, 50]. NeuGraph [29] is a distributed training framework built on top of TensorFlow [1] to address the challenge of GNN training on extremely huge graphs. Deep Graph Library (DGL) [44] and PyTorch-Geometrics [10] introduce a message-passing interface onto the popular deep learning system for easy GNN programming and execution. GNNAdvisor [46] proposes a runtime system to accelerate GNN workload on a single GPU, by taking advantage of the performance-relevant features of the model and input data. It also employs techniques of workload management and memory hierarchy mapping tailored for GNN to achieve better GPU hardware utilization. SeaStar [50] presents an intuitive vertex-centric programming interface for GNN programming and a couple of optimizations to generate efficient GPU kernels for GNN execution. FeatGraph [18] makes use of deep learning compiler TVM [6] to emit sparse matrix kernel for GNN computation on CPU and GPU. In this work, we focus on the performance of graph-related operations in GNN computation, conduct an in-depth profile and analysis in terms of several important performance factors. Based on the observations from the analysis, our TLPGNN uses a novel design coupled with optimization techniques to booster the performance of GNN workloads.

Graph processing systems. The growing importance of graph data has driven the development of numerous graph processing systems both on CPU and GPU platforms [20, 21, 24, 27, 30, 34, 41, 45]. Generally speaking, most of them are written as C++ template libraries, offering vertex-centric [32] or edge-centric APIs which can be utilized to generate highly optimized kernels for various graph algorithms. Gunrock [45] offers a set of flexible APIs to express graph operation primitives, uses GPU-tailored optimizations for memory efficiency and workload balance to achieve high performance. SIMD-X [27] utilizes just-in-time task management and push-pull-based kernel fusion to enable balanced workload assignment and deliver better performance. However, traditional graph processing systems are targeting workloads in which vertices and edges are associated with scalar features, ignoring the additional dimension of long feature size [18]. This makes those systems not suitable for GNN workloads. TLPGNN's design pays lots of attention to the additional feature dimension in GNNs and proposes optimizations to address the performance issue brought by the new factor, considering the feature parallelism.

9 CONCLUSION

In this paper, we present TLPGNN, a high-performance two-level parallelism design for efficient GNN computation on GPU. First, we conduct an in-depth analysis on several important perspectives related to the performance of GNN workloads: atomic operations, coalesced memory access, and kernel numbers. We obtain several insightful observations from our profiling and analysis. Based on the observations, we design a two-level parallelism paradigm: the

first level makes use of vertex-parallelism to avoid atomic operation and unnecessary memory traffics; the second level exploits feature-parallelism for coalesced memory access. To address the workload imbalance caused by vertex-parallelism, we propose a hybrid dynamic workload assignment to choose the best workload balancing method according to the characterizes of the input graph. The experiment results show that TLPGNN achieves significant speedups over existing solutions for GNN computation.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback and suggestions. This work was supported in part by DARPA under agreement number N66001-18-C-4033 and National Science Foundation grants 1618706, 1717774, and 2127207. The views, opinions, and/or findings expressed in this material are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense, National Science Foundation, or the U.S. Government.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Siddhant Arora. 2020. A survey on graph neural networks for knowledge graph completion. *arXiv preprint arXiv:2007.12374* (2020).

[3] Alaa Bessadok, Mohamed Ali Mahjoub, and Islem Rekik. 2021. Graph Neural Networks in Network Neuroscience. *arXiv preprint arXiv:2106.03535* (2021).

[4] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 93–104.

[5] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. 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*. 257–266.

[8] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2020. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982* (2020).

[9] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The World Wide Web Conference*. 417–426.

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[11] Qiang Fu and H Howie Huang. 2021. Automatic Generation of High-Performance Inference Kernels for Graph Neural Networks on Multi-Core Systems. In *50th International Conference on Parallel Processing*. 1–11.

[12] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. 2020. Magnn: Metapath aggregated graph neural network for heterogeneous graph embedding. In *Proceedings of The Web Conference 2020*. 2331–2341.

[13] Victor Fung, Jiaxin Zhang, Eric Juarez, and Bobby G Sumpter. 2021. Benchmarking graph neural networks for materials chemistry. *npj Computational Materials* 7, 1 (2021), 1–8.

[14] Yang Gao, Yi-Fan Li, Yu Lin, Hang Gao, and Latifur Khan. 2020. Deep learning on knowledge graph for recommender system: A survey. *arXiv preprint arXiv:2004.00387* (2020).

[15] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[16] Dichao Hu. 2019. An introductory survey on attention mechanisms in NLP problems. In *Proceedings of SAI Intelligent Systems Conference*. Springer, 432–448.

[17] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687* (2020).

[18] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. *arXiv preprint arXiv:2008.11359* (2020).

[19] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 119–132.

[20] Yuede Ji and H Howie Huang. 2020. Aquila: Adaptive parallel computation of graph connectivity queries. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 149–160.

[21] Yuede Ji, Hang Liu, and H Howie Huang. 2018. ispan: Parallel identification of strongly connected components with spanning trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 731–742.

[22] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[23] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[24] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: Vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

[25] Jongmin Kim, Taesup Kim, Sungwoong Kim, and Chang D Yoo. 2019. Edge-labeling graph neural network for few-shot learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 11–20.

[26] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[27] Hang Liu and H Howie Huang. 2019. Simd-x: Programming and processing of graph algorithms on gpus. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 411–428.

[28] Qingsong Lv, Ming Ding, Qiang Liu, Yuxiang Chen, Wenzheng Feng, Siming He, Chang Zhou, Jianguo Jiang, Yuxiao Dong, and Jie Tang. 2021. Are we really making much progress? Revisiting, benchmarking, and refining heterogeneous graph neural networks. (2021).

[29] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 443–458.

[30] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

[31] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. 2000. Automating the construction of internet portals with machine learning. *Information Retrieval* 3, 2 (2000), 127–163.

[32] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR)* 48, 2 (2015), 1–39.

[33] Seth A Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. 2014. Information network or social network? The structure of the Twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*. 493–498.

[34] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. 456–471.

[35] Nvidia. [n.d.]. Cuda C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications__technical-specifications-per-compute-capability

[36] NVIDIA. 2021. cuSPARSE. https://developer.nvidia.com/cusparse

[37] Nvidia. 2021. Nvidia Nsight Compute. https://developer.nvidia.com/nsight-compute

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[39] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[40] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[41] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[42] Chao Tian, Lingxiao Ma, Zhi Yang, and Yafei Dai. 2020. Pcgcn: Partition-centric processing for accelerating graph convolutional network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 936–945.

[43] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[44] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[45] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 1–12.

[46] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2020. GNNAdvisor: An Efficient Runtime System for GNN Acceleration on GPUs. *arXiv preprint arXiv:2006.06608* (2020).

[47] Wikipedia contributors. 2021. Biological network — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Biological_network&oldid=1039989954. [Online; accessed 25-August-2021].

[48] Wikipedia contributors. 2021. Graph (discrete mathematics) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Graph_(discrete_mathematics)&oldid=1017809268. [Online; accessed 25-August-2021].

[49] Wikipedia contributors. 2021. Molecular graph — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Molecular_graph&oldid=1032100381. [Online; accessed 25-August-2021].

[50] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 359–375.

[51] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[52] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 974–983.

[53] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems* 31 (2018), 5165–5175.