THE UNIVERSITY OF CHICAGO

ACCELERATING GNN AGGREGATION USING A VERTEX-CENTRIC APPROACH

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
RUIQI XU

CHICAGO, ILLINOIS
GRADUATION DATE

Dedication Text

Epigraph Text

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to express my gratitude to my advisor, Dr. Andrew A. Chien, for his unwavering guidance, patience, and expertise throughout this process. His insights have been invaluable to the development and completion of this work.

I am also thankful for our collaborators Dr. Roman Zubatyuk and Dr. Olexandr Isayev from Carnegie Mellon University as well as Ruize (Richard) Xu and Dr. Risi Kondor from the University of Chicago.

To my colleagues and peers in the UpDown Project, thank you for fostering a collaborative and motivating environment. Your help and feedback are integral to this work.

A special thanks to my family and friends for their encouragement, love, and understanding. Your belief in me has been my greatest source of strength throughout this journey.

ABSTRACT

While Graph Neural Networks (GNNs) serve as an important tool for learning graph-structured data, it presents new challenges to existing architectures. As opposed to being dominated by regular tensor operations with high arithmetic intensity, GNN introduces message passing, which allows vertices to exchange information with their neighbors, producing irregular and data-dependent behaviors. These characteristics make it difficult to express the algorithms and optimize their performance.

We demonstrate that UpDown, with its fine-grained MIMD architecture, enables a vertex-centric programming approach that suits the pairwise messaging model and maximizes data reuse. We also present task splitting for addressing degree imbalance and remapping for capturing data reuse. GPUs, by comparison, are limited by their primitives which fail to effectively express message passing, leading to unnecessary materializations of intermediate results and high performance overhead. We evaluate the performance of the UpDown architecture on GCN, GIN, and AIMNet, a state-of-the-art chemistry model. We show that compared to the H100 GPU, an UpDown system with an equal issue rate can achieve up to 45.62x speedup on aggregation for common models like GCN and 58.08x on AIMNet, which translates into a 6.05x and 5.68x improvement on the overall forward pass for the respective models.

CHAPTER 1
INTRODUCTION

In recent years, deep learning has experienced great success in a wide range of tasks including image recognition[18], language understanding[11], and speech recognition[17]. Graph neural networks extend this to non-Euclidean data, enabling processing and learning on graphs[39]. It has demonstrated performance advantages on tasks like node classification[23, 16], link prediction[47], and graph classification[44].

However, unlike many neural networks, GNNs have a heterogeneous workload. The operations can be largely separated into two stages: combination and aggregation[41]. The combination phase behaves similarly to other types of neural networks, where the feature vectors of vertices are transformed using one or multiple linear layers (MLP). Optimizations of these dense computations have been well-studied, with numerous famous solutions like TPUs[20] and GPUs. The aggregation phase is more dynamic, in which each source node creates a message to the destination node by applying some operations to its feature. The messages to the same destination node will be reduced using operators like sum and max. Due to the irregular nature of graphs, this produces traffic patterns that are hard to optimize with cache. Prior works have demonstrated that the aggregation phase typically dominates the execution time on GPUs[41], making it the target for accelerating GNN computation.

In the PyTorch Geometric library[12], aggregation is implemented in the following fashion on GPUs. First, it materializes a matrix containing the features of the source vertex for all the edges. Then, it manipulates the matrix based on the message function of the GNN model. The messages are aggregated using a gather function. While these operations are regular and therefore suitable for GPUs, they unnecessarily materialize the intermediate values, wasting instructions and memory traffic while creating a large memory footprint. The vertex-centric execution model, by contrast, avoids these issues by manipulating and consuming the messages locally. However, the most common programming primitives of

GPUs are not compatible with this approach. The Deep Graph Library (DGL)[36] is an attempt at this direction: it fuses the message generation and aggregation steps using a custom SpMM kernel. While it avoids materializing the messages, it still demands significant memory traffic due to the graph structure. To balance generality and performance, some operations also cannot be effectively fused.

The UpDown architecture provides an alternative solution. Its MIMD design naturally enables independent streams of vertex operations, allowing us to implement GNNs with vertex-centric programs and remove the overheads associated with intermediate values. We believe that this approach makes it easier to express and extend the models. Apart from that, the UpDown system provides abundant DRAM bandwidth, which is key to handling the irregular memory accesses required by the aggregation phase. It uses an event-driven model with fine-grain threads, effectively hiding the memory latency while generating sufficient memory parallelism to saturate the bandwidth. Combined with the do-all library for mapping and managing the vertex tasks, UpDown provides a clear path for running GNNs at a large scale. Furthermore, we demonstrate that UpDown provides greater flexibility in the choice of data structures: we can easily incorporate edge embeddings to implement more advanced GNNs.

In this paper, we study the aggregation phase of GNNs under a vertex-centric approach on UpDown. We demonstrate how to implement the operations under the event-driven model and evaluate the performance. We also discuss some bottlenecks on UpDown and propose ways to alleviate them. Additionally, we analyze common implementations on GPUs and compare their performance with UpDown.

1.1 Thesis Statement

UpDown's fine-grained MIMD architecture enables a vertex-centric approach for GNN aggregation, enabling it to achieve up to 60x performance compared to GPUs. The approach enables key benefits of code optimization, increased data reuse, flexible data structure, and algorithm extension.

1.2 Approach

We express the GNN aggregation computation as vertex-centric programs. The users can perform queries to obtain the vertex and neighbor information, generate the messages from neighbor and edge features on the fly, and merge the messages into the scratchpad. We store the graph using CSR-style neighbor lists that match the program access pattern. To efficiently spread the tasks across a large number of compute units, we rely on the UDKVMSR framework[37]. It receives an array of vertex IDs, distributes the entries to worker lanes, and invokes user-defined map tasks. We demonstrate that this baseline design already provides robust performance and saturates the memory bandwidth. For graphs with skewed vertex degrees, we design a splitting method that redistributes the tasks, proving the scalability to larger setups.

Furthermore, while aggregation typically produces irregular memory access patterns, it is possible to capture reuse opportunities using dataset characteristics or graph partitioning techniques. We demonstrate that the UDKVMSR framework can be adapted to enable fine-grained control of task mapping. Combined with our software-managed cache, we can reduce memory traffic significantly, boosting performance due to the memory-bound nature of the task.

1.3 Contributions

This thesis mainly discusses the performance challenges the GNN aggregation phase produces. It introduces an effective vertex-centric implementation on UpDown and compares the results with GPUs, the most popular form of machine learning accelerators. It also discusses possible optimizations based on the observed outcomes.

Specific contributions of this work include the following:

- Propose a vertex-centric strategy for implementing GNN aggregation on UpDown based on its architectural characteristics and software infrastructure.
- Demonstrate UpDown's performance advantages and programmability on common GNN layers and state-of-the-art GNN models.
- Evaluate the bottlenecks and implement optimizations like task splitting and remapping to improve throughput and scalability.
- Compare UpDown with GPUs, analyze the sources of performance gain, and evaluate the impact of accelerating aggregation in the application context.

1.4 Thesis Outline

The structure of this thesis is organized as follows. Chapter 2 provides the background information. In chapter 3, we introduce the UpDown hardware and software features. Chapter 4 highlights our approach for implementing and optimizing GNN aggregation operations on UpDown. Chapter 5 provides an evaluation of our approach and compares UpDown to GPUs. In chapter 6, we discuss the related work. Lastly, chapter 7 offers a summary and some directions for future work.

CHAPTER 2
BACKGROUND

2.1 Graph Neural Network

Most of the popular GNN designs fall under the message-passing framework. The source vertex will receive a message from all its neighbors based on their feature vectors and the edge information. Then the messages are reduced and transformed to become the input to the next layer or the output of the network.

Notation	**Meaning**
G	Graph, $G = (V, E)$
V	Vertices of G
E	Edges of G
n_v	Number of vertices in G
n_e	Number of edges in G
$N(i)$	Neighbors of vertex V_i
$D(i)$	Degree of vertex V_i
x_i^l	Feature of vertex V_i at layer l
$e_{i,j}^l$	Feature of edge E between vertex i and vertex j at layer l

Table 2.1: GNN Algorithm Notations

This computation can be expressed as follows[31]:

$$x_i^{l+1} = \text{Combine}\Big(x_i^l, \text{Aggregate}_{j \in N(i)}\big(\text{Message}(x_i^l, x_j^l, e_{i,j}^l)\big)\Big) \tag{2.1}$$

Take the graph convolution network (GCN) as an example, its message function would be scaling the neighbor feature by $\frac{1}{\sqrt{D_i * D_j}}$, its aggregate function is summation, and its combine function is a linear transformation. For the graph isomorphism network (GIN), the message function is simply taking the neighbor's feature and the aggregator is summation. The combine function adds a scaled target node embedding to the message and applies a MLP on the result.

In addition to these models, we also study AIMNet[49], which is a state-of-the-art model for chemical property prediction. It has edge embeddings with more sophisticated message functions. However, it still follows the same formulation.

Under this general framework, we make the observation that while combine is similar to most existing deep neural networks, requiring dense and regular matrix operations, aggregate and message produce data-dependent operations that are irregular. While the former is well-studied, with GPUs being a successful solution given their abilities to offer massive SIMD parallelism and take advantage of the reuse opportunities inherent in the algorithm, the latter presents some challenges. First, the nature of the graph means the message can be sent to any vertex in the graph. Whether we use the scatter or the gather pattern to handle this operation, the access pattern would be hard to optimize with cache. That implies we need a high number of threads to hide the DRAM latency. Second, the message and aggregate operations usually have low arithmetic intensity. That means we need high memory bandwidth to saturate the compute.

2.2 Graphic Processing Units

GPUs have played an important role in the development of machine learning. They use a SIMD architecture: a set of processing units that execute the same instruction on different data. It simplifies the GPU core design, enabling GPUs to provide massive parallelism.

The CUDA platform provides a programming interface for GPUs. Typical CUDA programs contain the following elements. The individual CUDA cores execute instructions in the thread. To share data between threads, they are assigned to the same thread block so that they run on the same multiprocessor. A grid of thread blocks is used to execute a CUDA kernel[15].

To achieve optimal performance, CUDA programs have to comply with a few constraints. First, because CUDA cores execute the same instruction, control divergence will cause parts of the cores to stall. For example, if some threads enter an if condition, the remaining threads have to wait for them to finish before continuing the program. Therefore, divergence typically prevents GPUs from hitting their peak throughput, making irregular computation expensive for them. Second, to minimize latency and avoid wasting bandwidth, the threads have to access a contiguous chunk of memory (typically 256 bytes) so that the access is not split into multiple operations. These requirements make it challenging to implement programs with dynamic behaviors. Additionally, factors like insufficient threads per SM, suboptimal block dimensions, shared memory bank conflict, thread launch overhead, and synchronization costs all impact kernel performance. Thus, most GPU programmers rely on optimized software like cuBLAS[1] and cuDNN[2], which are integrated into frameworks like PyTorch[27] and TensorFlow[4].

2.3 UpDown

UpDown is proposed as a solution to accelerate irregular graph applications. Compared to existing architectures, it has the following key characteristics. First, the accelerators provide high MIMD parallelism. One UpDown node contains 32 accelerators with 64 compute lanes that can execute independent instruction streams. With SIMD instructions, it provides FP32 throughput up to 16 TFLOPS. Second, the accelerators are directly connected to the DRAM, meaning that the memory accesses no longer go through the cache hierarchy. One compute node is equipped with 8 stacks of HBM3e memory with 8.8 TB/s of bandwidth, which surpasses most of the existing GPU designs. Third, UpDown uses event-driven programming with split-transaction DRAM access. This design supports far greater memory-level parallelism than traditional architectures, enabling UpDown to easily saturate the abundant bandwidth[30]. Additionally, each UpDown lane is equipped with 128 lightweight threads, which allows for fine-grained parallelism that helps hide DRAM latency.

The software infrastructure of UpDown is also designed for graph processing. The UD-

KVMSR framework follows the MapReduce model[10], distributing tasks to compute units through key/value pairs. The programmers can define the map function invoked for the KV pairs and tune the parameters for parallelism to achieve optimal performance while the library takes care of mapping and management. Graphs are represented using the Parallel Graph Abstraction (PGA) data structure. It stores the edge and vertex attributes using UpDown's Scalable Hash Table (SHT), which enables lookup through the object ID. The neighbors of the vertices are stored as a list of (neighbor ID, edge ID) pairs. PGA enables multiple tiers of edge list lengths so that we do not have to allocate memory based on the longest list. More importantly, these two libraries can work together: UDKVMSR can launch map tasks for each vertex or edge in the graph storage, greatly simplifying the programmer's task.

2.4 GPU Libraries for GNN

There are two commonly used GNN libraries for implementing GNNs on GPUs: PyTorch Geometric (PyG)[12] and Deep Graph Library (DGL)[36]. We evaluate the effectiveness of both in this work.

PyG represents the graph as a $(2, n_e)$ array. The first row represents the indices of the source vertices and the second row represents that of the destination vertices. A separate (n_v, #Features) matrix is used to store all the features.

Figure 2.1 illustrates the PyG implementation of GNN aggregation which we use as a comparison in this work. First, it expands the vertex feature matrix into a per-edge message matrix by performing index selection on the source indices of the edges. Then, the message generation phase is implemented by manipulating the matrices, usually scaling the rows in the matrix by some factors or performing a GEMM operation. After that, the messages are reduced through scatter add, where the message vectors are merged based on the destination. This step will produce a (n_v, #Features) matrix. While the aggregation

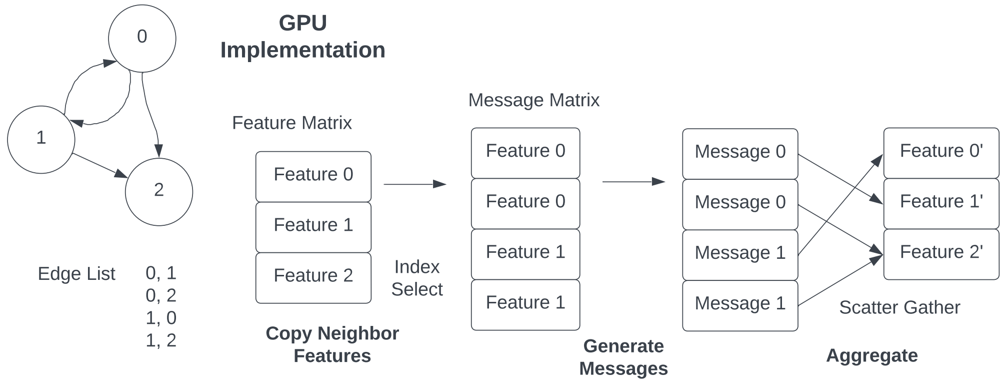


Figure 2.1: PyTorch Geometric Implementation of GNN Aggregation

step happens before the update step, for some models where the ordering does not affect the output like GCNConv, PyG reorders the two operations based on the observation that most update operations reduce the dimensions of the feature vectors, leading to fewer operations overall.

The PyG GPU implementation suffers from the following issues. The formulation of message passing as a dense matrix operation creates unnecessary data, memory traffic, and computation. Even without any manipulations to the message matrix, there is a minimal 2x increase in memory traffic compared to a vertex-centric approach (write back in index select and read in scatter add). The pointer computations and load instructions will increase accordingly. Meanwhile, since scatter-add creates updates to data-dependent destinations, the memory access pattern is hard to optimize with caches, producing greater redundant traffic.

AIMNET uses a custom implementation based on PyG which avoids some of these shortcomings. Instead of using the edge list, they adopt a neighbor matrix representation with a $(n_v + 1, \max(\text{degree}))$ matrix. Each row stores the neighbors of a vertex, with the extra entry padded by a dummy vertex. For their target datasets, this idea is feasible because there is limited degree skewness in their chemical graphs. While they still need to use index select to materialize a $(n_v * \text{max degree}, \#\text{features})$ matrix, they can perform the message

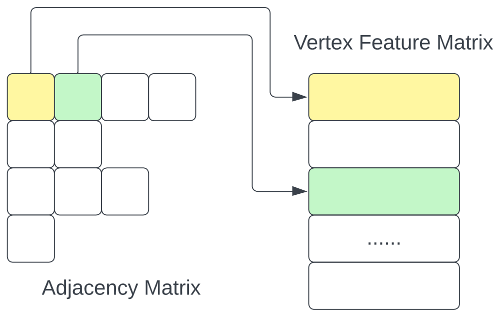


Figure 2.2: Deep Graph Library Implementation of GNN Aggregation

generation and aggregation steps using the optimized PyTorch Einstein sum function with good write locality.

DGL adopts a different approach which they call "fused kernel". They represent the graph as a CSR matrix and also store the vertex features in a dense array. They treat aggregation as a generalized SpMM operation and implement a custom kernel based on Gustavson's algorithm. As shown by Fig 2.2, each row in the adjacency matrix represents one destination vertex. The kernel will read all the nonzero entries, fetch the corresponding row from the feature matrix, and reduce them. The users can choose between a set of predefined message and aggregate operations, and the library will generate the fused CUDA kernel. Compared to the PyG approach, they avoid materializing the messages, leading to lower memory footprints.

2.5 Graph Relabeling and Partitioning

As pointed out by multiple works on GNNs[5], the aggregation phase produces irregular access patterns as it is graph structure dependent. At the same time, its operations have low arithmetic intensity, making the aggregation phase memory-bound. To alleviate that problem, we consider task remapping based on graph reordering and partitioning techniques to maximize the reuse opportunities of neighbor features for vertices assigned to the same processing unit.

The Cuthill-Mckee algorithm[9] was proposed as a way to minimize the distances of nonzeros to the diagonal in sparse matrices by relabeling the vertices. It performs a modified breath-first search, traversing the vertices in orders based on their predecessors and degrees. When applied to the adjacency matrices, the bandwidth reduction causes the vertices with a similar set of neighbors to have closer IDs, increasing the reuse opportunities when we perform aggregation. However, in practice, we find that it tends to cluster high-degree vertices toward the end, producing load imbalance.

The approach we end up using is METIS[21], a software designed for graph partitioning. The goal of METIS is to produce partitions that result in minimal cross-partition edges while balancing the number of vertices or edges in each partition for optimal parallel computing performance. The algorithm first transforms the original input into smaller graphs by merging connected vertices. Partitioning is performed on the smallest graph to reduce time consumption. Then, the graph is expanded step-by-step to its original size, with refinements to the partitions at each step. It is useful because when we assign the partitions to the compute units, the number of unique neighbors accessed decreases, making it easier to cache and reuse the data.

CHAPTER 3
UPDOWN ARCHITECTURE

In this chapter, we highlight the novel hardware features of UpDown, discuss the programming model, and introduce the libraries for storing graph data and managing parallel tasks.

3.1 UpDown Hardware

We base our design on the UpDown architecture[29] developed for the IARPA AGILE project, which aims to build novel systems for accelerating irregular graph applications. The setup we study is a small-scale design that matches the issue rate of the H100 GPU, meaning that it contains 13,516 compute lanes running at 2.0 GHz. The accelerators will be attached to 8 stacks of HBM3e memory, providing 8.8 TB/s total bandwidth. We will highlight some of its features in the sections below.

3.1.1 MIMD Design

UpDown adopts a MIMD design: each accelerator contains 64 in-order compute lanes that can execute independent instruction streams. Unlike CPUs, the absence of cache and out-of-order execution allows it to provide high parallelism. Compared to GPU's SIMD approach, UpDown offers greater programming flexibility, unlocking greater freedom for algorithms and software implementation.

3.1.2 Event-Driven Model with Global Communication

UpDown uses an event-driven execution model. Each lane is equipped with an event queue. The lanes will fetch entries from it and run the corresponding code segment based on the event label. One key feature of UpDown is the ability of lanes to perform global communication. Each lane in the system is assigned a network ID. They can trigger operations on any other lane by sending events. This simplifies parallel task management and enables flexible task mapping.

3.1.3 *Split-transaction Operations*

UpDown adopts a split-transaction design for memory accesses and library calls. The programmer issues the query and defines the event to be triggered when the data returns. That allows the compute lane to work on other tasks while that thread waits for necessary data.

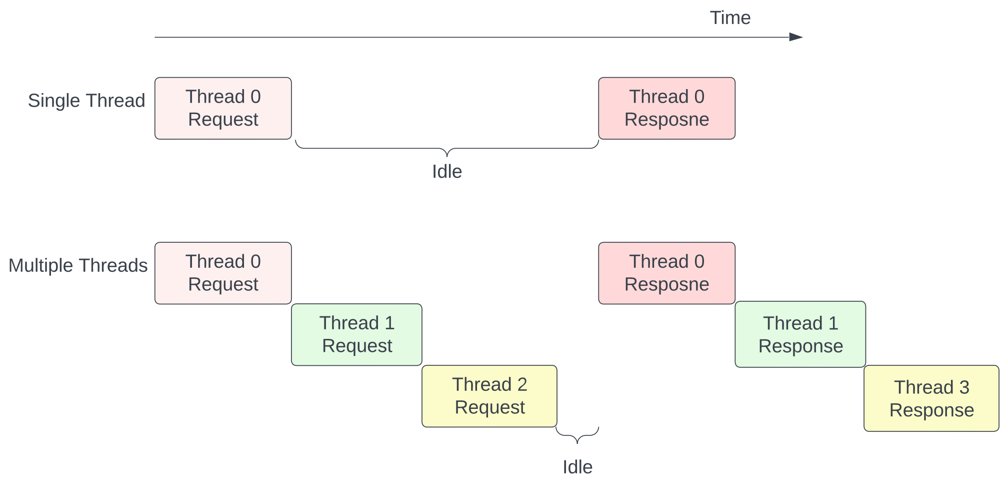


Figure 3.1: Effective of Threads in UpDown Programs

That design is backed by UpDown's use of lightweight threads. Every UpDown lane is equipped with 128 hardware threads, and they are selected by the thread ID encoded in the event word. This allows multiple tasks to share a compute lane. As shown by Fig 3.1, by having multiple threads on the same lane, the time spent waiting for the requests of the first thread to return can be effectively covered by issuing requests for the second and third threads, significantly reducing the idle time on the lanes. Thanks to the low 1-cycle context-switching cost, even fine-grained short invocations can achieve high efficiency, enabling high compute utilization by breaking down operations into smaller pieces that can be scheduled immediately when the data becomes available. Furthermore, as pointed out by our prior work[30], having more threads also helps increase the number of outstanding requests, which generates the memory parallelism necessary to saturate HBM bandwidth.

3.1.4 Software-managed Scratchpad

Unlike CPUs and GPUs which use cache to capitalize reuse, UpDown targets graph applications with irregular access patterns. Therefore, it uses a software-managed scratchpad for local memory. Each lane is equipped with 64 KB of scratchpad memory and can access the banks of other lanes on the same accelerator (4MB in total). Programmers can claim and release space using SpMalloc, a software allocator based on an implicit free list. Alternatively, the scratchpad can be partitioned statically.

3.2 UpDown Software

3.2.1 Programming UpDown

UpDown programs are written in UDWeave, a C-style programming language. There are two key concepts: threads and events.

Events are similar to functions in C. It specifies the operands available and the set of instructions that will be executed. Within an event, the lanes can perform computation, access their scratchpad memory, send messages to trigger events on any other lanes in the system, or issue DRAM operations. While creating messages and memory requests are conceptually similar to making function calls and data accesses in traditional architectures like CPUs and GPUs, the results are not immediately available after the invocation. Instead, the programmer has to write events to handle the return values.

Since the values in an event are local to its scope, programmers must use threads to keep track of the outstanding requests. Events declared under the same thread will run on the same set of hardware registers, meaning the programmer can preserve states between events.

3.2.2 Graph Representation

UpDown can choose between a range of possible graph representations. One of the most common approaches is the Parallel Graph Abstraction (PGA) library. The vertex features, edge features, and neighbor lists are all stored in hash tables and can be accessed with their IDs. The user program can invoke PGA events to perform retrievals and updates. To handle the skewness in neighbor list lengths, the library uses multiple layers of hash tables. The hash tables at the lower levels will have more buckets and smaller entry sizes since most graphs follow a power-law distribution, meaning that most vertices should have lower degrees. We only need to allocate a few long entries for the outliers. When configured properly, the space complexity can approach $O(n_e)$. This data structure also works well with our vertex-centric programs because we can obtain all the neighbors by doing an in-order traversal of the array. Furthermore, since many graph datasets do not have contiguous IDs, which are required for feature look-ups in dense matrices, using hash tables can save the pre-processing step.

We foresee some interesting use cases for this data structure. First, for GNNs that require sub-graph structures, we can create new hash tables for them and enable forward and backward look-ups. The adjacency matrix and the edge list approach only capture pair-wise relationships. Second, some GNNs might require creating subgraphs or performing negative sampling. In those cases, being able to perform tasks like checking whether a connection between two vertices exists will become handy. Since we generate edge IDs by hashing the source and destination vertex IDs, we can compute the expected ID and perform a $O(1)$ query to obtain the result. In short, we believe that this powerful data structure will unlock greater design space for GNNs.

3.2.3 Parallel Task Management

One of the key challenges associated with writing parallel programs is task management. UpDown uses the UDKVMSR library[37] to achieve that. Users need to provide a set of

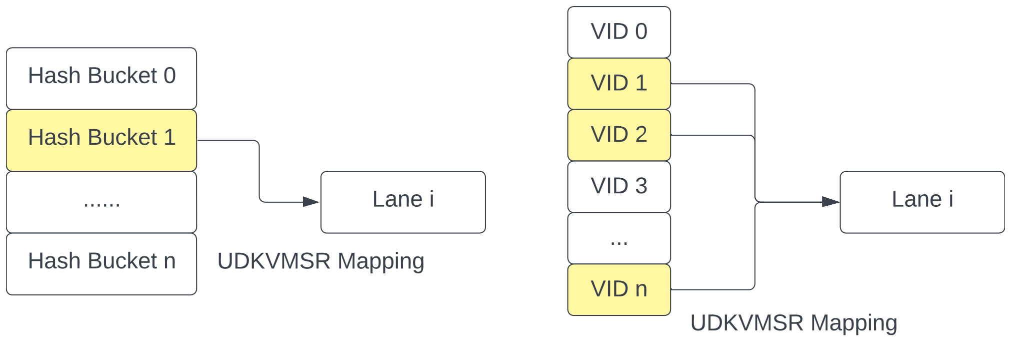


Figure 3.2: UDKVMSR Default Mapping

keys, an event that they want to trigger on each key, and the set of available accelerators. UDKVMSR spreads the tasks based on user-defined constraints, manages their status, and terminates once all tasks are complete.

The library has a few different modes of operations that enable easy and flexible task mapping as illustrated by Fig 3.2. When we use the PGA, users can simply pass the descriptor of the PGA and the event that each vertex should invoke. The UDKVMSR do-all library will distribute the work, dividing the tasks by assigning hash buckets to the workers, which then run the map function for all bucket entries. Thanks to the randomness produced by the hash function, we should expect each worker to receive roughly similar numbers of vertices, and the high-degree vertices should be distributed evenly among workers. UDKVMSR also provides a 1D array option. The user can provide a DRAM array, key count, and a value size, and MSR will read the key-value pairs in this array and invoke tasks on all the entries.

CHAPTER 4
GNN AGGREGATION ON UPDOWN

In this chapter, we will discuss how we utilize the hardware and software features of UpDown to implement and optimize GNN aggregation operations. We will also compare the UpDown and GPU approaches to discuss the expected benefits.

4.1 Vertex Centric Programs

We choose to implement GNN models using vertex-centric programs. The user needs to define a set of events starting with one that receives the target vertex ID. Based on that, the user can issue reads to the graph library to obtain the relevant data, perform message generation, and write back the aggregated message. A general template is included below. These events will be triggered and managed by the UDKVMSR library, which allows us to apply it to any stored graphs. We will use the GCN layer as an example to illustrate the implementation.

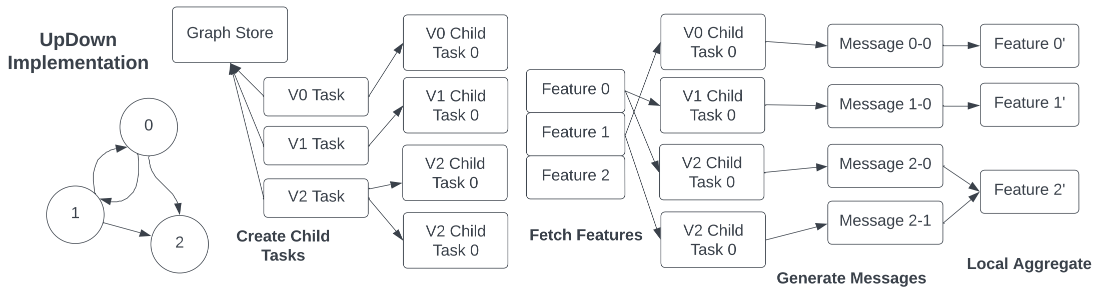


Figure 4.1: UpDown Implementation of GNN Aggregation

The program starts with the receive_task event in the vertex_program thread, where the user retrieves the necessary information about the destination vertex. In the GCN case, we fetch its degree, edge list address, and feature entry address. To store the aggregation results, we also need to allocate and initialize a region in the scratchpad using our SpMalloc library. We issue all four requests in the first event to maximize our ability to hide the DRAM latency. Once all the results have been received, we start creating child tasks to process the neighbors as shown by the generate messages part in Fig 4.1.

The child tasks are implemented in the neighbor_program thread in the template. For GCN, each task gets a vertex ID, the destination vertex degree, and the scratchpad address holding the partial aggregation results. It fetches the neighbor degree from PGA and computes the scale factor. Then, it streams the neighbor feature from DRAM. For each received segment, it scales the feature data to generate the messages. After that, the child task reads the current values on the scratchpad and updates the entry. Once the entire feature vector has been processed, the child task will send a return event to its parent. Note that it is possible to split the child tasks to run on multiple lanes and merge the partial results for additional parallelism. Once all the child tasks are complete, we know that the result in scratchpad is the aggregated message and can be flushed back to DRAM. To modify the model, the user only needs to update the neighbor program, which makes this approach extensible.

Listing 4.1: A template for vertex-centric program

```
thread vertex_program{
    long* local accum_ptr;
    long* neighbor_list;
    long degree, id;

    event receive_task(long vertex_id){
        send_spmalloc_allocate_event(MESSAGE_SIZE);
        send_get_neighbor_list_event(id);
        // Issue requests to retrieve information about source vertex
        id = vertex_id;
    }

  // Events handling return values are omitted

    event start_processing(){
        long * neighbor_entry_addr = neighbor_list;

        for(i = 0; i < degree; i = i + 1){
            send_dram_read(neighbor_entry_addr, fetch_neighbor_return);
            neighbor_entry_addr = neighbor_entry_addr + 1;
        }
    }

    event fetch_neighbor_return(long neighbor_id, long edge_id){
        send_event_new_thread(neighbor_id, id, accum_ptr, \
         neighbor_program::receive_task);
    }

    event child_task_return(){
        // Check for child thread termination.
        // On completion, it can return or perform additional tasks.
    }
}
```

```
thread neighbor_program{
    long* neighbor_feature;
    long* local accum_ptr;

    event receive_task(long neighbor, long target, long* local sp_ptr){
        get_vertex_feature(neighbor);
        // Get neighbor information

        accum_ptr = sp_ptr;
    }

    event receive_feature(long* feat_addr){
        stream_feature(feat_addr, FEATURE_WIDTH);
    }

    event stream_feature(){
        // Perform operations on received segment
        // Update partial result
    }
}
```

We believe a key benefit of this approach is the access locality. Under UpDown's MIMD model, we can easily define multiple steps of data fetching and transformation. The data segments are manipulated in registers and aggregated with the partial results in the scratchpad. We only need to flush the final outputs to DRAM at the end, saving valuable memory bandwidth.

Because GPUs adopt a SIMD architecture, GPU programmers have to reason about algorithms from a matrix perspective, performing uniform operations on entries to utilize fine-grained vectorization. However, the GNN aggregation involves non-uniform operations on the vertex feature matrix. The rows need to be transformed based on the source and destination features of the edges into messages, and the messages need to be aggregated based on the destination to form the new feature. The programmers need to either separate these data movements and manipulations by materializing the message matrix or trying to fuse them at the cost of reducing the flexibility of the message generation function.

For example, in GCN, the messages are produced by multiplying the source feature by $1/\sqrt{D(V_{src}) * D(V_{dst})}$. There are two approaches for implementing this. The PyG approach generates the message matrix, scales them using the per-message factor, and uses scatter-gather to sum up the messages. The DGL implementation first multiplies each row in the vertex feature matrix by $1/\sqrt{D(V)}$ (achieves the effect of scaling by $1/\sqrt{D(V_{src})}$), applies the fused SpMM kernel for aggregation, and then multiplies each row by $1/\sqrt{D(V)}$ again to obtain the output (achieves the effect of scaling by $1/\sqrt{D(V_{dst})}$). As the fused kernel is not truly customizable, additional manipulations are still required.

4.2 Graph Representation

While PGA offers a good starting point, hash tables usually come with a performance and memory overhead due to the hash bucket look-ups. Furthermore, the datasets we consider have contiguous IDs, meaning we can directly compute the entry addresses. To ensure memory consistency during updates, PGA also requires a range of accelerators to serve as management lanes. Each accelerator lane will respond to the accesses to a subset of buckets, caching the writes to those entries into their scratchpad before the DRAM updates return. For a skewed graph, the lane holding the hot-spot vertex might become a global bottleneck.

In this work, we create a read-only CSR-style representation with linear arrays for vertex and edge features, removing the need for dedicated lanes. The neighbor lists are stored contiguously in an array. Typically it contains the vertex IDs of neighbors. For datasets with edge features, the lists will store (VID, EID) pairs. To query the edge list, the lanes can fetch the target vertex's neighbor list start and end offsets based on its ID. This approach shifts the stress from one compute lane to the memory system, which has significantly greater capacity. For our vertex-centric approach, since the aggregated messages are written to independent memory locations after completion, this data structure is sufficient. To switch between this design and the PGA, users only need to change the event label for the query since the interface remains unchanged.

There are a few common graph representations used by GPU GNN libraries. Because GNN computations are sometimes written as $Y = AXW$, in which A is the adjacency matrix, X is the vertex feature matrix, and W is the weight used by the combine step, some libraries like the first iteration of AIMNet use a dense (n_v, n_v) matrix to store the connection between vertices. However, such an approach has a clear disadvantage: for sparsely connected graphs, most entries in the adjacency matrix will be zeros, wasting storage space. Furthermore, when we perform the aggregate step, a lot of the neighbor features will end up being multiplied by zero, wasting instructions. Thus, AIMNet has moved away from this representation for larger molecules. We should note that for smaller and denser graphs, this approach avoids the irregular accesses of neighbor features: the program takes advantage of tiled matrix multiplication to maximize reuse. There is also no need to perform index selection which saves instructions.

Edge list is also a popular approach used mainly by PyTorch Geometric. It encodes the connectivity through a $(2, n_e)$ array. The first dimension stores the source vertex ID and the second dimension stores the destination vertex ID. Compared to the $O(n_v^2)$, it reduces the space consumption to $O(n_e)$. It is also suitable for the PyG implementation because the index select and scatter-gather operation can directly run on the first and second dimension respectively. These operations extract parallelism by operating on the edges concurrently. While this data structure enables the implementation of aggregation from common primitives, it leads to high memory traffic and footprint. Furthermore, to access features like vertex degree, users need to perform additional processing on the edge list.

Both the latest version of AIMNet and DGL use a neighbor list representation of the graph. For AIMNet, because their graphs have limited skewness, they can pad all the neighbor lists to the same length, which allows them to use the Einstein sum function provided by PyTorch to perform the aggregation step. DGL is a more general-purpose library. Therefore, they have chosen CSR representation to ensure good performance on a range of graphs. They also design all their functions around this representation, writing an optimized SpMM kernel to provide high performance on common operators. However, when the operations do not fit under the fused kernel framework, they still need to resort to the edge list approach used by PyG for greater flexibility.

4.3 Capturing Reuse

To exploit reuse, we develop a read-only cache that enables retrieval of vertex features based on vertex ID. It can be utilized when there are reuse opportunities. The cache can be configured to operate in two modes: local and shared. In the local mode, each lane manages a separate set of entries. Although the capacity would be smaller as the scratchpad is divided (64KB per lane), it ensures that all scratchpad accesses are local to the lane, which has lower costs. In the global mode, each lane would handle a range of vertex IDs, and requests would be forwarded accordingly. In both cases, the cache will return the local address of the data to the user.

We use a simple replacement policy. The vertex IDs are mapped to the cache entries based on their least significant bits. Each entry has a counter indicating the number of threads accessing the data. When all the lanes free the data, the entry becomes available for the next request. When multiple requests are mapped to the same slot, the ones that do not get the slot will spin-wait.

This software-defined cache offers a few advantages. First, it removes the expensive hardware used to maintain memory consistency. In our program, since all the partial results are private to lanes, contention can be easily avoided. Second, it gives users the ability to customize the design by changing the capacity allocation or the replacement policy. For example, for graphs with extreme skewness, it might make sense to perform eviction based on vertex degrees. Additionally, we can flexibly determine what data to cache. If we know there are no reuse opportunities, we can avoid the read and write costs through streaming while saving precious local memory.

Having a cache alone is not sufficient: prior works[41] have shown that GNN aggregation lacks reuse unless we process the tasks in a certain order. For example, in AIMNet, we found that the vertices with similar IDs have a high degree of overlaps in their neighbor lists. For vertex with id i, on average 65% to 73% of its neighbors have appeared in the neighbor list of $i-1$ on the two datasets we focus on. Thus, while the do-all over array interface simplifies the parallel task creation, mapping tasks to lanes without explicit control (e.g. based on hash buckets) might not always be desirable. Luckily, we can enforce ordering using custom map tasks.

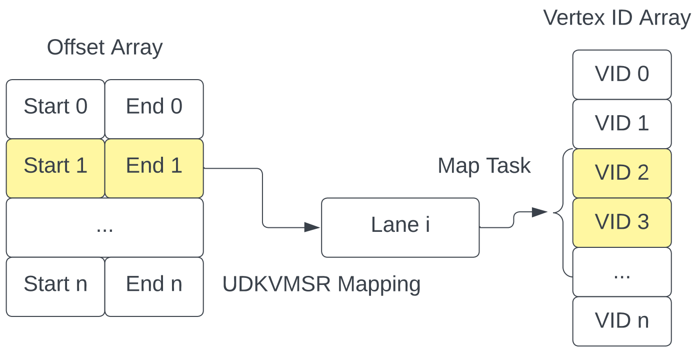


Figure 4.2: UDKVMSR Custom Mapping

In this work, we mainly rely on two different approaches corresponding to the two cache configurations. If we want to map a group of entries to a lane, we can use a pair of start and end offsets as the key. The offsets can either express an ID range or a set of entries in an ID array. The map task will manage the vertex tasks in this case as shown by Fig 4.2. Since the map task is the smallest unit of operation, we get the guarantee that those vertices will run on the same physical location. Alternatively, it is sometimes more desirable to map a set of tasks onto the same accelerator. In that case, we still use the offsets as map task inputs. The map tasks forward the job to lane 0 on all accelerators, and that lane will then spread the work onto all lanes.

For AIMNet, we use IDs to determine the division. However, for the GNN datasets, we find that the contiguous IDs do not imply similar neighbors. Thus, we use the METIS partitioning algorithm to generate task division. Since METIS partitions have minimal connections to outside edges, while running the assigned vertex tasks, the accelerators should be able to reuse features stored in their cache. For the local cache approach, we create one METIS partition for each lane. For the UD cache version, we create larger partitions and assign them to accelerators. Since the partition sizes are usually larger than the cache capacity on one accelerator, we generate multiple partitions per accelerator in the UD cache case.

4.4 Load Imbalance

Given the power-law distribution of vertex degrees in graphs, we expect to encounter load imbalance between vertex tasks, which challenges the practicality of our approach. We choose to solve that problem by redistributing the neighbor tasks for high-degree vertices.

The strategy we employ is straightforward: if a lane receives a vertex with degree above a threshold, it will split the neighbor list and run message tasks on a range of lanes. Since the GNN aggregators are usually order-invariant, each lane receiving this task can produce a partial result by aggregating messages generated by its assigned subset of neighbors. Then

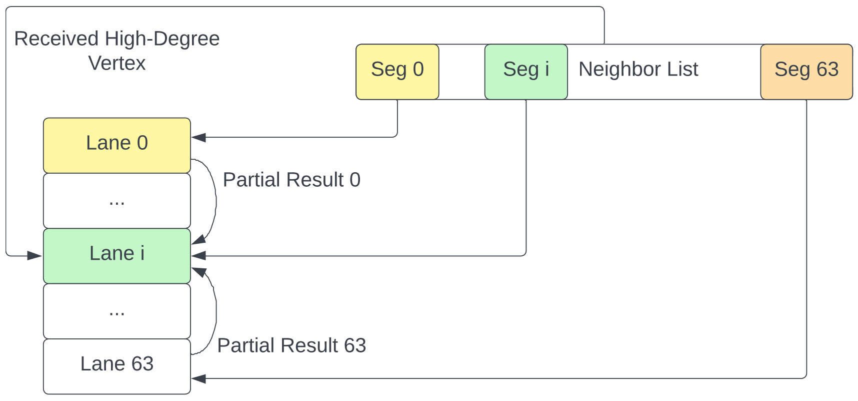


Figure 4.3: Message Task Redistribution based on Global Communication

these partial results will be sent back to the lane initiating the process, which then reduces them to obtain the final output.

For simplicity, we use a constant cutoff for the datasets, and the tasks will be spread onto the lanes within the same accelerator (64-way split). This process is illustrated by Fig 4.3. Note that a more optimized strategy might be to create multiple tiers of splitting, which balances between the number of partial results produced, which determines the workload on the lane initiating the split, and the maximum number of neighbors in the produced tasks. There is also no limit to the number of lanes the task can spread since the partial results are private to each lane, and the final exchange happens through global messaging.

CHAPTER 5

EVALUATION

In this section, we evaluate UpDown's performance on GNN aggregation operations. We will first introduce the methodology for performing the evaluation. Then, we demonstrate the importance of aggregation in the application with a workload breakdown on GPUs. After that, we compare UpDown and GPU performance on the baseline UpDown implementation and the version with caching and reordering optimizations. Based on the results, we put the speedups back into the application context. Lastly, we show that our design can scale to the target configuration.

5.1 Methodology

5.1.1 Models and Datasets

Dataset	**# Vertices**	**# Edges**	**Avg Deg**	**# Features**
YouTube[42]	1,134,890	2,987,624	2.63	602
PubMed[43]	19,717	88,648	4.50	500
Live Journal[42]	3,997,962	34,681,189	8.67	200
Flickr[45]	89,250	899,756	10.08	500
Yelp[45]	716,847	13,954,819	19.47	300

Table 5.1: Dataset Statistics

Dataset	**# Vertices**	**# Edges**	**Avg Deg**	**# Max Deg**
Molecule 5786	5,786	257,406	44.49	71
Molecule 13226	13,226	593,752	44.89	71

Table 5.2: AIMNet Dataset Statistics

For the GCN and GIN models, we choose some frequently used datasets from SNAP[24] and the PyG graph collection[12]. Since some datasets do not have vertex features, we follow the approach used in GRIP[22], generating a 602-element feature vector for each vertex. The

Live Journal dataset is an exception: due to the high number of edges, PyG will run out of memory. Thus, we reduce its feature count to 200.

We only consider one GNN layer in our experiments because the layers will be processed sequentially in both cases. For the GCN layer, because GPUs perform combine first to reduce the dimension for later stages, we set the output channel to match the input channel so that when comparing the aggregate phase, the works performed by both platforms are equal. Since the computations for the aggregation phase and combination phase are proportional to the feature width, this choice will not affect the workload breakdown. For GIN, such optimizations are not possible, and we set the output dimension to 64 for all datasets. As GIN models typically use an MLP in the combination phase, we add a second linear layer with 64 input and output channels.

5.1.2 AIMNet Performance Proxy

Instead of modeling the complete AIMNet forward pass, we extract the aggregation portion and build functional equivalent versions on UpDown. AIMNet has two aggregation steps, and each can accept two sets of vertex feature shapes (1D and 2D). In the 1D case, the vertex features are one-entry values. For the 2D case, the vertex features are 16×16 matrices. Thus, we build proxies for each of the four combinations. We compare the aggregation time for each set of feature shapes by combining the results from the two steps.

The first aggregation step takes vertex features and 16-element edge features. For the 2D feature case, each column in the neighbor feature matrix is multiplied by the corresponding column in the edge feature to produce the message. To perform contiguous accesses of vertex features, we implement it as elementwise multiplication of each row in the feature matrix with the edge feature. For 1D vertex features, the messaging step becomes scaling the 16-element edge feature by the vertex entry. The messages are reduced using summation.

In the second aggregation step, the edge features are 16×3 matrices. For 2D vertex

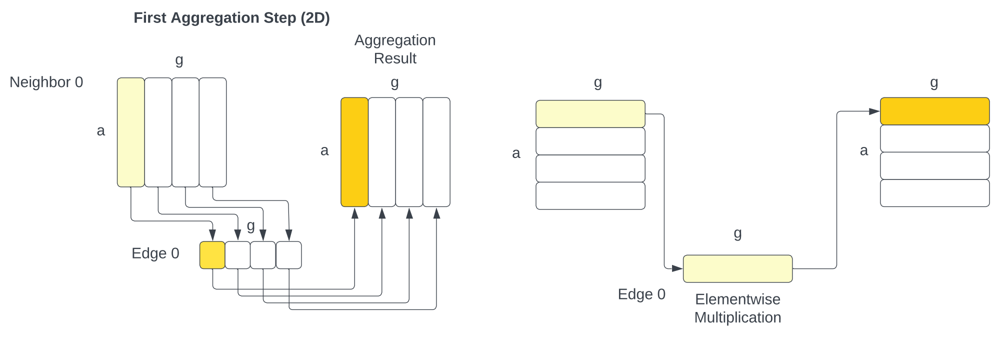


Figure 5.1: AIMNet Aggregation Step 1 with 2D Features

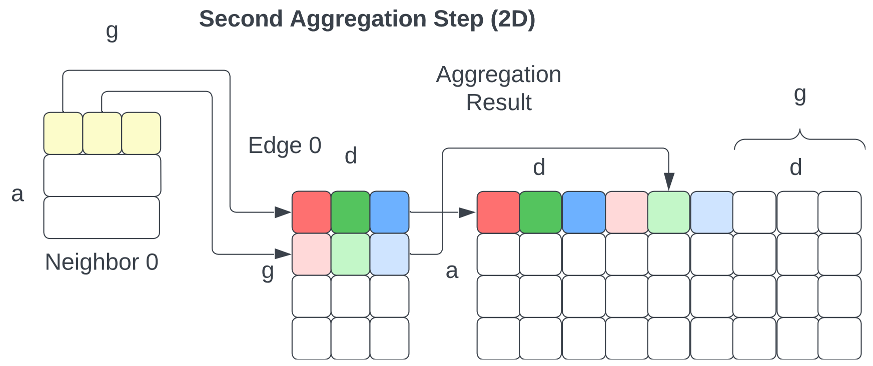


Figure 5.2: AIMNet Aggregation Step 2 with 2D Features

features, the entries in every vertex feature row are used to scale the corresponding 3-element vectors in the edge feature, producing a $(16, 48)$ message matrix as shown in Fig 5.2. The 1D feature case is similar to that in the first step, except the product becomes $(16, 3)$ matrices due to the shape of the edge feature. The messages are also summed together to produce the output.

5.1.3 Platform Setup

The key parameters of the systems we compare are listed in Table 5.3. To ensure a fair comparison between the UpDown design and GPU, we scale up the UpDown node design to contain 13.5k lanes, providing the same issue rate as the CUDA cores in the H100 GPU.

We project UpDown's speedup over H100 using the following approach. We obtain the

Platform	**Compute Units**	**On-Chip Memory**	**Off-Chip Memory**
UpDown (Equal Issue Rate)	13,516 Lanes @ 2.0 GHz	4 MB scratchpad shared by every 64 lanes	8 stacks of HBM3e with total bandwidth of 8.8 TB/s
NVIDIA H100[3]	16,896 Cores @ 1.6 GHz	33.8 MB L1 data cache/shared memory & 50 MB L2 cache	80 GB HBM3 with bandwidth of 3.36 TB/s

Table 5.3: System Setup

results with 512 lanes on YouTube, Yelp, and Live Journal. For the smaller graphs like PubMed, Flickr, and AIMNet molecules, we use 256 lanes. We scale the time consumption accordingly assuming linear speedup to obtain the issue-rate limited time. The choice to use smaller machine sizes for the experiments ensures each lane gets a sufficient number of tasks to amortize the library overhead and achieve latency-hiding. We will show later in this chapter that the design has good scalability on two of the larger graphs. For the memory bandwidth limited time, we divide the traffic by the system bandwidth of 8.8 TB/s in our design. The final projection time is computed by taking the greater value of these two projected times to obey the resource constraints.

5.2 Workload Breakdown

We first profile the PyG implementation of the GCN and GIN model on the H100 GPU. The time consumption is broken down by kernels and shown in Fig 5.3 and 5.4. There are a few notable observations. First, for both GCN and GIN, the index select takes a significant portion of time because of the large message matrix size. Since the elementwise kernels operate on the materialized message matrix, they also account for a large portion of the execution time. Second, the GEMM operations account for anywhere between 4.5% to 40% on GCN and 1.4% to 8.4% on GIN. The variation is caused by the differences in average degree. The aggregation workload is typically proportional to the number of edges whereas

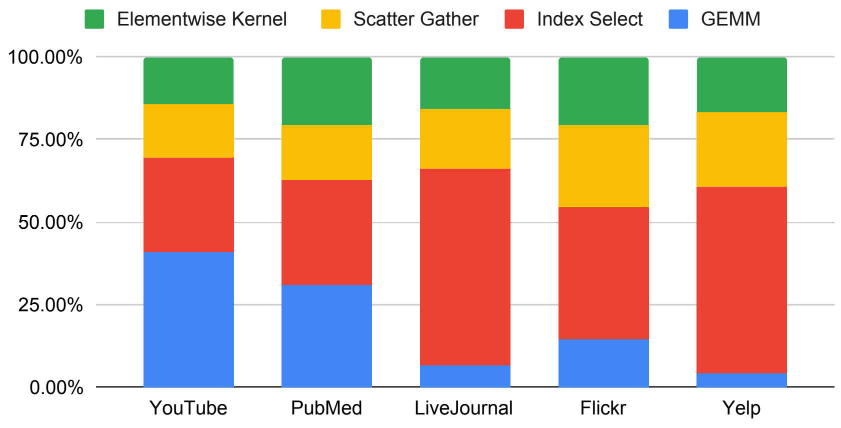


Figure 5.3: H100 GPU Time Consumption Breakdown on GCN under PyG

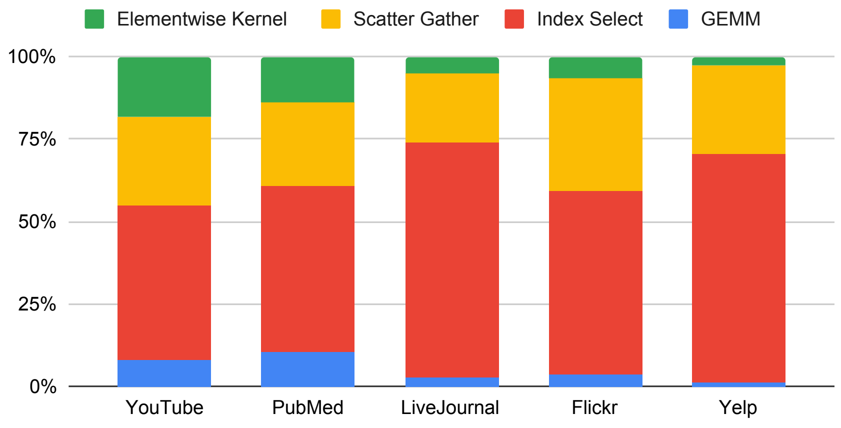
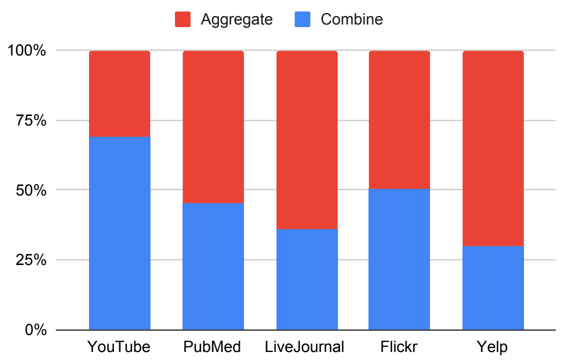


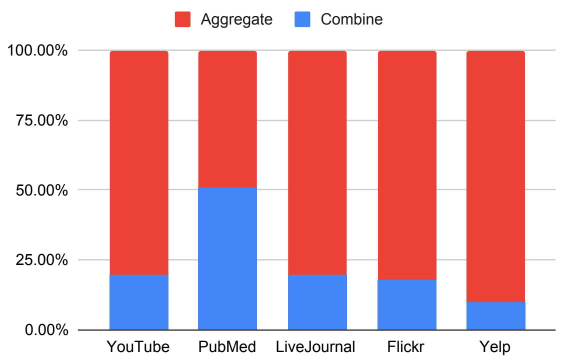
Figure 5.4: H100 GPU Time Consumption Breakdown on GIN under PyG

the combination workload is proportional to the number of vertices. Thus, for denser graphs like Yelp, more time will be spent on aggregation. Overall, these ratios suggest optimizing GEMM is less important than improving the aggregation phase. It is necessary to highlight that these results were obtained under FP32, which does not have tensor core support on H100. If we can use tensor cores to further accelerate GEMM, we can expect its ratio to drop even further.

The results from DGL are shown in Fig 5.5. While the GEMM workload is consistent across the two setups, we can see that a significantly smaller portion of time is spent on



(a) GCN



(b) GIN

Figure 5.5: H100 GPU Time Consumption Breakdown under DGL

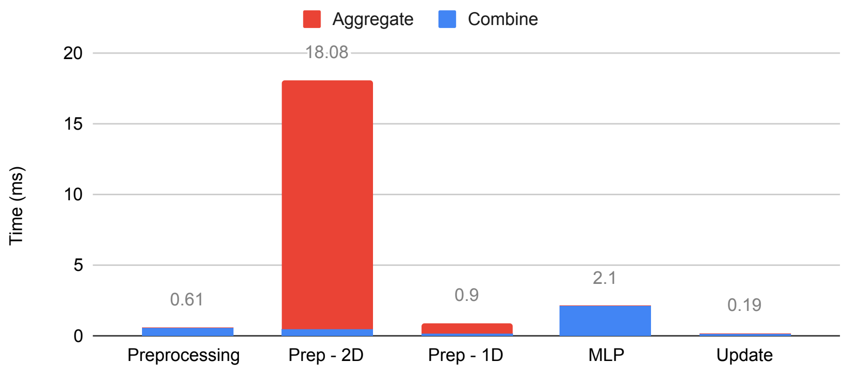
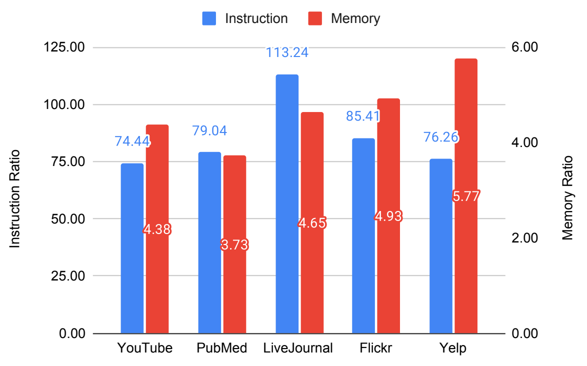


Figure 5.6: H100 GPU Time Consumption Breakdown on AIMNet on Molecule 13226

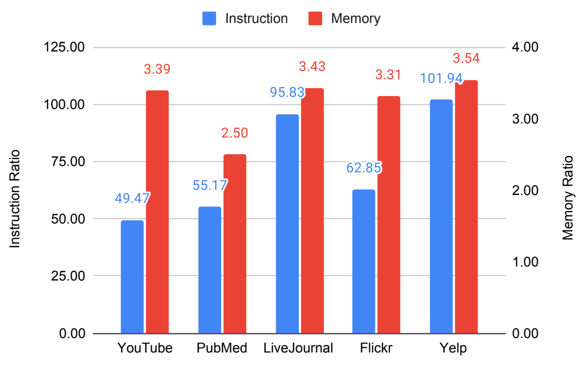
aggregation compared to PyG. This is mainly because DGL fuses multiple kernels together to improve efficiency. Despite that, we can see in many cases, aggregation still takes up a higher percentage of the time consumption.

We also profile the AIMNet run on the H100 GPU. The AIMNet forward pass involves four stages: preprocessing, preparation, MLP, and update. For each stage, we divide the computation into aggregate and combine. In this case, combine not only refers to GEMM operations but also includes dense and regular tensor operations on which we do not expect to see a benefit for UpDown. As shown by Fig 5.6, a significant portion of the time consumption comes from aggregation. These results serve as the motivation behind our focus on the aggregation phase.

5.3 Aggregation Phase Comparison



(a) GCN

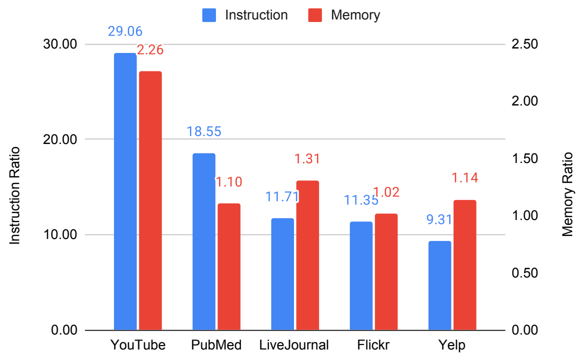


(b) GIN

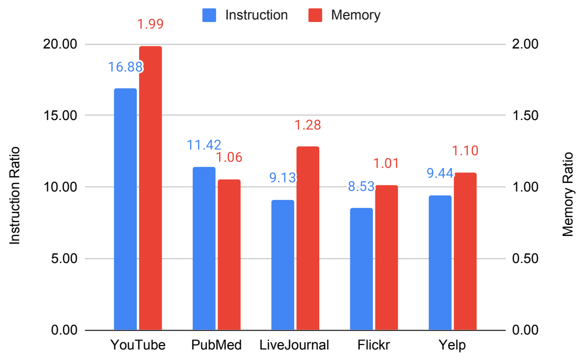
Figure 5.7: Instruction and Memory Traffic Ratio between PyG on H100 GPU and UpDown

We first compare the instruction and memory traffic between the PyG implementation of GCN and GIN and the UpDown implementation. The results are normalized based on UpDown's statistics (UpDown is 1.0). As shown by Fig 5.7, PyG requires significantly more instructions and memory accesses. For graphs with higher average degrees like Yelp, the memory ratio tends to be greater due to the materialization and manipulation of the message matrix. Since GNN aggregation has low arithmetic intensity, such high traffic will negatively impact performance. Additionally, it limits the maximum graph size since the results have to fit into the DRAM.

The comparison with DGL is shown in Fig 5.8. Compared to PyG, the gap between UpDown and GPU reduces. However, we still observe higher instruction usage. In terms of memory, despite the use of fused kernels and having a 50 MB L2 cache, GPU still generates more accesses. There are two reasons. First, the GCN and GIN operations are not fused perfectly. For GCN, the message scale factor $1/\sqrt{D(V_{src}) * D(V_{dst})}$ requires elementwise multiplication before and after the custom SpMM kernel, producing additional traffic. For



(a) GCN



(b) GIN

Figure 5.8: Instruction and Memory Traffic Ratio between DGL on H100 GPU and UpDown

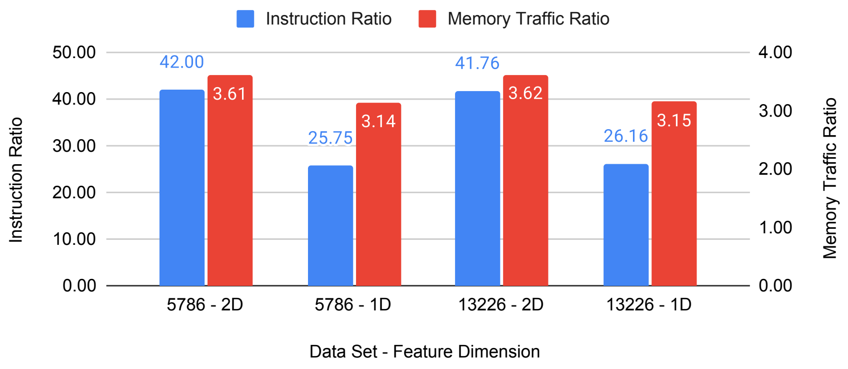
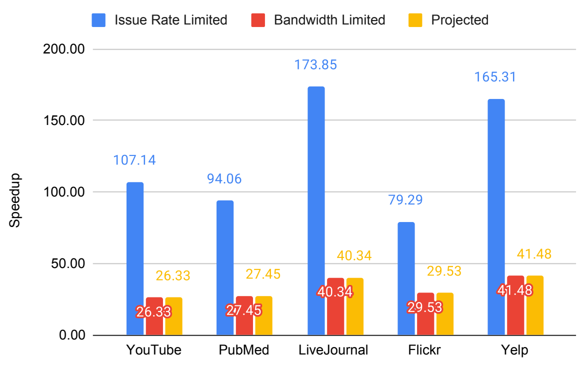


Figure 5.9: Instruction and Memory Traffic Ratio between AIMNet on H100 GPU and UpDown

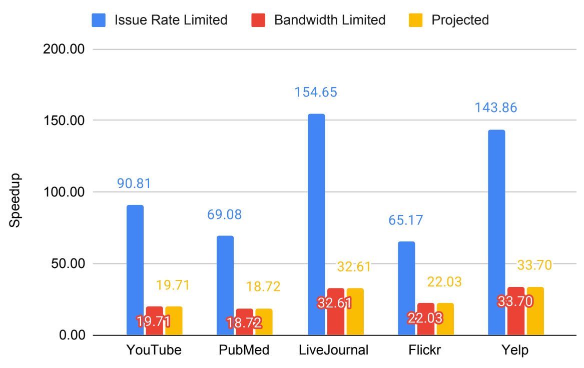
GIN, the addition of original vertex features to the aggregated messages requires additional DRAM access. Second, since the graph connectivity is irregular, the L2 cache is insufficient, producing hit rates in the range of 30-40%.

The results on AIMNet are consistent with our prior findings. Although AIMNet improves aggregation write locality compared to PyG's scatter add, it still materializes the messages due to its more sophisticated message generation algorithm. Overall, we demonstrate that even the state-of-the-art GPU libraries require more work than our UpDown implementation.

As a result of PyG's high bandwidth and instruction usage, UpDown achieves a projected geomean speedup of **32.40x** on GCN and **24.56x** on GIN. With DGL, the gap becomes

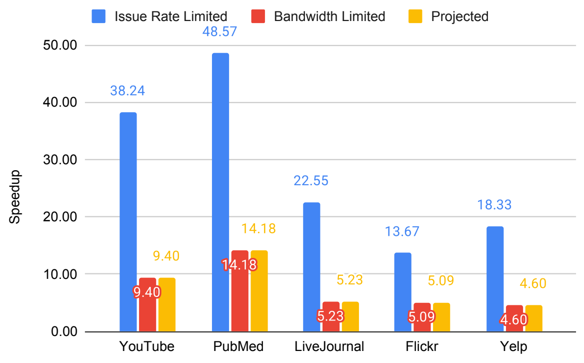


(a) GCN

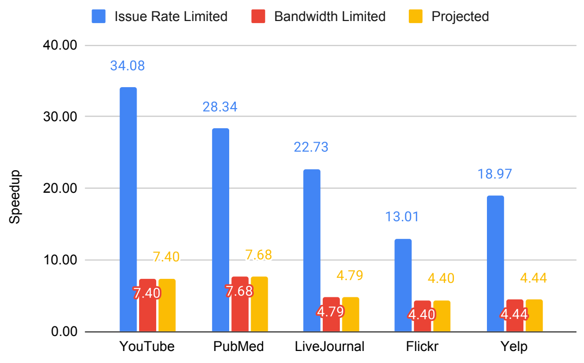


(b) GIN

Figure 5.10: UpDown Projected Speedup over PyG on H100 under Various Conditions



(a) GCN



(b) GIN

Figure 5.11: UpDown Projected Speedup over DGL on H100 under Various Conditions

narrower. However, we still expect a geomean speedup of **6.96x** on GCN and **5.56x** on GIN. Another notable result is that in all cases, the memory bandwidth is the limiting factor. Take GCN on DGL as an example, the geomean issue-rate limited speedup is 25.36x, which is 3.64x higher than the memory-bound speedup.

When compared to AIMNet, UpDown achieves up to **36.57x** projected speedup on the aggregation phase. Thanks to AIMNet's more complex aggregation operations, the gaps between issue rate limited and bandwidth limited speedups also reduce to between 1.76x to 2.76x. However, UpDown is still memory-bound in those cases.

These results suggest we can further improve performance by extracting the reuse oppor-

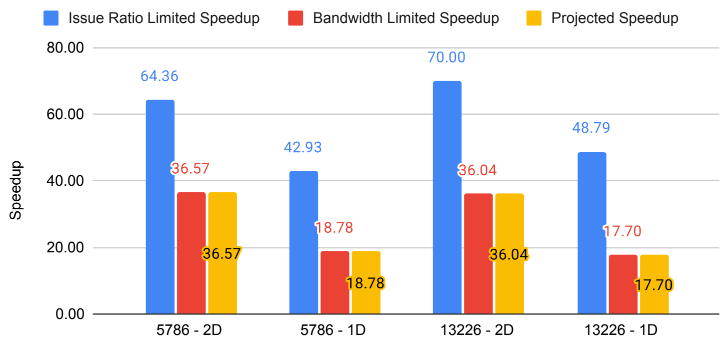


Figure 5.12: UpDown Projected Speedup over H100 on AIMNet under Various Conditions

tunities even if it comes at the cost of increasing instruction count.

5.4 Exploiting Reuse

For these experiments, we cache at the granularity of vertex features. We use 32 entries per lane for Yelp and Live Journal (2048 entries for the UD cache case). The remaining datasets use 16 entries per lane (1024 entries per UD) due to their longer feature vectors. We only consider GCN in this case as the access patterns are similar.

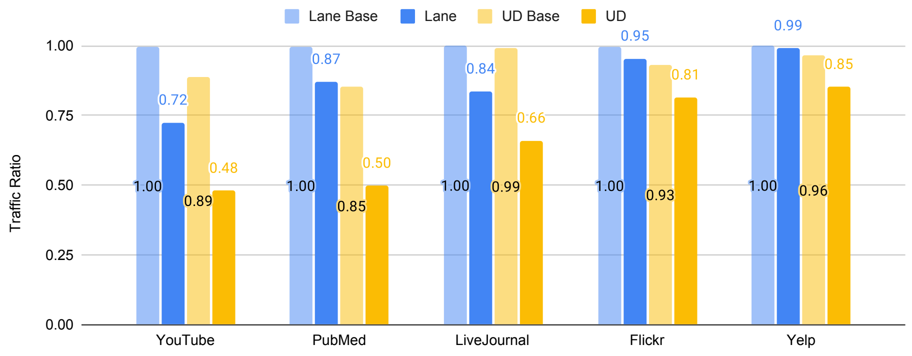
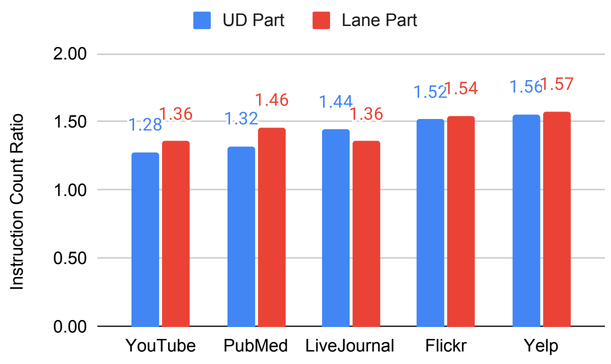
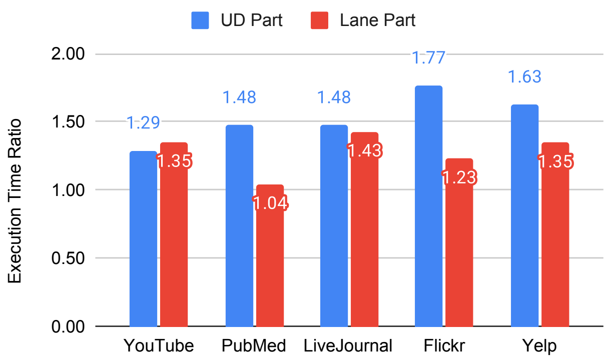


Figure 5.13: UpDown Memory Read Traffic with and without Remapping

We plot the memory read traffic in Fig 5.13. The results are normalized based on the traffic on the baseline version without cache. The lighter bars are measured with the original



(a) Instruction Ratio

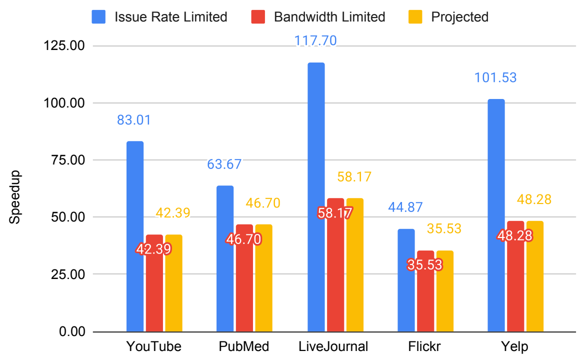


(b) Execution Time

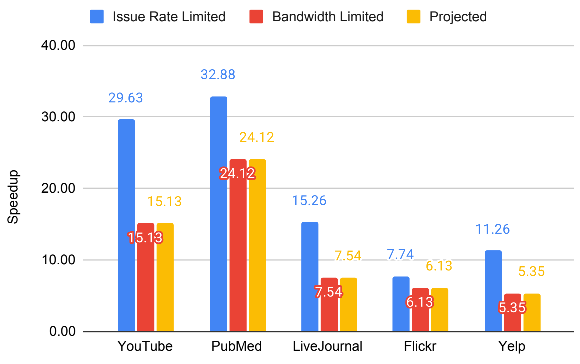
Figure 5.14: Overhead of UpDown Software Cache

random mapping with the same cache setup. We can see that for most datasets we get little to no traffic reduction. Thus, the observations match our prediction about the limited usefulness of cache on graph applications. After remapping, the UD cache can reduce read traffic by up to 50%.

We then evaluate the cost of adding the cache in terms of execution time and instruction count. The results are normalized based on the baseline code and shown in Fig 5.14. The software cache produces anywhere between 28-57% instruction count increase and 4-77% execution time increase, which is reasonable given the aggregation operation is relatively simple for these two networks, meaning the feature access cost will dominate.



(a) PyG



(b) DGL

Figure 5.15: UpDown Projected Speedup with Cache over H100

Since the UD cache version produces the best result, we perform the speedup analysis based on that configuration. As shown by Fig 5.15, the gap between issue rate limited and bandwidth limited cases reduces compared to Fig 5.10a and 5.11a. Since the program is still in the memory-bound region, the cost of the cache is not an important concern. With cache, the speedup becomes **45.62x** compared to PyG and **9.80x** compared to DGL, which represents a **41%** improvement.

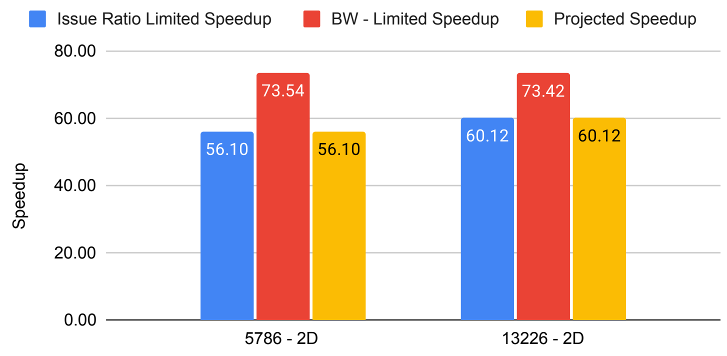
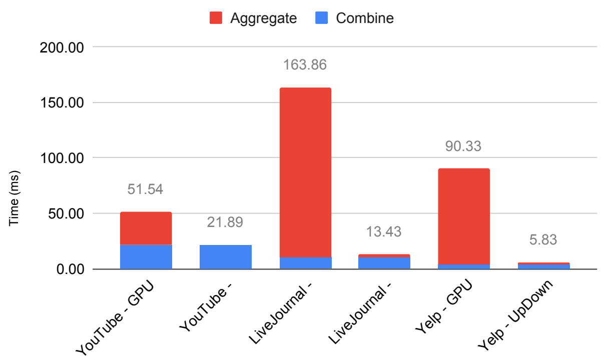


Figure 5.16: AIMNet UpDown with Cache Projected Speedup over H100

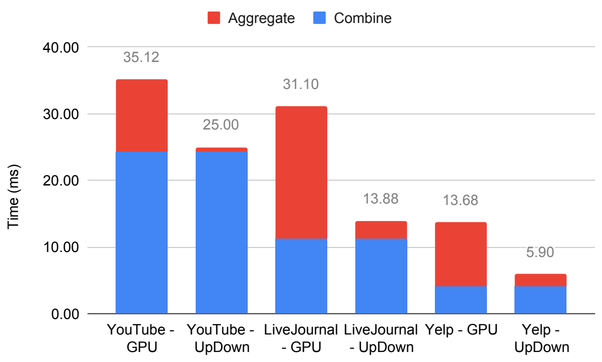
For AIMNet, we only apply the optimization to the case with 2D vertex features because the DRAM traffic produced by reading the one-element vertex feature in the 1D case is small compared to that for retrieving the edge feature, and the edge features have no reuse since they are only used once. The results are consistent with the previous experiments, with a geomean speedup of **58.08x**, which represents a **60%** improvement from the case without cache. However, unlike GCN, the AIMNet proxy is projected to be compute-bound in this case as shown in Fig 5.16.

5.5 Application-level Analysis

To analyze the impact of UpDown on the entire forward pass of GNNs, we make the assumption that the final design will achieve performance parity with H100 on GEMM and regular dense operations, which is reasonable as we target having equal issue rate and the UpDown design has SIMD2 FP32 instructions. The results below are projected under both issue rate and memory bandwidth constraints.



(a) PyG



(b) DGL

Figure 5.17: UpDown Projected GCN Application Runtime with Cache

For GCN, we project a **5.80x** speedup over PyG and a **1.84x** speedup over DGL on the entire forward pass. The improvements are **11.79x** and **2.70x** respectively for GIN. The speedup over DGL is lower because aggregation accounts for a smaller portion of the time consumption. With cache and remapping based on METIS, the speedup on GCN becomes **6.05x** over PyG and **2.04x** over DGL.

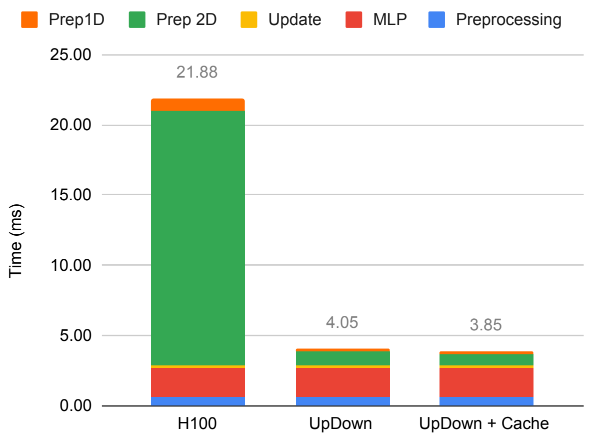


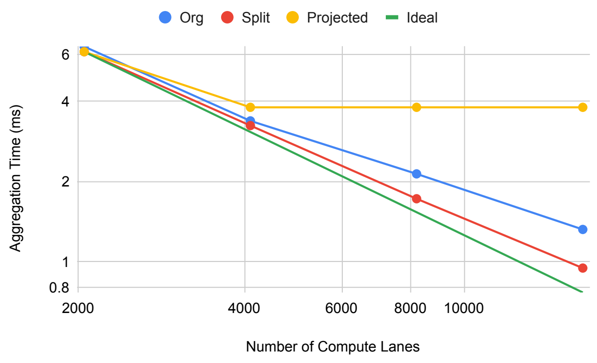
Figure 5.18: AIMNet UpDown Application-level Speedup over H100

For AIMNet, we plot the projected application time consumption for molecule 13226 in

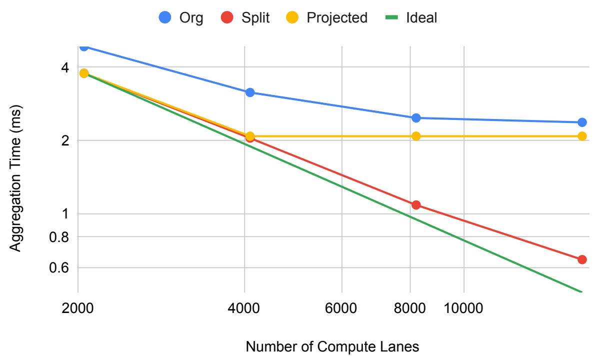
Fig 5.18. With UpDown, the Prep 2D portion is reduced significantly. The overall speedup is projected to be **5.40x** without cache or **5.68x** with cache. Note that based on Amdahl's Law, the maximum speedup we can obtain in this case is 6.22x.

5.6 Scaling to the Full Design Specification

We validate the scalability of our design using two of the larger datasets: Yelp and LiveJournal. Compared to Yelp, LiveJournal has a lower average degree and a higher vertex count, meaning it's less likely to be bottlenecked by load imbalance.



(a) LiveJournal



(b) Yelp

Figure 5.19: UpDown Scaling Results

In Fig 5.19, the blue line represents the original setting without applying the memory bandwidth limit, and the red line represents the version with task splitting. It is also not subject to bandwidth limits. The yellow line represents the splitting version after applying the bandwidth limit. The green line is the ideal scaling result based on splitting version performance at 2048 lanes assuming linear speedup.

On LiveJournal the original design does not perform significantly worse than the splitting version. If we consider the memory bandwidth, both versions will end up being memory-bound. However, for Yelp, the performance of the original version stops increasing beyond 8192 lanes, at which point it does not hit the memory limit. The splitting version, on the other hand, continues to scale to the design point.

Additionally, we observe that the version with splitting hit the memory limit at around 4,000 compute lanes, meaning we can achieve the projected speedup with roughly 29.6% of the issue rate H100 with current memory technology.

Overall, we believe these results justify our projection based on issue rate and memory bandwidth limits, proving the degree skewness is unlikely to make the vertex-centric approach unpractical.

CHAPTER 6
RELATED WORK

6.1 Sparse Tensor Accelerators

Sparse matrix-matrix multiplication shares many common challenges with GNN workloads, and they have received abundant attention in the architecture field. SpArch[48] is one of the earlier works targeting generalized sparse matrix-matrix multiplication (SpGEMM). The authors build a custom accelerator that uses the outer product approach. They pipeline the generation and merging of partial matrices, design a matrix representation to reduce the number of partial matrices, and use a scheduler based on Huffman tree to optimize merging order. SIGMA adopts the inner product method, using a forwarding network to route data to the processing elements so that they are highly utilized despite the sparsity[28]. GAMMA[46] observes that the inner product approach has poor input reuse whereas the outer product requires complex merging operations. It uses Gustavson's algorithm with a custom cache optimized for its access patterns and a scheduler for balancing the workload. Additionally, they propose an algorithm for row reordering to maximize reuse. Spada[25] proposes selecting between these two dataflows based on input characteristics. They design an eviction policy that keeps entries fetched by the adjacent rows in the first matrices in the cache to maximize reuse. Additionally, they suggest grouping rows based on non-zero counts since rows with similar lengths tend to share similar column index distributions.

6.2 GNN Models

Apart from the models we discussed in this work, numerous designs have been proposed to improve GNN performance and address different challenges. GraphSAGE[16] suggests that existing methods require that all the nodes are present during training, which makes generalizing to unseen nodes hard. Furthermore, for larger graphs, it is challenging to fit the data into GPU memory. Thus, the writers propose neighborhood sampling as a way to improve generalizability and scalability. The graph attention network[35] introduces attention mechanisms to weigh the contributions of neighbor vertices dynamically. A follow-up to this work claims that the order of computation in GAT is suboptimal, causing the scores to be unconditioned on the query node[6]. They propose the variant GATv2 and show that it is more expressive. Simplified Graph Convolution (SGC)[38] proposes that the activation function and weight matrices between layers should be removed, allowing the resulting network to perform pure feature propagation. By precomputing the result of A^k, SGC reduces the entire forward pass to one matrix multiplication. It also reduces training costs thanks to the use of one learnable weight matrix. Principal Neighborhood Aggregation[8] combines multiple aggregators like mean, sum, max, and standard deviation. The writers believe that one aggregation function fails to capture the diversity of information in the neighborhood of vertices. They also use scaling to offset the limitation in existing models which cause high-degree vertices to dominate the representation. Additionally, the work by Sun et al.[34] highlights the importance of higher-order interactions between nodes in capturing complex relations in many domains and proposes a framework for incorporating it in GNNs.

In short, the wealth of research in this area highlights the importance of general-purpose accelerators in this field. We believe that the flexibility offered by MIMD architectures will unlock greater design space for ML researchers.

6.3 GNN Accelerators

One of the earliest attempts at building an accelerator for GNNs is AWB-GCN [13]. The writers adopt a SpMM-based formulation. It treats the feature matrix as a sparse matrix as well and decomposes the GNN computation into two SpMM operations. They then focus on addressing the imbalance in non-zero distributions caused by the graph structures and dataset properties, using approaches like work sharing among neighbors, shuffling workloads to prevent clustering of non-zeros, and splitting the dense rows to improve performance. In HyGCN[41], the writers instead suggest that GNN workloads exhibit a heterogeneous workload: the messages and the model weights are dense while the adjacency matrix is sparse, meaning the computation can be separated into a dense combination phase and a sparse aggregation phase. Their aggregation engine uses a CSC adjacency matrix as its input. It runs a set of columns (destination vertices) at a time, processing the neighbors in blocks so that their features fit into the on-chip cache. For the combination phase, they use a systolic array to achieve efficient matrix multiplication.

While most of the accelerators following it adopt this workload division, they propose optimizations for various aspects of the computation. Rubik[7] uses locality-sensitive hashing to cluster vertices with more shared neighbors, maximizing the cache hit rate when fetching neighbor features for vertices in the group. It also tries to reuse intermediate aggregation results for neighbor pairs shared by multiple vertices. EnGN[19] uses a ring-edge-reduce dataflow which moves data across PEs and performs aggregations in a selective fashion to avoid the random accesses of neighbor features. I-GCN[14], a more recent work, attempts to address the challenges associated with poor cachability of neighbor features by introducing a BFS-style algorithm that identifies "islands" in the graph which have high internal connections and low external connections.

Some works in this area focus on improving the flexibility of accelerators for emerging models. DeepBurning-GL[26] presents a GNN performance analysis tool and a set of prebuilt templates for generating accelerator designs. GRIP[22] proposes a programming model that divides the GNN into three execution phases: edge-accumulate, vertex-accumulate, and vertex-update. In each phase, it invokes a user-defined function to enable customization. Multiple programs can be composed together to enable complex models. Writers of FlowGNN[31] point out that there are special GNN designs that make the approaches and optimizations proposed by other accelerator papers invalid. First, for models like principal neighborhood aggregation, which use aggregation operations other than sum, the computation can no longer be expressed as SpMM. Second, many models require edge embeddings in the process of message generation, meaning that the same source vertex can send different messages depending on the edge information. Thus, existing optimizations like merging common neighbor pairs no longer apply. Lastly, anisotropic GNNs like GAT, in which the messages are generated based on both the source node and the neighbors of the destination node, require materializations of explicit messages. Therefore, they propose dataflow which performs explicit message passing for generality.

CHAPTER 7
SUMMARY AND FUTURE WORK

In this paper, we propose a vertex-centric approach for performing GNN aggregation on UpDown. Supported by the MIMD architecture, our program generates messages on the fly and merges them directly in local memory, minimizing the memory traffic. The GPU solutions, by comparison, lack a way to flexibly express and implement the computation when the operation involves simultaneous data movement and transformation, resulting in excessive materialization and instruction overhead. Although fused kernels alleviate the problem, they suffer from limited generality. Furthermore, while the GPU interface does not expose control for task distribution, UpDown can perform fine-grained mapping, allowing it to better capture reuse opportunities. Additionally, we prove that by splitting and redistributing high-degree vertex tasks, the approach can scale under degree skewness.

We study two popular GNN libraries and an advanced chemistry GNN based on custom implementation. Compared to a state-of-the-art H100 GPU, the UpDown design with an equal issue rate achieves a 45.62x speedup on aggregation compared to PyG and a 9.80x speedup compared to DGL on GCN. For AIMNet, a more sophisticated model, UpDown can deliver 58.08x speedup compared to the optimized implementation. For the full forward pass, this translates into 1.91x and 6.80x over PyG and DGL on GCN and 5.68x on AIMNet.

While this work shows the potential of UpDown, there are many areas left to explore. First, we believe the message generation will become more complex to capture additional information about the interaction between vertices. For example, as discussed earlier, the main improvement from GAT to GATv2 is the replacement of scalars from source and destination to vectors, which allows more information about the two endpoints to be exchanged. Under the current formulation for fused kernels, this leads to the materialization of the large $O(n\_e)$ message matrix with vector entries. UpDown, by comparison, can naturally express all the steps, significantly reducing the memory traffic. We should study how the workload

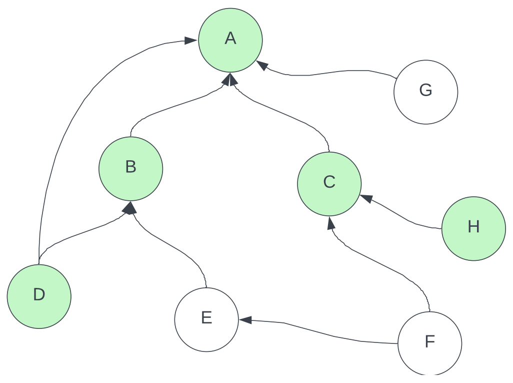


Figure 7.1: Example of Ego Network

changes as more models move in this direction, validating that vertex-centric programs will produce greater benefits.

Second, neural networks with dynamic behaviors are also gaining attention thanks to the success of Mixture-of-Experts (MoE) models. Similarly, the relational graph convolution network[32] generates messages by selecting a weight matrix based on the edge type and performing matrix-vector multiplication with the source vertex feature. This operation is still poorly optimized, with PyG using a loop to iterate through all the edge types. Given UpDown's MIMD flexibility, it would be interesting to see how we can better perform this computation.

One additional challenge was proposed by IARPA, which aims to study GNNs on extremely large graphs with high skewness. The assumption is that the graph can no longer fit into the memory of a single device, and it would be impractical to use traditional GNN algorithms under which every neighbor is considered in the forward pass. In their design, before inference, we need to sample a five-layer ego network around the target vertex through a BFS-style traversal like the one shown in Fig 7.1. Then, a two-layer GCN is applied to the sampled network.

This design poses many challenges to traditional architectures like GPU. First, BFS creates irregular memory access patterns, thread divergence, and load imbalance, which are challenging for SIMD architectures. Second, their ego network sampling algorithm requires checking connectivity between sampled vertices and all existing vertices. Without specialized data structures like hash tables, this leads to high lookup overhead. Additionally, since we only need the inference result of one vertex, we only need to compute for vertices in a 2-hop neighborhood. However, this conditional computation is hard to implement on GPUs.

As a follow-up study, we can revise the program developed for the IARPA evaluation to study the convergence behavior of this model, proving that the design adequately performs vertex classification. Then, we can study the model on a traditional system with CPUs and GPUs: either running sampling on the CPU and transferring the results to the GPU or using optimized GPU libraries to perform both operations. This would allow us to conduct an end-to-end comparison of traditional systems with UpDown, which will shed light on the challenges involved in implementing these novel network designs.

Overall, we believe these problems reflect that neural network workloads are not necessarily dense and regular: they might require performing complex data manipulation, conditional behaviors, and expensive queries, which are suboptimal in SIMD systems. However, since GPUs are the dominant solution for machine learning acceleration, it has precluded the development of models with undesirable behaviors. With UpDown, there are opportunities to explore larger design space, which can result in the creation of more efficient models.

REFERENCES

[1] cublas. *NVIDIA*, 2023.

[2] Nvidia cudnn. *NVIDIA*, 2023.

[3] Nvidia h100 tensor core gpu architecture. *NVIDIA*, 2023.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow, Large-scale machine learning on heterogeneous systems, November 2015.

[5] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L. Abellán, Yash Ukidave, Ajay Joshi, John Kim, and David Kaeli. Gnnmark: A benchmark suite to characterize graph neural network training on gpus. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–23, 2021.

[6] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2022.

[7] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, Yunji Chen, and Yuan Xie. Rubik: A hierarchical architecture for efficient graph learning, 2020.

[8] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Velickovic. Principal neighbourhood aggregation for graph nets. *CoRR*, abs/2004.05718, 2020.

[9] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172, New York, NY, USA, 1969. Association for Computing Machinery.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[12] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *CoRR*, abs/1903.02428, 2019.

[13] Tong Geng, Ang Li, Tianqi Wang, Chunshu Wu, Yanfei Li, Antonino Tumeo, and Martin C. Herbordt. UWB-GCN: hardware acceleration of graph-convolution-network through runtime workload rebalancing. *CoRR*, abs/1908.10834, 2019.

[14] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin C. Herbordt, Yingyan Lin, and Ang Li. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization, 2022.

[15] Pradeep Gupta. Cuda refresher: The cuda programming model. *NVIDIA*, 2023.

[16] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[17] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[19] Lei He. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *CoRR*, abs/1909.00155, 2019.

[20] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[21] George Karypis and Vipin Kumar. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. 01 1995.

[22] Kevin Kiningham, Philip Levis, and Christopher Ré. Grip: A graph neural network accelerator architecture. *IEEE Transactions on Computers*, 72(4):914–925, 2023.

[23] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[24] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[25] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating sparse matrix multiplication with adaptive dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 747–761, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Shengwen Liang, Cheng Liu, Ying Wang, Huawei Li, and Xiaowei Li. Deepburning-gl: an automated framework for generating graph neural network accelerators. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *CoRR*, abs/1912.01703, 2019.

[28] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70, 2020.

[29] Andronicus Rajasukumar, Jiya Su, Yuqing, Wang, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Tianchi Zhang, Jianru Ding, Wenyi Wang, Ziyi Zhang, Moubarak Jeje, Henry Hoffmann, Yanjing Li, and Andrew A. Chien. Updown: Programmable fine-grained events for scalable performance on irregular applications, 2024.

[30] Andronicus Rajasukumar, Tianchi Zhang, Ruiqi Xu, and Andrew A. Chien. Updown: A novel architecture for unlimited memory parallelism. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '24, page 61–77, New York, NY, USA, 2024. Association for Computing Machinery.

[31] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1099–1112, 2023.

[32] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

[33] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[34] Tianyi Sun, Andrew Hands, and Risi Kondor. P-tensors: a general formalism for constructing higher order message passing networks, 2023.

[35] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[36] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019.

[37] Yuqing Wang, Andronicus Rajasukumar, Tianshuo Su, Marziyeh Nourian, Jose M Monsalve Diaz, Ahsan Pervaiz, Jerry Ding, Charles Colley, Wenyi Wang, Yanjing Li, David F. Gleich, Hank Hoffmann, and Andrew A. Chien. Updown: Programmable fine-grained events for scalable performance on irregular applications, 2024.

[38] Felix Wu, Tianyi Zhang, Amauri Holanda de Souza Jr., Christopher Fifty, Tao Yu, and Kilian Q. Weinberger. Simplifying graph convolutional networks, 2019.

[39] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.

[40] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.

[41] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Hygcn: A GCN accelerator with hybrid architecture. *CoRR*, abs/2001.02514, 2020.

[42] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[43] Zhilin Yang, William W. Cohen, and Ruslan Salakhutdinov. Revisiting semi-supervised learning with graph embeddings. *CoRR*, abs/1603.08861, 2016.

[44] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *CoRR*, abs/1806.08804, 2018.

[45] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. Graphsaint: Graph sampling based inductive learning method. *CoRR*, abs/1907.04931, 2019.

[46] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. Gamma: leveraging gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 687–701, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *CoRR*, abs/1802.09691, 2018.

[48] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, page 261–274. IEEE, February 2020.

[49] Roman Zubatyuk, Justin S. Smith, Jerzy Leszczynski, and Olexandr Isayev. Accurate and transferable multitask prediction of chemical properties with an atoms-in-molecules neural network. *Science Advances*, 5(8):eaav6490, 2019.