CSC 485B/586C GPU Computing - Final Report

Sadie Johansen
University of Victoria
Victoria, Canada
adj@uvic.ca

Erfan Golpour
University of Victoria
Victoria, Canada
erfangolpour@uvic.ca

Gaganjot Kaur
University of Victoria
Victoria, Canada
gaganjotkaur@uvic.ca

Abstract

Sparse matrix multiplication (*SpMM*) is central to Graph Neural Network (GNN) workloads, particularly for large and irregular graphs. In this study, we compare multiple GPU-based SpMM implementations—including PyTorch, CuPy, and custom PyCUDA kernels—and propose a *BFS-based partitioning pipeline* tailored to GNN computations. While breadth-first search has been widely used for partitioning in distributed systems, our approach adapts BFS expansions to enforce practical edge limits per cluster, matching the memory and concurrency characteristics of modern GPUs.

Once clusters are formed, each submatrix (adjacency plus features) is processed via an *edge-centric* SpMM kernel. We exploit GPU features such as multiple streaming multiprocessors (SMs), available shared memory, and the potential for `tensor_core` operations on compatible architectures. Additionally, we apply pipeline parallelization to overlap kernel execution with data transfers, using multiple CUDA streams to reduce idle GPU time.

Experimental results—covering Erdős–Rényi, scale-free, and real-world graphs and large matrices—demonstrate that our BFS-driven pipeline can match or exceed performance from well-optimized frameworks under certain conditions. We conclude by outlining strategies to integrate this partitioning approach into multi-GPU or distributed settings, highlighting the practical benefits of BFS-centric decompositions for large-scale GNN applications.

Keywords

GPU, GNN, Sparse Matrix Multiplication, CUDA

ACM Reference Format:
Sadie Johansen, Erfan Golpour, and Gaganjot Kaur. 2026. CSC 485B/586C GPU Computing - Final Report. In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Efficient computation is paramount in the field of Graph Neural Networks (GNNs), especially when dealing with large-scale graphs. Sparse matrix multiplication is a core operation in GNNs, used extensively in message passing and feature aggregation. Leveraging the parallel processing capabilities of Graphics Processing Units (GPUs) can significantly accelerate these computations. However, optimizing sparse matrix operations on GPUs presents challenges due to irregular memory access patterns and the sparsity of data.

This project compares different implementations of sparse matrix multiplication on GPUs, utilizing PyTorch, CuPy, and custom implementations in PyCUDA (including a BFS-based version). By evaluating these methods across various graph types, we seek to identify strategies that maximize computational efficiency and resource utilization. Furthermore, we have explored incorporating network decomposition techniques to improve performance in distributed GPU environments.

In this report, we outline our problem statement, review relevant literature, describe our methodology and experimental setup, present preliminary results, and discuss our next steps toward achieving our project goals.

1. **PyCUDA Implementations**: Utilizes PyCUDA to write custom CUDA kernels in Python.
2. **PyTorch Implementation**: Employs PyTorch's built-in sparse tensor operations with GPU acceleration.
3. **CuPy Implementation**: Leverages CuPy's NumPy-like syntax for GPU arrays and sparse operations.

Our experiments are conducted on a diverse set of graphs, including synthetic graphs generated using the Erdős-Rényi and Scale-Free models, and real-world graphs such as the Karate Club, Les Misérables networks, social circles graph from Facebook(Meta) and Condense Matter Collaboration Network from Arxiv. We measure processing time and GPU memory usage to evaluate the performance of each method.

Furthermore, we emphasize our plans to incorporate network decomposition techniques, specifically hierarchical decompositions with $O(\log n)$ levels, where each level contains clusters formed by Breadth-First Search (BFS) up to a certain distance. These decompositions are intended to optimize computations in distributed GPU systems by improving data locality and enabling parallel processing of subgraphs. We have taken a step in this direction by implementing this graph decomposition strategy on a single GPU.

2 Related Work

Efficient sparse matrix multiplication on GPUs has been extensively studied due to its significance in scientific computing and machine learning. Several important directions include:

Inductive Learning and Sampling. GraphSAGE [7] introduced sampling-based approaches to efficiently learn node embeddings in large-scale graphs. This inspired many subsequent works that reduce computational overhead by focusing on local neighborhoods.

Graph Partitioning. Cluster-GCN [5] demonstrated that partitioning large graphs into smaller clusters can improve mini-batch training scalability. Similarly, advanced partitioning methods [8]

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

have been deployed to optimize BFS expansions on parallel architectures.

High-Performance BFS Locality. While BFS expansions are a standard technique in distributed graph analytics, their performance critically depends on memory locality. Prior work by Beamer et al. [4] highlights that substantial locality does in fact exist in BFS, and demonstrates how careful ordering can reduce cache misses and improve overall throughput. This insight aligns with our cluster-based BFS approach, which seeks to group nodes to preserve locality on modern GPUs.

Kernel Fusion. NeuGraph [10] proposed fusing multiple GNN operations into single GPU kernels, thereby reducing memory traffic and improving cache locality. Our BFS-based approach shares a similar goal of reducing overhead via load-balancing subgraphs on the GPU.

Multi-Level Optimizations and Tiling. Recent work has explored tile-based algorithms for mixed dense-sparse matrix multiplication [9], as well as more general tile-based SpMM routines [13]. Such methods dynamically switch between dense tiling and CSR-based multiplication based on submatrix density. Our approach follows a similar logic, enabling each extracted cluster to pick the most suitable SpMM kernel.

Sparse GPU Libraries. cuSPARSE [11] has long been the de facto standard for optimized GPU-based sparse operations. Frameworks like CuPy and PyTorch build upon these kernels, though specialized or fused kernels (e.g., BFS expansions + partial feature calculations) can often outperform generic library calls in certain use cases.

High-Performance Graph Processing. Gunrock [12] is a well-known CUDA library specialized in GPU graph analytics, offering optimized primitives (e.g., BFS, PageRank, SSSP) and a frontier-based abstraction layer. Such work demonstrates the effectiveness of load-balancing and queue-based designs for GPU graph traversal, aligning closely with the BFS-based expansions in our partitioning approach. In addition, another recent framework, GraphBLAST [14], adopts a linear-algebraic interface for GPU-based graph processing (including BFS and SpMM). By expressing graph operations in terms of matrix-matrix multiplication primitives, GraphBLAST achieves high performance on sparse datasets, similar to our BFS-centric pipeline. Integrating concepts from GraphBLAST—such as fine-tuned sparse kernels—could further reduce memory overhead in our system.

Taken together, these works highlight the breadth of techniques—graph sampling, partitioning, kernel fusion, tiling, specialized sparse libraries, and frontier-based GPU graph processing—that can substantially improve GPU-based sparse matrix multiplication, especially in the context of GNNs. Our experiments build on these insights, focusing on a BFS-based graph decomposition approach and evaluating different kernel implementations (PyCUDA, PyTorch, and CuPy) on diverse graph benchmarks.

3 Background

3.1 Sparse Matrix Multiplication in GNNs

GNNs operate by iteratively updating node representations through the aggregation of information from neighboring nodes. This process often involves multiplying a *sparse* adjacency matrix $\mathbf{A}$ with a *dense* node representation matrix $\mathbf{H}^{(l)}$ (sometimes also called a feature matrix at layer l), followed by a weight matrix $\mathbf{W}^{(l)}$:

$$\mathbf{H}^{(l+1)} = \sigma\big(\mathbf{A}\,\mathbf{H}^{(l)}\,\mathbf{W}^{(l)}\big),$$

where:

- $\mathbf{A}$ is the (sparse) adjacency matrix of the graph,
- $\mathbf{H}^{(l)}$ is the current node representation (feature) matrix,
- $\mathbf{W}^{(l)}$ is the layer-specific weight matrix,
- σ is an activation function (e.g., ReLU, Softmax, or Sigmoid).

Efficient computation of the product $\mathbf{AH}^{(l)}$ is crucial for large-scale graphs to avoid excessive memory usage and long runtimes.

3.2 GPU Computing and Optimizations

GPUs are well-suited for parallel computations due to their numerous processing cores. However, achieving high performance requires careful optimization:

- **Memory Hierarchy Utilization**: Reducing accesses to slower global memory by maximizing the use of faster shared memory and caches.
- **Increasing L1 Cache Hits**: Organizing data and computations to improve the likelihood of data being found in the L1 cache, reducing memory latency. This is a particularly challenging approach.
- **Coalesced Memory Access**: Arranging data so that consecutive threads access consecutive memory locations, minimizing latency.
- **Thread Divergence Minimization**: Designing kernels to minimize divergence, ensuring that threads within a warp execute the same instructions.

4 Problem Description

The primary challenge addressed in this project is the efficient implementation of sparse matrix multiplication in GNNs on GPUs. Specifically, we aim to:

- **Compare Implementations**: Evaluate different GPU-based methods for sparse matrix multiplication in terms of processing time and memory usage.
- **Optimize Performance**: Apply GPU optimization techniques to improve computational efficiency via increasing parallel performance.
- **Handle Diverse Graphs**: Test implementations across various graph types and sizes to assess scalability.
- **Prepare for Distributed Systems**: Develop strategies that can be extended to distributed GPU environments using network decompositions.

5 Methodology

5.1 Overview

We implement and compare different methods for performing sparse matrix multiplication on GPUs using existing libraries as well as our own PyCUDA implementations. Each method offers unique advantages and poses distinct challenges which we explore in this project. We describe each method and explain how its design leverages GPU architectures for improved performance.

5.2 Hierarchical Network Decomposition

Network decomposition involves partitioning a graph into clusters to facilitate parallel processing. A common strategy is to build a *hierarchical decomposition* with $O(\log n)$ levels, where level i consists of clusters formed by breadth-first searches (BFS) up to a distance of 2^i. This ensures that all nodes are covered in at least one cluster per level, providing multiple "views" of the graph at different scales.

Multi-Layer BFS Implementation. In our code, we use a `multi_layer_bf` routine (see the snippet provided) that launches a GPU-based BFS kernel for a user-defined or calculated based on GPU properties distance limit depending on the implementation. We repeatedly pick "seed" nodes among the unvisited set and expand outward until reaching distance d or a global edge limit. This process creates overlapping clusters (via "ghost expansions"), guaranteeing node coverage while bounding each cluster's size. The BFS kernel also prunes out-of-bound neighbors and uses atomic operations to record node visitation, enabling efficient parallel expansions.

Cluster Sparsity and Tile-Dense vs. CSR Kernels. Once clusters are identified, each subcluster is extracted into a submatrix of the original adjacency (or feature) matrix. Depending on each submatrix's dimension and density, we switch methods as in [9] and either:

- *Use a tile-based dense-kernel* if the submatrix is relatively small and sufficiently dense, benefiting from shared memory tiling and contiguous memory access.
- *Use a CSR-based SpMM kernel* otherwise, which handles sparser submatrices more efficiently by skipping zero-valued entries.

This dual strategy harnesses the strengths of both dense tiling and CSR-based approaches, improving utilization of GPU resources.

Erdős–Rényi Example and Randomness. With highly randomized graphs like Erdős–Rényi, the multi-layer BFS expansions typically produce clusters that still appear fairly random in composition, as illustrated in Figure 2. Nonetheless, this hierarchical decomposition helps ensure each node is included in multiple BFS expansions (at various depth scales), fostering both overlap for redundancy and controlled submatrix size for parallel processing.

Ultimately, these *multi-layer BFS* clusters form the foundation of our hierarchical decomposition approach, balancing global coverage, local connectivity, and GPU-friendly submatrix extraction for mixed dense/sparse kernels.

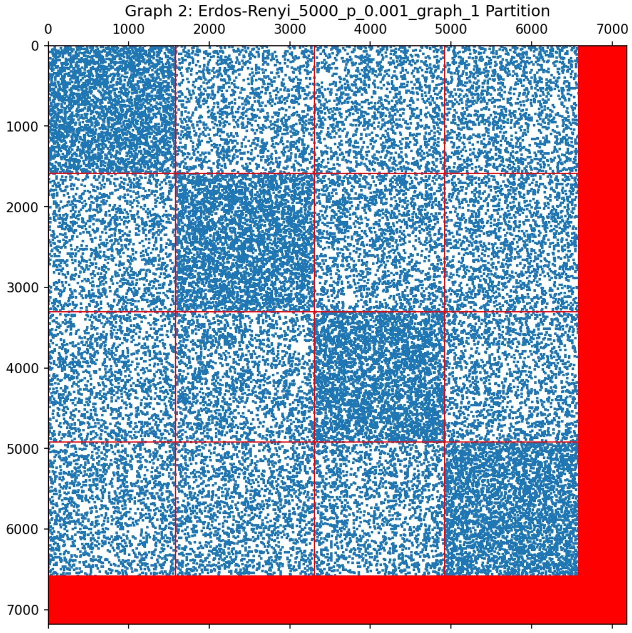

Figure 1: Result of a layered decomposition.

5.3 Spectral -> Jaccard -> BFS Decomposition

In order to efficiently multiply a sparse graph matrix **A** by a feature matrix **F** (often denoted SpMM: *Sparse Matrix-Matrix Multiplication*), we propose a two-phase process:

(1) **Graph Decomposition:** Partition the graph into a set of smaller subgraphs (clusters).
(2) **Submatrix SpMM:** Perform SpMM on each cluster's submatrix independently and merge the partial results.

This approach leverages GPU concurrency and better caching to reduce the overhead of processing large, irregular sparse structures at once and allows for processing graphs that do not fit in memory. Below, we describe each phase in detail.

5.3.1 *Decomposition Step: Spectral+BFS Clustering.*

1) Spectral Initialization. We begin by computing the graph Laplacian **L** and extracting the Fiedler vector (the eigenvector corresponding to the second-smallest eigenvalue). This vector provides a global, coarse partition: nodes whose Fiedler component is above a threshold are labeled `1`, and others `0`.

2) Batched Edge Weight Calculation. To refine our local expansions, each edge gets a "weight" representing node similarity (e.g., a Jaccard-like intersection over union of neighbors). Because large graphs can trigger kernel timeouts, we implement a batched approach in `compute_edge_weights_batched`, splitting the vertex set into multiple slices and synchronizing after each kernel invocation. The kernel itself (`compute_edge_weights_kernel`) computes the intersection size between neighbors of two nodes, normalizes by their union, and stores this as an edge weight.

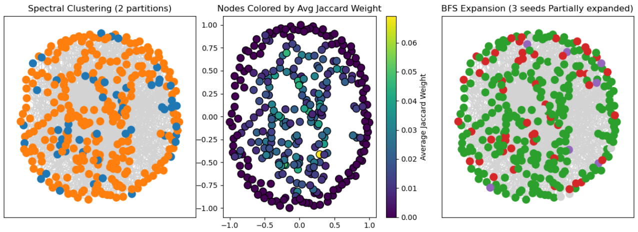

Figure 2: Stages of clustering process.

3) Balanced BFS Partitioning. Starting from the initial spectral labels, each cluster *seed* attempts to grow via a GPU BFS expansion, as defined in `balanced_bfs_kernel`. Key elements of the BFS logic:

- **BatchQueue Structure:** A specialized queue is split into multiple fixed-size "batches." Each thread uses atomic operations to enqueue or dequeue nodes in a less-contended manner. This structure is declared in `PARTITION_KERNELS` with routines `init_batch_queue`, `queue_push_batch`, and `queue_pop_batch`.
- **Edge-Limit Stopping Condition:** Each cluster enforces a global `max_edges_per_cluster` threshold to limit BFS expansion. Once a cluster consumes its allowance of edges, BFS terminates for that seed to ensure we do not exceed memory constraints.
- **Edge-Weight Gatekeeping:** A node is only claimed by a cluster if its connecting edge weight exceeds a threshold (e.g., 0.5). This enforces a higher "cohesion" within each discovered cluster.

By iterating this process over a list of candidate seed nodes, we ultimately assign (or skip) the majority of nodes in the graph, producing a set of subgraphs that each has a bounded number of edges.

4) Extraction of Submatrices. After the BFS expansions finalize all clusters, the host code (e.g., `extract_submatrices` or `parallel_extract_submatrix`) gathers each cluster's adjacency rows and columns into smaller CSR submatrices. Optionally, the relevant rows of $\mathbf{F}$ (the feature matrix) are also extracted, forming submatrices $\mathbf{F}_c$ for each cluster c. If any cluster is too small or fails checks, it may be discarded or merged.

5.3.2 Submatrix SpMM Optimization. Once we have a set of small submatrices $(\mathbf{A}_c, \mathbf{F}_c)$, each cluster's SpMM operation is straightforward to parallelize:

- **Reduced Kernel Runtime:** Each submatrix $\mathbf{A}_c$ is denser or at least smaller than the full matrix, leading to better GPU occupancy and fewer memory stalls.
- **Thread-Block Tuning:** Because each submatrix is smaller, we can adjust block and grid dimensions to achieve coalesced memory reads. This is handled in code by `calculate_grid_size` and associated logic, ensuring we do not exceed the device's `max_threads_per_block`.
- **Parallel Merging:** The partial products $C_c = \mathbf{A}_c \times \mathbf{F}_c$ (computed by `sparse_matmul_kernel`) are gathered back on the CPU. For overlapping boundary nodes (those belonging to multiple clusters), we average the partial feature results.

Overall, this decomposition-based SpMM pipeline (Spectral+BFS partitioning followed by submatrix SpMM) capitalizes on:

1. *Limiting Per-Cluster Size:* Ensuring that BFS expansions do not exceed a set number of edges keeps the memory footprint predictable and avoids GPU-out-of-memory errors.
2. *GPU Concurrency:* Multiple submatrices can be processed in parallel, either by the same GPU kernel with different queues or across multiple kernels, thereby improving throughput for large graphs.
3. *Feature Locality:* Because the partitioning BFS uses locally computed edge weights (e.g., Jaccard similarity), nodes in the same cluster tend to share many neighbors. This significantly increases the likelihood that these nodes will access or update the same feature matrix rows/columns, thereby enhancing cache reusability.

In summary, the decomposition step (Section **3) Balanced BFS Partitioning**) ensures each submatrix is a manageable "chunk" for the GPU, and the subsequent SpMM phase can be run efficiently on these chunks, improving both performance and memory usage compared to a naive, single-shot SpMM on the entire graph. Implementations were done that used a range of combinations of these steps with the final tested implementation being a simpler approach that skips Spectral Clustering.

5.4 PyTorch Implementation

5.4.1 Algorithm Design. PyTorch provides built-in support for sparse tensors and GPU acceleration. For operations on sparse tensors, PyTorch relies on cuSPARSE, a library within the CUDA toolkit designed to handle sparse matrix computations on NVIDIA GPUs efficiently. cuSPARSE provides optimized routines for sparse matrix-vector and sparse matrix-matrix operations, leveraging GPU parallelism to improve performance. PyTorch integrates sparse tensor operations with its autograd engine, allowing gradients to be computed through sparse matrix operations during backpropagation. This makes it particularly useful for training GNNs. The algorithm involves:

Algorithm 1 Sparse Matrix Multiplication using PyTorch

```
Input: Sparse adjacency matrix $\mathbf{A}$ and feature matrix $\mathbf{H}$.
Convert $\mathbf{A}$ and $\mathbf{H}$ to PyTorch sparse tensors on the GPU.
Perform multiplication using PyTorch's sparse matrix multiplication function:
    $\mathbf{C} = \mathbf{A} \cdot \mathbf{H}$
Output: Resultant matrix $\mathbf{C}$.
```

5.4.2 GPU Optimization Benefits.

- **High-Level Abstraction**: Simplifies implementation while leveraging optimized backend operations.
- **Automatic Differentiation**: Facilitates integration with autograd for GNN model training.
- **Efficient Memory Management**: PyTorch handles data transfers and memory allocations efficiently.

5.5 CuPy Implementation

5.5.1 Algorithm Design. CuPy offers NumPy-like syntax for GPU arrays and supports sparse operations using cuSPARSE. The algorithm is as follows:

Algorithm 2 Sparse Matrix Multiplication using CuPy

```
Input: Sparse matrices $\mathbf{A}$ and $\mathbf{H}$.
Convert $\mathbf{A}$ and $\mathbf{H}$ to CuPy sparse CSR matrices.
Perform multiplication using CuPy's dot function:
    $\mathbf{C} = \mathbf{A} \cdot \mathbf{H}$
Output: Resultant matrix $\mathbf{C}$.
```

5.5.2 GPU Optimization Benefits.

- **Backend Optimizations**: CuPy leverages cuSPARSE, which includes highly optimized routines for sparse computations.
- **Ease of Use**: Similar syntax to NumPy allows for quick implementation and prototyping.
- **Memory Efficiency**: CuPy manages GPU memory effectively, reducing overhead.

6 Pipeline Parallelization Setup

6.1 Algorithm Overview

The targeted objective of this project is to integrate network decomposition techniques to enhance performance in distributed GPU environments, particularly when training large-scale graphs across multiple GPUs. As an initial step and foundational approach, we have focused on implementing a graph decomposition strategy on a single GPU. This implementation serves as a simplified version of a broader algorithm. We have implemented many algorithms with varying parallelization strategies for sparse matrix multiplication in Graph Neural Networks, combining graph partitioning, pipelining data transfer, and GPU parallelization to experiment.

6.2 Stream of Decomposed Clusters

We first decompose the graph into clusters that are manageable in size and localized around high-weight edges. Specifically, we rely on two key steps:

(1) **GPU-Accelerated Partitioning:** A METIS/BFS hybrid routine (e.g., `create_clusters_metis_bfs_gpu`) or a spectral+balanced-BFS approach (see `gpu_partition_graph`) iteratively identifies dense subgraphs by expanding from seed nodes. We enforce a maximum edge threshold (e.g., `max_edges_per_cluster`) to cap each cluster's size.

(2) **Dynamic Cluster Sizing:** In practice, parameters such as

$$\max\_edges\_per\_cluster = \log_2(n) \times \frac{|E|}{n},$$

can be used as a heuristic for controlling cluster size. The BFS expansions also incorporate "ghost" nodes for boundary completeness, ensuring no abrupt truncation of edges that connect to unvisited nodes.

As soon as each cluster is formed, we extract its corresponding submatrices ($\mathbf{A}_c$ from the adjacency and $\mathbf{F}_c$ from the feature matrix) and push these submatrices onto a work queue. Multiple clusters may overlap (i.e., a node can appear in more than one cluster) to guarantee coverage and finer-grained parallel opportunities.

6.3 Multi-Level Parallelization

Our approach supports two complementary levels of parallelism:

6.3.1 Level 1: Cluster-Level Parallelization.

- **Concurrent Cluster Processing:** Each newly formed cluster is submitted to a pool of threads (or processes). A `ThreadPoolExecutor` (or `multiprocessing` pool) may manage this stage, where each cluster's extraction (`extract_submatrices` or `parallel_extract_submatrix`) and subsequent GPU multiplication is handled independently.
- **Adaptive Batching:** In the `GPUPipeline` class, clusters are processed in configurable "batches" to avoid excessive kernel launches and to adapt to GPU resource constraints. The batch size can be set based on the number of SMs, threads per SM, or memory considerations.
- **Flexible Worker Count:** The total number of CPU workers (threads or processes) can be tuned to match the CPU resources available for cluster extraction or multi-GPU orchestration.

6.3.2 Level 2: GPU-Level Parallelization.

- **Within-Cluster SpMM:** For each cluster submatrix $(\mathbf{A}_c, \mathbf{F}_c)$, the pipeline launches a GPU kernel (e.g., `sparse_matmul_kernel`) that parallelizes the multiplication across thread blocks. The blocks/threads layout is typically:

  $$\text{Grid: } \left(\lceil \tfrac{n_{\text{out}}}{32} \rceil,\ \lceil \tfrac{n_{\text{in}}}{32} \rceil,\ 1\right), \quad \text{Block: } (32,\ 32,\ 1),$$

  where $n_{\text{in}}, n_{\text{out}}$ represent the submatrix dimensions. Each thread computes one element (or a small tile) of the output, performing memory coalescing and, if applicable, shared-memory tiling or BFS-style adjacency lookups.
- **Shared GPU Contexts and Streams:** The `CUDAKernelManager` maintains a context and multiple CUDA streams so different clusters can be launched in parallel if hardware resources permit. This complements the cluster-level concurrency, allowing multiple submatrix multiplications to overlap on the GPU.
- **Adaptive Sparse vs. Dense Kernel:** For submatrices that are large but sparse, the code uses a CSR-based kernel; for smaller or denser submatrices, a tile-based approach is used (as described in references like [9]) to improve cache locality.

6.4 Scaling Beyond a Single GPU

Since the pipeline processes each cluster independently, we can naturally extend this approach to multi-GPU systems or even distributed clusters:

- **Multi-GPU Scheduling:** Each GPU runs its own `CUDAKernelManager` instance, and the CPU-level thread pool dispatches submatrices/clusters to whichever GPU becomes available.
- **Hierarchical Partition:** Large graphs are first partitioned globally (possibly on the CPU) into broad regions, which are then subdivided by the BFS expansions on multiple GPUs.

Each GPU focuses on sub-partitions, reducing inter-GPU communication.

- **Result Merging:** Final results (e.g., partial multiplications) are gathered back on the CPU, or aggregated across GPUs, ensuring correctness even if clusters overlap.

This design thus combines:

(1) *A BFS-centric clustering strategy* to keep clusters within size/memory bounds,
(2) *Multi-level parallelization* (clusters on the CPU, kernels on the GPU), and
(3) *Scalable multi-GPU usage* for very large graphs.

All these features enable high performance on large-scale problems, where a single GPU alone would be insufficient to process the entire adjacency matrix at once.

6.5 Pipeline Optimization

Data transfer and computation are overlapped in our current simplified version using a pipeline approach:

$$\text{Timeline:}\quad \underbrace{T_1^{load}}_{\text{Cluster 1}} \rightarrow \underbrace{T_1^{compute}}_{\text{Cluster 1}} \parallel \underbrace{T_2^{load}}_{\text{Cluster 2}} \rightarrow \underbrace{T_2^{compute}}_{\text{Cluster 2}} \parallel \underbrace{T_3^{load}}_{\text{Cluster 3}} \rightarrow \cdots \tag{1}$$

Where:

- T_i^{load}: Time to transfer cluster i data to GPU
- $T_i^{compute}$: Time to compute matrix multiplication for cluster i
- $\|$ indicates parallel execution

Depending on the cluster calculation parameters and graph size, multiple clusters can be parallelized as separate processes to pack more clusters into the GPU at once, or to allow for interweaving calculations during process downtime.

6.6 Result Combination

Results from different clusters are combined using weighted averaging in this simplified approach while exploring decomposition strategies:

$$R_{ij} = \frac{\sum_{c \in C} R_{ij}^c}{|C_{ij}|} \tag{2}$$

where R_{ij}^c is the result for node i feature j from cluster c, and $|C_{ij}|$ is the number of clusters containing node i.

6.7 Time Complexity

For this simplified test we have:

- Graph Decomposition: $O(|V| + |E|)$
- Per-Cluster Processing: $O(\frac{nnz(A)}{k} \cdot \frac{nnz(B)}{k})$
- Total Processing: $O(\frac{nnz(A) \cdot nnz(B)}{p})$

where k is the number of clusters, p is the degree of parallelization and nnz is the number of non-zero elements.

6.8 Memory Complexity

Peak memory usage per cluster is bounded by:

$$M_{cluster} = O(\frac{|V|}{k} \cdot |F| + \frac{|E|}{k}) \tag{3}$$

where $|F|$ is the feature dimension and k is the number of clusters.

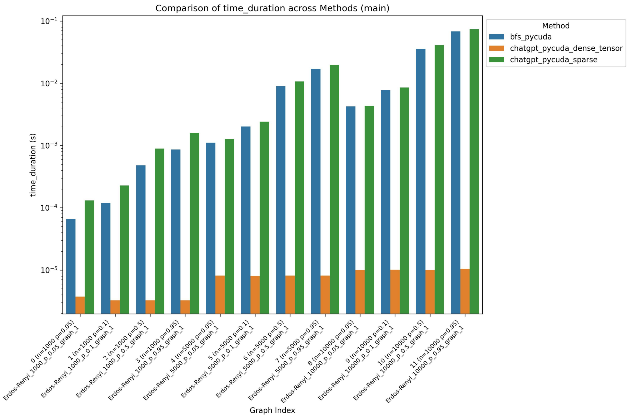

Figure 3: Time Comparison between Dense Multiplication using Tensor Cores vs BFS Clustering vs Sparse Multiplication

7 Exploiting Tensor Cores for SpMM

Tensor Cores are a specialized hardware feature in NVIDIA GPUs designed to accelerate mixed-precision matrix computations, particularly in deep learning and high-performance computing. By supporting operations with half-precision floating-point numbers (FP16) while leveraging higher precision accumulators (e.g., FP32), Tensor Cores achieve significant improvements in computational throughput. This section explores the application of Tensor Cores to Sparse Matrix Multiplication (SpMM), evaluates their performance against our existing CUDA-based implementations, and discusses potential integrations into our workflows.

7.1 Experimentation

We implemented a Tensor Core-based matrix multiplication routine by adapting NVIDIA's sample code for warp-level 16×16×16 matrix multiplication derived from NVIDIA's CUDA Programming Guide example [1]. This routine leverages the Warp Matrix Multiply-Accumulate (WMMA) API to maximize hardware utilization. Our goal was to compare the performance and accuracy of Tensor Core-based methods against standard CUDA implementations developed earlier in this project, including dense multiplication and sparse multiplication with BFS clustering.

The experiments were conducted on a modern NVIDIA GeForce RTX 3070 Laptop GPU with Tensor Core support. Input matrices were designed to emulate realistic sparsity patterns, ensuring fair comparison with our BFS-based sparse implementations.

7.2 Performance Analysis

7.2.1 Speed. The Tensor Core implementation demonstrated a performance advantage over most other implementations.

Figure 3 highlights the performance comparison across implementations, showcasing the Tensor Core's significant speedup.

This speed advantage arises from the hardware's ability to execute multiple FP16 operations per clock cycle, coupled with reduced memory bandwidth requirements due to the smaller data size.

7.2.2 *Accuracy.*

While the speed benefits are clear, accuracy is a notable tradeoff. When compared against CPU-based FP64 results computed using NumPy, discrepancies were observed:

```
Max absolute difference: 0.8378889044746756
CPU result - min: 0.0
max: 4.8307543754577637
mean: 1.24893857538700104
GPU result - min: 0.0
max: 5.6686432799324393
mean: 1.29842385732104053
```

These results were obtained from multiplying two square matrices of shape (1000, 1000) and comparing the results with what Numpy reports. The observed inaccuracies stem from the FP16 precision, which has a limited dynamic range and is susceptible to rounding errors. As noted in NVIDIA's documentation [?], this is an inherent tradeoff of mixed-precision computing.

7.3 Pros and Cons of Tensor Cores

7.3.1 *Advantages.*

- Performance Boost: Tensor Cores offer a significant speedup for compute-intensive matrix operations.
- Reduced Memory Usage: FP16 data requires half the memory of FP32, enabling larger problem sizes to fit in memory.
- Energy Efficiency: Lower precision computations typically consume less power.

7.3.2 *Disadvantages.*

- Precision Loss: FP16's limited precision may not suffice for all applications, particularly those requiring high numerical stability.
- Integration Complexity: Leveraging Tensor Cores requires adherence to specific data alignment and memory layout requirements, adding complexity to implementation.

7.4 Integration into Current Workflows

Tensor Cores can be integrated into our existing implementations with minimal disruption by replacing the current CUDA kernels used for matrix multiplication. The WMMA API provides a straightforward mechanism to achieve this substitution. However, the integration requires careful consideration of:

Precision Management: Ensuring that inputs can tolerate FP16 precision or implementing hybrid approaches where critical computations use higher precision.

Opportunity Cost: Tensor Cores share GPU resources with CUDA cores. Using Tensor Cores for SpMM might reduce the availability of CUDA cores for other parallel tasks, potentially creating bottlenecks.

7.5 Broader Implications and Opportunities

The shift towards mixed-precision computing aligns with trends in machine learning and high-performance computing, where performance gains often outweigh precision tradeoffs. By adopting Tensor Cores, we position our implementations to leverage future advancements in GPU hardware, including:

- Mixed-Precision Training: Facilitating integration with neural network workflows that are increasingly FP16-centric.
- Hybrid Precision Strategies: Combining FP16 and FP32 computations to balance accuracy and performance.

8 Experimental Setup

8.1 Datasets

We use both synthetic and real-world graphs to evaluate the performance of the implementations.

8.1.1 *Synthetic Graphs.*

Erdős-Rényi Graphs. Generated using the $G(n, p)$ model, where n is the number of nodes and p is the probability of edge creation. We vary n and p to create graphs with different sizes and sparsity levels [8].

Scale-Free Graphs. Created using the Barabási-Albert model, generating graphs with power-law degree distributions characteristic of many real-world networks.

8.1.2 *Real-World Graphs.*

Social circles: Facebook. includes data on users and their social connections, structured as a network graph (4039 nodes and 88234 edges). It contains "circles" (similar to friend lists or groups), along with anonymized attributes about the users, such as age, education, and work.[3]

Condense Matter Collaboration Network. Arxiv COND-MAT (Condense Matter Physics) collaboration network is from the e-print arXiv and covers scientific collaborations between authors papers submitted to Condense Matter category. If an author co-authored a paper with author j, the graph contains a undirected edge from i to j. If the paper is co-authored by k authors this generates a completely connected (sub)graph on k nodes. This dataset covers papers in the period from January 1993 to April 2003 and has 23133 nodes and 93497 edges.[2]

8.1.3 *SuiteSparse Matrix Collection.*

SuiteSparse Matrix Collection. It is a large and actively growing set of sparse matrices that arise in real applications. This collection serves as a benchmark for researchers and developers working on algorithms and software for sparse matrix computations. The matrices originate from domains such as computational science, engineering, optimization, machine learning, and network analysis, showcasing a wide variety of sizes, sparsity patterns, and structures. The collection has a total of 2893 matrices and includes metadata for each matrix such as name, group, size, nonzeros, kind, and date. We have calculated the sparsity of every matrix during preprocessing and stored it for filtering and downloading matrices from this collection for given size and sparsity ranges. [6]

8.2 Hardware and Software Environment

NVIDIA GTX 970, PyCUDA, PyTorch, CuPy

8.3 Performance Metrics

- **Processing Time**: Measured using high-resolution timers, ensuring synchronization with GPU operations.
- **GPU Memory Usage**: Monitored using CUDA's memory management APIs and custom memory tracking functions.

8.4 Measurement Methodology

To ensure fair comparisons:

- All implementations use the same data representations and formats.
- Experiments are repeated multiple times to account for variability
- GPU memory is cleared between runs to avoid residual data.
- GPU warmup is performed multiple times before executing operations on real data for a fair comparison.
- Synchronization functions are used to ensure accurate timing.

9 Results

9.1 Results on Synthetic Graphs

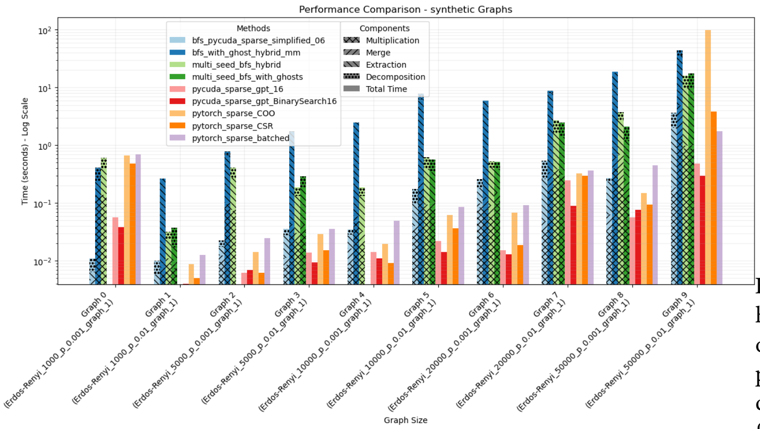

Figure 4: Sparse Matrix Multiplication Processing Times for Erdos-Renyi Graphs. Shows the approximate full times for various stages calculated with time.perf_counter().

To evaluate the performance of our BFS-based sparse matrix multiplication algorithm with multi-level parallelization, we conducted experiments on synthetic graphs generated using the Erdős–Rényi model. The graphs consisted of 1000,10000,20000,50000, and 100000 nodes with varying edge probabilities to simulate different sparsity levels. This allowed us to assess the algorithm's efficiency across a spectrum of graph densities.

Figure 4 presents preliminary runtime comparisons of multiple methods across these graphs. Although PyTorch's built-in routines are typically well-optimized for many workloads, we often observe our BFS-based approach outperforming PyTorch, particularly on certain graph structures or densities. We suspect this is due to additional overheads inherent in PyTorch's sparse handling, whereas our GPU-centric BFS kernels and partitioning strategies can better adapt to some graph topologies. Our implementations have also focused on Erdős-Rényi which allows for specialization whereas PyTorch is a more general framework and will likely far outperform our implementation on a wide range of graphs.

Initially, our `pycuda_sparse_bfs` approach determined the number of edges to explore using a formula that mixed node and edge counts. In practice, this led to reduced performance on both dense and sparse graphs because we were not utilizing available resources. By refocusing the expansion criteria on the actual edge count (rather than arbitrary ratios of nodes to edges) and properties of the GPU being used, we rebalanced the BFS expansions. This adjustment ensures that sparser graphs do not suffer from under utilization, while denser graphs do not exceed practical bounds during the BFS expansions.

Beyond refining the BFS threshold, we introduced an additional layer of parallelism by setting up a *pipeline* that overlaps data transfers with computation through *CUDA context switching* and *CUDA streams*. Concretely, our pipeline uses one stream to load the next cluster's data into GPU memory while another stream computes SpMM for the current cluster. This approach reduces idle periods and increases GPU occupancy. Moreover, we take advantage of the GPU constraints determined in our code, such as:

- **Max Threads per Block** (`max_threads_block`) to configure the BFS kernels' launch dimensions for optimal occupancy,
- **Available Memory** (`available_memory`) to size each cluster so that it fits into GPU memory along with any required feature data,
- **Number of SMs** (`num_sms`) to coordinate batch sizes and concurrency levels, ensuring balanced load across the GPU's streaming multiprocessors.

By tailoring BFS expansions and SpMM kernel launches to these hardware attributes, our pipeline can dynamically adjust block sizes, queue lengths, and concurrency, significantly improving throughput. Thus, the synergy of a carefully tuned BFS-based partitioning, combined with a GPU-aware pipeline design, delivers higher performance on a range of synthetic and real-world graph workloads.

9.2 Results on Real-World Graphs

Figure 5 demonstrates how different implementations perform on large graphs taken from the Stanford SNAP repository. Notice that these graphs are generally extremly dense (p=0.9+). Consequently, the PyTorch implementations, particularly chatgp_pytorch_sparse, perform notably better on dense graphs like facebook_combined.txt. This is unsurprising given that PyTorch is optimized for general-purpose workloads, including dense matrix computations, and leverages highly optimized CUDA kernels tailored for such data structures.

9.3 Results Limitations

While the current implementation shows promise, there are limitations to address. The formula for determining cluster sizes, though improved, is almost certainly not optimal for all graph types. Further research is needed to develop a more adaptive clustering strategy that can adjust to various graph structures and densities.

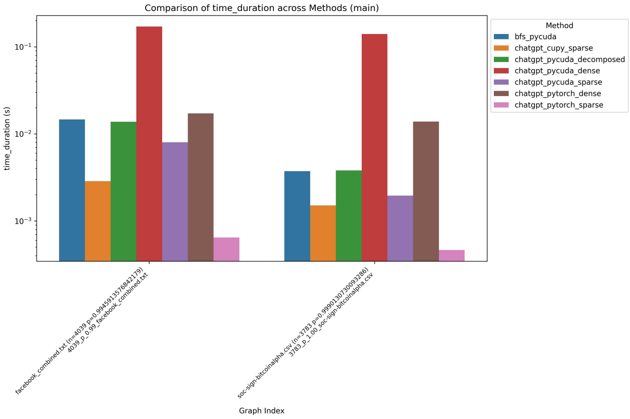

Figure 5: Processing Time on Large, Real-World Graphs

Combining results from multiple clusters introduces overhead, particularly when nodes belong to multiple clusters. Our current method uses weighted averaging to combine results, which is not not the most efficient approach. Exploring alternative methods for result aggregation will likely reduce this overhead.

We also attempted to enhance the algorithm by integrating compressed sparse row (CSR) and compressed sparse column (CSC) formats, aiming to optimize memory access patterns similar to cuSPARSE’s [11] block sparse row (BSR) format. However, implementing a hybrid CSR-CSC approach proved challenging due to the complexity of managing different data representations and the inconsistencies encountered in code generation using AI tools like ChatGPT and Claude.

10 Profiling and Performance Analysis

Profiling is a critical component in optimizing GPU-based algorithms, providing insight into performance bottlenecks and resource utilization. Our revised approach leverages Nvidia’s Nsight Compute CLI (NCU) in conjunction with the NVIDIA Tools Extension Library (NVTX) to gather in-depth metrics across all implementations, regardless of the backend (e.g., PyTorch, PyCUDA, C++ CUDA). This unified toolchain enhances the profiling precision and versatility compared to our initial approach, where profiling capabilities were fragmented by library-specific limitations.

10.1 Leveraging NCU for Unified Profiling

The adoption of NCU has streamlined our profiling capabilities, enabling comprehensive metric collection across all implementations. Key benefits of this tool include:

- **Unified Profiling Command:** NCU’s independence from the underlying framework (PyTorch, PyCUDA, etc.) allows for a single profiling command to be applied consistently across all implementations. This approach reduces dependencies on framework-specific profiling methods and enhances reproducibility.
- **Expanded Metric Collection:** By harnessing NCU, we can access a suite of advanced metrics previously unavailable to us, such as the average number of active warps per active cycle, DRAM throughput, kernel launch characteristics, etc. all of which provide valuable insight into the performance of our implementations. Figure **??** demonstrates one of such metrics. sm_cycles_active corresponds to the average number of cycles with at least one warp in flight.

These expanded metrics empower us to perform a more nuanced performance analysis, tailored to each implementation’s unique characteristics.

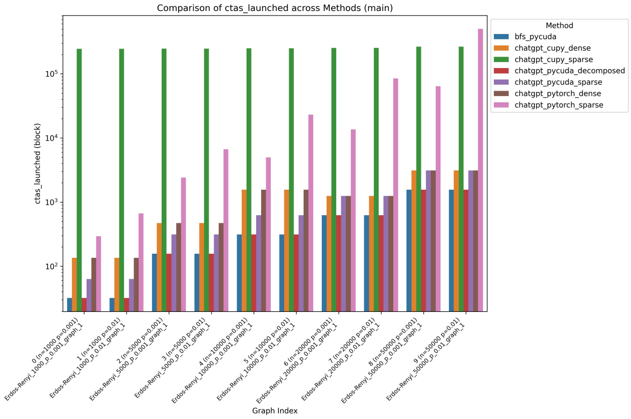

Figure 6: Comparison of CTAs Launched Across Configurations

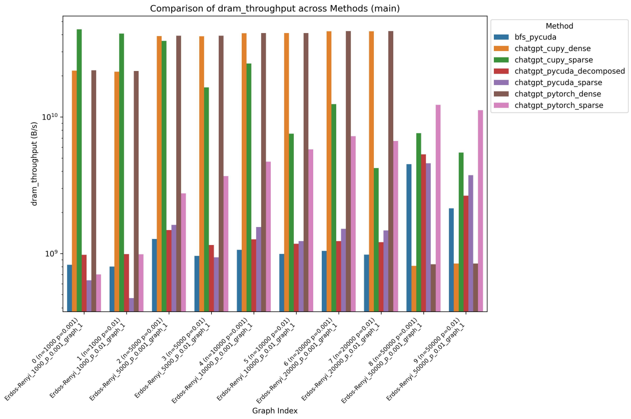

Figure 7: DRAM Throughput Comparison Across Configurations

10.2 Comparisons

The following results have been produced using a modern NVIDIA GeForce RTX 3070 Laptop GPU:

10.2.1 Key Observations:

- CTAs Launched (Figure 6): CuPy and PyTorch exhibit a higher number of CTAs (Cooperative Thread Arrays) launched compared to our implementation. This is expected as we did not fully utilize this feature.

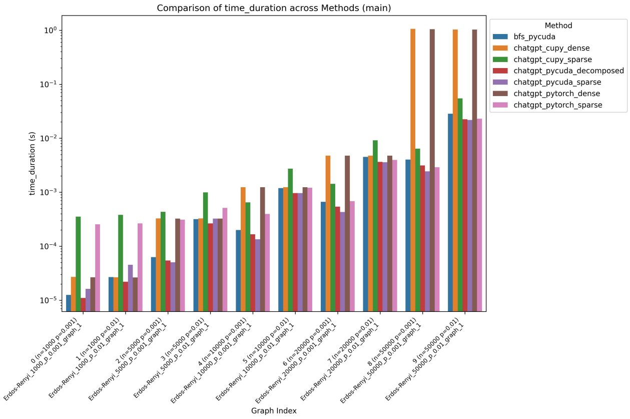

Figure 8: Execution Time Duration Comparison Across Configurations

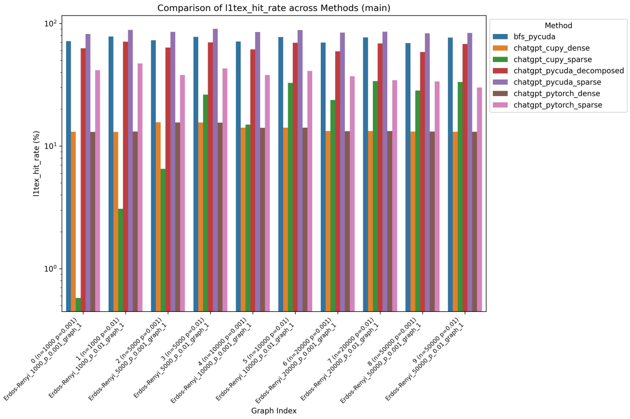

Figure 9: L1 TEX Cache Hit Rate Comparison Across Configurations

DRAM Throughput (Figure 7): The DRAM throughput for the CUDA-based approaches surpasses that of the PyTorch-based implementation for graphs with higher densities. Efficient memory access patterns in the CUDA kernels likely contribute to this improvement, minimizing bottlenecks caused by memory latency.
Execution Time Duration (Figure 8): Execution time for the optimized implementations is consistently lower than or comparable to the reference implementation (chatgpt_pytorch_sparse). This highlights the effectiveness of kernel decomposition and sparsity exploitation strategies in reducing computational overhead.
L1 TEX Cache Hit Rate (Figure 9): Our CUDA implementations maintain a comparable L1 TEX cache hit rate to that of the reference methods. This metric underscores the importance of well-structured memory access patterns in high-performance GPU programming.

These results demonstrate the advantages of tailored CUDA implementations over generic frameworks like PyTorch for specific workloads. By carefully optimizing kernel design, memory access, and thread allocation, significant performance gains can be achieved.

10.3 Enhanced Code Block Profiling with NVTX

We implemented NVTX annotations to further refine our profiling capabilities, dividing each script into three distinct execution blocks:

1. **Preparation:** This block encompasses data loading, preprocessing, and other setup steps.
2. **Warmup:** Here, we perform preliminary runs to prepare the GPU and stabilize execution.
3. **Main Execution:** The core kernel execution phase, where primary computational operations occur.

This segmentation allows us to analyze each block independently, thereby pinpointing which phase contributes most to memory and computational overhead. NVTX's integration with NCU enables precise profiling of each block, ensuring that memory, compute efficiency, and latency metrics reflect the actual workload of each code segment.

11 Discussion

Our experiments demonstrate that:

- **Performance Gains**: The initial results presented are limited to small ranges of test graph sizes and sparsity.
- **Ease of Use vs. Performance**: PyTorch and CuPy provide ease of implementation with reasonable performance, suitable for rapid prototyping.
- **Impact of Graph Structure**: Graphs with different sparsity and connectivity affect the efficiency of sparse matrix multiplication, highlighting the need for adaptable methods.

We observed challenges in measuring GPU memory usage accurately due to asynchronous operations and the need for proper synchronization. Additionally, optimizing for one graph type may not generalize to others, emphasizing the importance of flexible optimization strategies.

12 Future Work

Our ultimate goal is to develop a self-improving system that *learns over time* how best to cluster graphs for sparse matrix multiplication in GNNs, drawing upon both *prior experimental data* and *evolutionary search* to refine partition strategies. Currently, each run conducts BFS- or edge-centric expansions and partitions the graph anew; however, for many GNN tasks—such as multi-epoch training or iterative inference—the same sparse adjacency matrices are multiplied repeatedly. By saving effective clusters from one multiplication, we can significantly reduce overhead for subsequent computations.

Cumulative Experimentation and Parameter Tracking. Since the performance of each partitioning strategy (e.g., BFS-based, edge-centric, spectral, or hybrid) depends on factors like node degree distribution and memory constraints, capturing these results over multiple runs allows the system to build a knowledge base of "what worked best for a given type of graph." We plan to store:

- *Final cluster assignments* that achieved high throughput or low memory overhead,
- *Evolved parameter settings* (e.g., batch size, BFS limits, number of partitions), derived from repeated runs on similar graphs,
- *Performance metrics* (throughput, memory usage, kernel execution times) for each tested strategy.

These records can then guide future multiplications on graphs with comparable topologies.

Adaptive Evolutionary Tuning. Although it was set aside to focus on GPU kernel improvements, one version of our system already leverages evolutionary algorithms (and, optionally, machine learning models) to propose new parameter sets for BFS expansions and cluster creation. Expanding this approach involves:

(1) *Seeding the evolutionary search* with high-performing cluster assignments from previous runs, rather than always starting from random or heuristically chosen partitions.
(2) *Refining the fitness function* to incorporate not just raw throughput, but also long-term stability (e.g., whether partition quality remains high across a range of feature dimensions or mini-batch sizes).
(3) *Integrating data from partial runs*: if only a subset of the graph changes (e.g., new edges appear or a few nodes are removed), the system can partially update existing clusters instead of redoing the entire BFS or edge decomposition from scratch.

Cluster Persistence and Reuse. For GNNs that reuse the same adjacency matrices over multiple epochs or inference passes, we envision a scheme where once the system *discovers* an efficient partition (through BFS, spectral, or hybrid expansions), that cluster assignment can be saved to disk. When the same graph is loaded in future sessions, the system quickly retrieves the previously best-known clusters—skipping the entire decomposition overhead and immediately running SpMM. Over time, repeated evolutions may discover an *even better* partition, which then replaces the old one for subsequent runs.

Broader Applications and Distributed Systems. Although our current focus is on single-GPU tests, these learning-driven strategies naturally extend to multi-GPU or distributed clusters, where consistent data locality is critical. Saving and refining partial partitions across devices will be essential for scaling out to very large graphs. By combining BFS expansions (or alternative edge-centric kernel strategies) with distributed scheduling, we hope to maintain high GPU occupancy while preventing communication bottlenecks.

Overall, our future efforts will center on a *long-lived, adaptive platform* that intelligently refines its clustering approaches for each new dataset, preserving the best results for subsequent multiplications. By harnessing repeated evolutionary searches, performance data logs, and cluster reusability, we aim to deliver a robust system that continually optimizes sparse matrix multiplication for GNN workloads—both in single- and multi-GPU environments.

13 Limitations

Despite the promising performance gains observed in certain scenarios, our methodology faces several constraints and open challenges:

(1) **Graph-Dependent Efficacy.** The BFS-based partitioning approach may underperform on graphs that exhibit highly skewed degrees or less "local" structure. In such cases, expansions may introduce unbalanced clusters that either exceed GPU memory capacity or fail to efficiently exploit parallelism.
(2) **Overlapping Clusters and Merging Overhead.** Our current pipeline allows nodes to appear in multiple clusters for coverage and concurrency. However, merging the partial outputs (e.g., averaging feature updates) increases overhead and introduces complexity. This overhead can offset the benefits of local parallelism, especially for large numbers of overlapping clusters.
(3) **Hardware-Specific Tuning.** We rely on GPU attribute detection (e.g., available memory, maximum threads per block, number of SMs) to guide cluster sizing and concurrency. Different GPUs (or multi-GPU systems) may require distinct tuning strategies, diminishing portability unless further auto-tuning or dynamic adjustment is implemented.
(4) **Incomplete Spectral Integration.** While we present a spectral + BFS approach, many advanced partitioning techniques remain unexplored. The current spectral initialization can be insufficient for complex real-world graphs with heterogeneous topologies.
(5) **Profiling and Memory Measurement Gaps.** Although we introduced NCU and NVTX for unified profiling, our memory tracking solution still faces synchronization challenges due to asynchronous kernel launches. Some GPU memory usage patterns may thus remain partially unobserved, leaving potential hidden bottlenecks unaddressed.

13.1 Improve Data Visualization

With the current profiling system now providing robust and comprehensive data, future work will focus on optimizing algorithms based on these metrics and enhancing visualization techniques for clearer metric interpretation. We anticipate that profiling-guided optimizations, particularly in memory handling and warp utilization, will yield measurable performance improvements.

13.2 GPU Optimization Techniques

Applying methods learned in class, we plan to:

- **Optimize Memory Hierarchy Usage**: Use shared memory and registers effectively to minimize latency.
- **Enhance Cache Utilization**: Structure data in localized regions such that cache utilization may increase.
- **Kernel Fusion and Tuning**: Combine operations and adjust kernel parameters for better performance guided by profilers.

13.3 Improve Memory Usage Measurements

Our current memory monitoring approach still has room for improvement. For phase 3 of this project, we will:

- **Improve Current Solutions:** Increase the reliability of our concurrent memory tracking solution to better reflect real-time memory usage during kernel execution.
- **Better CUDA Integration**: Leveraging official CLI tools offered by NVIDIA (e.g., nvidia-smi) to develop memory monitoring solutions.

13.4 Scalability Testing

We will extend our experiments to even larger graphs (depending upon the availability of powerful GPUs) to assess the scalability of our implementations and optimizations. This includes testing on graphs with millions of nodes and edges such as the Twitch network graphs from Stanford SNAP repository. We attempted to run our methods on larger graphs (Google dataset: 107614 nodes and 12238285 edges) but our GPUs ran out of memory.

14 Conclusion

We have presented a BFS-centric approach to sparse matrix multiplication in GNNs that harnesses hierarchical decompositions, Jaccard-based edge weighting, and pipeline parallelism to enhance throughput. By decomposing the graph into GPU-manageable clusters, we alleviate some of the irregularities inherent to large, sparse adjacency matrices. The pipeline further exploits concurrency by interleaving data loading and computation, guided by GPU resource awareness (thread blocks, memory availability, and concurrency features). Across synthetic and real-world datasets, our approach demonstrates the potential to match or outperform standard libraries under certain conditions, especially for high-sparsity or specialized topologies.

Nonetheless, our experiments highlight important trade-offs: overly aggressive BFS expansions can inflate cluster overlap, while insufficient expansions risk underutilizing GPU resources. Future work will focus on automated cluster-sizing heuristics, dynamic concurrency control, and exploring distributed GPU implementations. In addition, deeper integration of advanced partitioning algorithms (e.g., multi-cut methods, refined spectral embeddings) and extended profiling of memory behavior are needed to fully optimize the BFS-based pipeline. Ultimately, this research paves the way toward more efficient, adaptable frameworks for large-scale GNN computation in both single- and multi-GPU environments.

References

[1] [n. d.]. *CUDA C++ Programming Guide.* Retrieved December 20, 2024 from https://docs.nvidia.com/cuda/cuda-c-programming-guide/#wmma-example

[2] [n. d.]. *SNAP: Network datasets: Condense Matter collaboration network.* Retrieved November 10, 2024 from https://snap.stanford.edu/data/ca-CondMat.html

[3] [n. d.]. *SNAP: Network datasets: Social circles.* Retrieved November 10, 2024 from https://snap.stanford.edu/data/ego-Facebook.html

[4] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *2015 IEEE International Symposium on Workload Characterization.* IEEE, 56–65.

[5] 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.

[6] Timothy A. Davis and Yifan Hu. 2011. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS), Volume 38, Issue 1* 38, 1 (2011), 1–25.

[7] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[8] Da Li and Michela Becchi. 2013. Deploying graph algorithms on gpus: An adaptive solution. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing.* IEEE, 1013–1024.

[9] Zhonggen Li, Xiangyu Ke, Yifan Zhu, Yunjun Gao, and Yaofeng Tu. 2024. HC-SpMM: Accelerating Sparse Matrix-Matrix Multiplication for Graphs with Hybrid GPU Cores. *arXiv preprint arXiv:2412.08902* (2024).

[10] 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.

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

[12] 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.

[13] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing.* Springer, 672–687.

[14] Carl Yang, Aydın Buluç, and John D Owens. 2022. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)* 48, 1 (2022), 1–51.

A Project Repository

The source code for this project is available on GitHub at the following link:

- https://github.com/Kayisn/GPUGNN

This repository contains all code, documentation, and relevant resources used in the development of this project.