Enabling Accelerators for Graph Computing

A Dissertation Presented

by

Kaustubh Shivdikar

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

April 2023

To my family,
for their endless love, support, and encouragement.

Contents

List of Figures

List of Tables

List of Acronyms

Acknowledgments

I wish to extend my profound appreciation to my advisor, Dr. David Kaeli, for his pivotal role in my academic journey. His remarkable foresight and unwavering dedication have played a crucial role in shaping my path towards academic excellence. Dr. Kaeli's profound expertise and innovative spirit have served as a beacon of guidance through challenging periods, illuminating the path with his profound knowledge and strategic insight. His exceptional commitment to research and education has not only inspired me but also significantly influenced my professional development. The supportive and enriching environment he fostered has been instrumental in navigating the complexities of doctoral research, especially as an international student adapting to a new academic and cultural setting. Dr. Kaeli's mentorship extends beyond academic achievements, providing invaluable lessons in resilience and integrity. I am eternally grateful for the privilege of being mentored by a scientist of such high calibre and integrity, whose influence will undoubtedly echo throughout my career.

I extend my heartfelt gratitude to Prof. Ajay Joshi for his invaluable insights on Homomorphic Encryption, Prof. John Kim for his expertise on network topologies, Prof. José Luis Abellán for his deep knowledge on Graph Neural Networks, and Prof. Devesh Tiwari for his guidance on compute micro-architectures. Their contributions have been instrumental in shaping my research and enhancing my understanding of these complex areas. Additionally, my sincere thanks are due to all faculty members and administrative staff of the College of Engineering at Northeastern University for their unwavering support throughout my doctoral journey.

My sincere and warm thanks go also to my lab colleagues: Amir Taherin, Derek Rodriguez, Julian Gutierrez, Malith Jayaweera, Matin Raayai, Michael Shen, Neal Livesay, Nicolas Agostini, Sana Anvari, Trinayan Baruah, Yifan Sun, Alexander Ingare, and Zlatan Feric, for their invaluable contribution to my academic and personal growth. Their collaboration, insight, and camaraderie have been instrumental in fostering a nurturing and productive research environment. It has been a privilege to work alongside such dedicated and talented individuals. Their diverse perspectives and expertise have greatly enriched my research experience. Furthermore, their support and encouragement have been a constant source of motivation, making my journey through the intricacies of academic research both rewarding and memorable.

A Ph.D. journey that lasted 6 years, 7 months, and 25 days (or 2429 days in total) was made possible by the unwavering support from my friends Ahan Kak and Ashwin Shirsat. Sharing the joys of highs and support in the times of lows. Finally, last but by no means least, words will fail to express my gratitude to my family, who have been my rock through this journey, especially my parents Chandrakant and Anagha, and my brother Saumil Shivdikar. This achievement would have not been possible without their love, support, and encouragement.

Abstract of the Dissertation

Enabling Accelerators for Graph Computing

by

Kaustubh Shivdikar

Doctor of Philosophy in Electrical and Computer Engineering

Northeastern University, April 2023

Dr. David Kaeli, Advisor

The advent of Graph Neural Networks (GNNs) has revolutionized the field of machine learning, offering a novel paradigm for learning on graph-structured data. Unlike traditional neural networks, GNNs are capable of capturing complex relationships and dependencies inherent in graph data, making them particularly suited for a wide range of applications including social network analysis, molecular chemistry, and network security. The impact of GNNs in these domains is profound, enabling more accurate models and predictions, and thereby contributing significantly to advances in these fields.

GNNs, with their unique structure and operation, present new computational challenges compared to conventional neural networks. This requires comprehensive benchmarking and a thorough characterization of GNNs to obtain insight into their computational requirements and to identify potential performance bottlenecks. In this thesis, we aim to develop a better understanding of how GNNs interact with the underlying hardware and will leverage this knowledge as we design specialized accelerators and develop new optimizations, leading to more efficient and faster GNN computations.

A pivotal component within GNNs is the Sparse General Matrix-Matrix Multiplication (SpGEMM) kernel, known for its computational intensity and irregular memory access patterns. In this thesis, we address the challenges posed by SpGEMM by implementing a highly optimized hashing-based SpGEMM kernel tailored for a custom accelerator. This optimization is crucial to enhancing the performance of GNN workloads, ensuring that the acceleration potential of custom hardware is fully realized.

Synthesizing these insights and optimizations, we design state-of-the-art hardware accelerators capable of efficiently handling various GNN workloads. Our accelerator architectures are built on our characterization of GNN computational demands, providing clear motivation for our

approaches. Furthermore, we extend our exploration to emerging GNN workloads in the domain of graph neural networks. This exploration into novel models underlines our comprehensive approach, as we strive to enable accelerators that are not just performant, but also versatile, able to adapt to the evolving landscape of graph computing.

Chapter 1

Introduction

> In the vast landscape of computation,
> graphs stand as the bridges connecting
> isolated islands of data, creating a coherent
> world from chaos.
>
> ―――――――――――――――――――――――
> Inspired by Donald Knuth

Graph computing traces its lineage back to some of the earliest pursuits of mathematics. Historically, graph theory took its first steps in the 18th century with Leonhard Euler's formulation of the Seven Bridges of Königsberg problem, where he proved that it was impossible to traverse each of the city's seven bridges once and only once without retracing any step [57]. This abstract representation allowed for the modeling of a wide variety of systems, from social interactions to intricate molecular structures. However, for much of its history, graph theory remained primarily an academic endeavor with limited computational exploration, owing to the computational constraints of the era.

With the digital revolution of the late 20th century, computing power saw unprecedented growth. As industries started grappling with vast amounts of interconnected data—from the nascent Internet's web pages, to the massive social networks—there emerged, a pressing need for effective means to process and analyze this data. It was in this backdrop that graph computing began its ascent, evolving from theoretical speculations to practical, essential toolsets. The challenge shifted from simply understanding graph structures to efficiently processing them, leading to the exploration of specialized hardware and software solutions. This combination of data-centric challenges and available computational power provides the motivation for this thesis.

1.1 Background and Motivation

Graphs have seen a growing role in modern computational domains. With the rise of vast amounts of complex, interconnected data, traditional data processing methods have often fallen short. Enter Graph Neural Networks (GNNs), a specialized neural network architecture designed to handle such data and extract insights from these intricate connections. GNNs have shown remarkable potential in various domains, offering solutions where conventional methods have struggled. Their capacity to model relational data naturally fits a wide range of applications, spanning from social network analysis to molecular chemistry. To delve deeper into their capabilities, here is a list of GNN applications.

1. **Social Network Analysis**: Graphs are essential in Social Network Analysis (SNA) to understand interactions within networks such as Facebook and Twitter. By examining these structures, SNA can detect key influencers, community structures, and predict potential trends or misinformation spread [31, 181, 197].

2. **Computer Networks**: Graphs represent devices and communication pathways in computer networks, aiding in tasks such as routing and fault detection. Graph-based algorithms optimize network design and manage potential vulnerabilities [68, 81, 217].

3. **Hardware Security**: Circuits can be represented as graphs to detect anomalies, such as potential Hardware Trojans. Graph Neural Networks (GNNs) further enhance detection capabilities, pinpointing subtle irregularities [4, 76, 204].

4. **Bioinformatics**: In bioinformatics, graphs are used to model entities, for example, protein interactions and genetic sequences. They help identify conserved patterns in DNA or RNA, study metabolic pathways, and analyze genomic variations [18, 116].

5. **Financial Markets**: Graphs illuminate interactions in financial markets. They help analysts identify correlated assets, assess systemic risks, and optimize algorithmic trading strategies [20, 55, 77].

6. **Neuroscience**: Representing the brain's complex structure, graphs in neuroscience help analyze neural networks, understand (Functional magnetic resonance imaging) fMRI data, and study the brain's topological properties [15, 24, 59, 167].
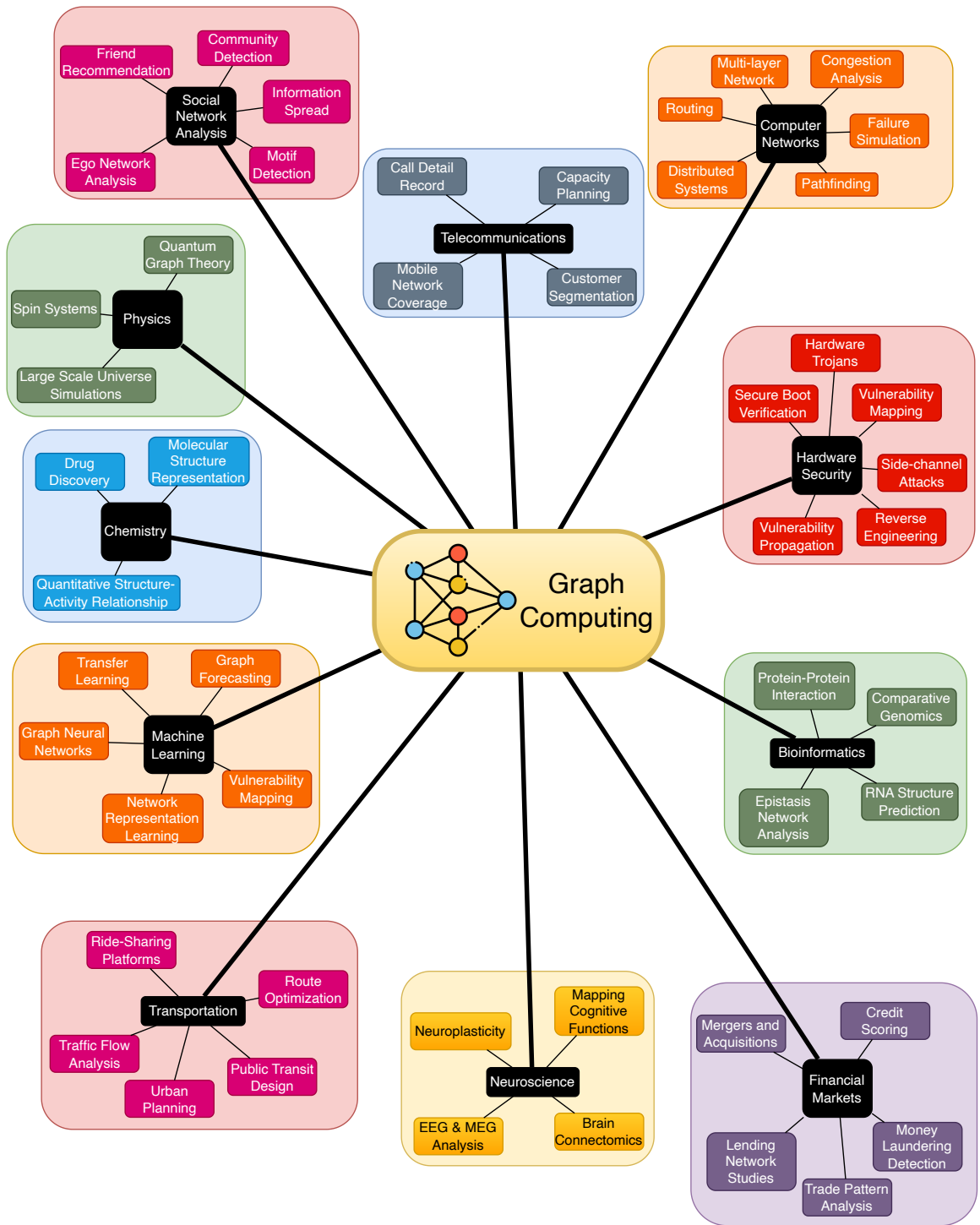
Figure 1.1: Graph Computing Applications

7. **Transportation and Logistics**: Graphs map transportation networks, aiding in solving optimization problems such as route planning and vehicle routing. They help ensure efficient deliveries, optimize public transit, and adapt to disruptions [47, 69, 108, 201].

8. **Chemistry**: Molecular structures in chemistry are represented using graphs, predicting chemical properties, identifying isomers, and aiding in drug design [12, 178, 180, 190].

9. **Physics**: Graphs in physics are used to model quantum states, lattice structures, and network systems. They aid in understanding quantum computing, material properties, and system dynamics [13, 29, 79, 192].

10. **Machine Learning and Data Mining**: In machine learning, graphs facilitate clustering, classification, and feature extraction. They model non-Euclidean data, aiding in semi-supervised learning and geometric deep learning [23, 34, 191, 218].

11. **Telecommunications**: Graphs underpin the telecommunications sector, assisting in routing, optimization, and network slicing for 5G technologies [3, 64, 140, 187].

This list showcases the wide impact of concepts from graph theory across a multitude of domains. Figure 1.1 illustrates specific applications of graphs across various domains. As interconnected systems become increasingly central to understanding our world, the role of graph theory is likely to grow even more essential.

In the wake of the rising importance of graph-based computations, the hardware landscape within the compute industry began to undergo key shifts. Traditional Central Processing Units (CPUs), initially designed for sequential tasks, started incorporating SIMD-based graph extensions to enhance parallel processing capabilities [216].Graphics Processing Units (GPUs), with their inherent parallelism, were enhanced with kernel support tailored specifically for graph algorithms [148, 175]. Beyond these general-purpose processors, the industry also witnessed the advent of domain-specific accelerators [86, 115, 153, 203], specifically crafted to speedup graph computations, addressing the unique challenges and demands that graph algorithms present.

On the software front, comprehensive software stacks have been designed from the ground up, specifically focusing on facilitating efficient graph processing. Libraries such as PyTorch-Geometric [58] and Deep Graph Library (DGL) [186] emerged, offering robust platforms for researchers and developers to implement and optimize graph algorithms. These software advances, in tandem with hardware innovations, have pushed the field of graph computing further forward.

4

However, as with any rapidly evolving domain, while significant strides have been made, the journey is far from over. The vast potential of graph computing continues to present both challenges and opportunities that require further exploration and innovation.

1.2 Challenges in Accelerating Graph Computing

As promising as GNNs are, they are not without their computational challenges. Given the inherently recursive nature of GNNs, coupled with the irregular structure of many real-world graphs, we find significant bottlenecks in terms of their scalability and performance. Parallelizing GNN computations, which seems to be a logical solution given the abundance of task-level parallelism in graph processing, is riddled with data dependencies. The core challenges lie in the low spatial locality of data, which results in memory access inefficiencies and the computation stalls, making it difficult to achieve scalable parallel performance. Addressing these challenges is pivotal to harnessing the full potential of GNNs and making them a feasible solution for large-scale, real-world applications.

1.2.1 Scalability Concerns in Graph Neural Networks

GNNs present unique challenges in handling large graphs, which often involve billions to trillions of nodes and edges [60]. The computational and memory demands of processing such exascale graphs escalate rapidly, often making computations infeasible on standard computational infrastructures. For instance, the Friendster social network graph comprises over 65 million nodes and 1.8 billion edges [200]. When applying GNNs to this magnitude of data, the iterative and recursive nature of these networks requires massive memory bandwidth and computational power. Even more complex graphs from biological and cosmological simulations are looming on the horizon, potentially reaching the exabyte scale [9]. Addressing the scalability challenges with GNNs on these graphs will be paramount to unlocking new scientific discoveries and advances in several fields.

1.2.2 Task-Level Parallelism in Graph Computations

Graph computations inherently possess a rich vein of task-level parallelism, given that many operations can theoretically be conducted simultaneously across different nodes or subgraphs [125]. This natural parallelism emerges from the decentralized structure of graphs, where independent subtasks can be identified and processed in parallel, especially in sparse graph computations. However,

this potential for parallelism is intermingled with intricate data dependencies among nodes and edges. These dependencies arise from the interconnections in the graph, leading to situations where the output of one task is contingent upon the result of another, thus necessitating synchronization and communication [17]. Consequently, fully capitalizing on the inherent task-level parallelism, while managing data dependencies, poses significant challenges, requiring advanced strategies to ensure efficiency and accuracy in graph computations.

1.2.3   Data Spatial Locality and Computational Irregularity

Graph computations, especially in the realm of GNNs and large-scale graph analytics, encounter two predominant challenges: 1) data spatial locality and 2) computational irregularity [38]. First, the lack of data spatial locality implies that successive operations might access data dispersed across memory, leading to increased cache misses and degraded memory performance. Graphs, being inherently non-uniform, result in unpredictable memory access patterns, often unable to take advantage of cache hierarchies in modern processors [104]. Second, computational irregularity in graph algorithms surfaces due to the diverse node degrees and edge distributions, causing workload imbalance in parallel computing scenarios. This irregularity complicates efficient scheduling on parallel hardware and requires sophisticated load-balancing techniques to mitigate performance imbalance [97]. Both challenges pose barriers to fully realize the benefits of parallelism in graph computing.

1.3   Objectives and Contributions

The development of a graph accelerator tailored for GNN workloads requires following a systematic approach. Initially, our efforts are concentrated on conducting comprehensive benchmarks and characterizations of the GNN workloads, along with the various kernels targeted for optimization. Subsequently, our attention shifts to addressing a significant bottleneck in GNN workloads—the SpGEMM kernel—by devising and implementing an optimized version specifically for a custom accelerator. Finally, we integrate these insights and optimizations to architect a hardware accelerator, specifically designed to enhance the performance across a diverse range of GNN workloads. A summary of these objectives and their associated contributions is presented in Table 1.1.

Objective	Contribution
Analyze the architectural impact of computing Graph Neural Networks.	Provide a benchmark suite for comprehensive evaluation of how different architectural components influence the performance and efficiency of GNN computations.
Accelerate SpGEMM kernel on custom accelerator.	Propose and implement optimization strategies for the SpGEMM kernel, resulting in significant performance improvements on a custom hardware accelerator.
Design a new accelerator to accelerate multiple GNN workloads.	Develop a novel hardware accelerator architecture tailored for various GNN workloads, demonstrating versatility and improved performance across sparse and dense compute kernels.

Table 1.1: Summary of Objectives and Contributions

1.4 Dissertation Organization

- Chapter 1: This chapter sets the stage for the thesis by highlighting the significance and impact of computations based on graph structures.

- Chapter 2: This chapter provides background on the foundations of graph theory, providing a comprehensive overview of the various types of graphs, their properties and their representations. It also introduces machine learning on graphs, with a specific focus on Graph Neural Networks (GNNs), explaining their structure, functionality and applications.

- Chapter 3: This chapter presents a thorough review of previous work in the domain, discussing existing approaches and solutions in workload characterization, GPU acceleration, Coarse-Grained Reconfigurable Arrays (CGRAs), and custom accelerators specifically designed for GNN workloads.

- Chapter 4: We present a GNN benchmark suite in this chapter, offering a curated collection of workloads and tools to assess the performance of GNN computations. In addition, we conduct extensive analysis of the architectural requirements to efficiently run GNN workloads, examining how different components and configurations affect overall performance.

- Chapter 5: We explore the SMASH (Sparse Matrix Atomic Scratchpad Hashing) algorithm, a novel SpGEMM kernel optimization aimed at enhancing GNN processing. We discuss the development and implementation of SMASH, including its various versions tailored to exploit distinct architectural features for improved efficiency in GNN workloads.

- Chapter 6: In this chapter, we introduce NeuraChip, a custom CGRA-based accelerator designed to meet the unique demands of GNN computations. We provide detailed discussion on the architecture, including its heterogeneous processing approach, adaptive hash-based compute mapping, and mechanisms for rolling evictions, highlighting how NeuraChip addresses critical bottlenecks in GNN acceleration.

- Chapter 7: The concluding chapter discusses the key insights and contributions of this work, ranging from the development of a GNN benchmark suite, to the introduction of algorithmic and hardware innovations such as SMASH and NeuraChip. We reflect on the impact of these contributions on the field of GNN acceleration. We also cover potential directions for future GNN research to explore further advancements in graph computing.

Chapter 2

Fundamentals of Graph Computing and Accelerator Architectures

Before delving into the details of our accelerator proposal for graph computing, we explore the foundational concepts of graph computing and accelerator design. This chapter serves as a foundation, offering a systematic overview of the essential background, terminology and challenges that characterize graph computing, as well as its impact on hardware accelerators. By first reviewing these foundational elements, this chapter aims to provide the necessary background required to understand GNN accelerator design.

2.1 Graph Theory Basics

Graph theory is a field within mathematics that explores the structure of interconnected nodes and edges. Graphs have gained widespread recognition for their ability to represent non-euclidean data. Graphs can represent many real-world problems, from network topologies and social networks to transportation systems and molecular structures. This section provides an overview of the fundamental elements of graphs, before diving into the use of graph-based applications in machine learning. Figure 2.1 represents a social network graph, demonstrating how individuals are connected. We also provide the associated adjacency matrix, which quantitatively captures these connections. Additionally, for each individual in the network, a feature vector is provided that captures various attributes or characteristics associated with that individual.

Mathematically, a graph G is a pair of vertices (i.e., nodes) V and edges E, that is represented as:

$$G = (V, E) \tag{2.1}$$

where V is a set of vertices, and E is an unordered set of pairs of vertices. A singular edge e within the set of edges E is represented as $e = \{x, y\}$ or simply $e = xy$, where x and y are endpoints (nodes) of the edge. x and y are said to be neighbors or adjacent nodes in the graph G.
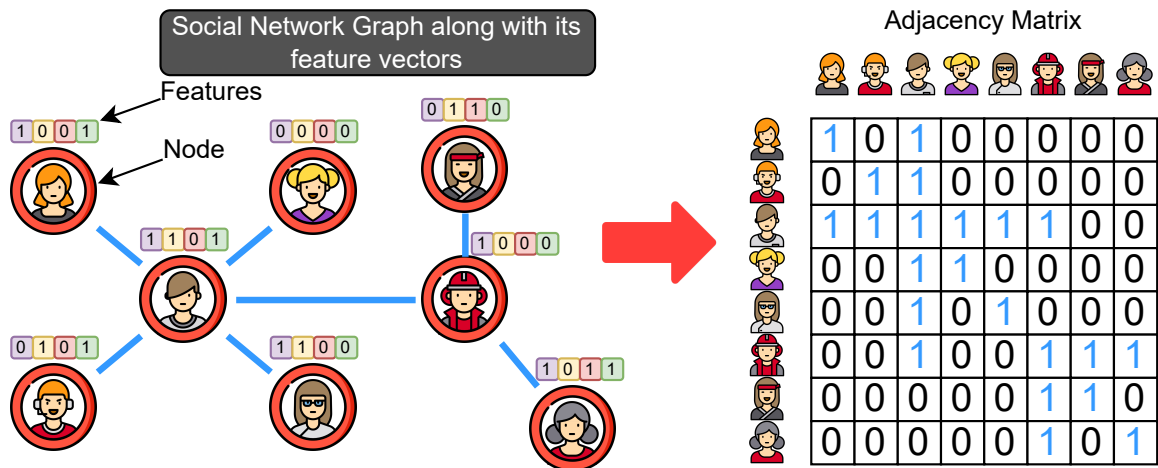
Figure 2.1: An example of a social network graph and its corresponding adjacency matrix. Each node in the graph is associated with a feature vector that contains the node's attributes.

2.1.1 Graph Properties

Graph properties provide insight into the structure, characteristics and behavior of graphs. Here are some of the fundamental properties and characteristics of graphs:

1. **Degree**: The number of edges incident on a vertex. In directed graphs, we differentiate between in-degree (number of incoming edges) and out-degree (number of outgoing edges).

2. **Order and Size**: The order of a graph refers to the number of vertices and the size refers to the number of its edges.

3. **Diameter**: The longest shortest path between any two vertices.

4. **Radius**: The minimum eccentricity of any vertex in the graph. The eccentricity of a vertex is the greatest distance from the vertex to any other vertex.

5. **Girth**: The length of the shortest cycle in the graph.

6. **Adjacency**: Two vertices are said to be adjacent if they are connected by an edge.

7. **Clique**: A set of vertices where each pair is adjacent.

8. **Path**: A sequence of vertices where each adjacent pair is connected by an edge.

9. **Cycle**: A path that starts and ends at the same vertex. A cyclic graph is a graph that contains at least one cycle, which is a sequence of vertices where the first and last vertices are the same, and each pair of consecutive vertices in the sequence is connected by an edge. An acyclic graph is a graph that contains no cycles.

10. **Connectivity**: A graph is connected if there's a path between every pair of vertices. In directed graphs, if a graph is strongly connected, there is a directed path between any pair of vertices.

11. **Connected Components**:

    For undirected graphs: A connected component is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph (i.e., the main graph). In simpler terms, a connected component is a "piece" or "part" of the graph where there's a route between any pair of nodes within that piece, but no connection to nodes outside of it.

    For directed graphs: Connected components are further classified as Strongly Connected Components and Weakly Connected Components.

    Strongly Connected Components: A strongly connected component of a directed graph is a maximal strongly connected subgraph. This means that for every pair of vertices u and v in the subgraph, there's a directed path from u to v and a directed path from v to u.

    Weakly Connected Components: If you were to ignore the directionality of the edges in a directed graph, and the graph becomes connected, then the graph is said to be weakly connected. The maximal subgraphs of this type are the weakly connected components.

12. **Planarity**: A graph is said to be planar if it can be embedded (i.e., drawn) in the plane such that no edges intersect or cross each other except at their endpoints (vertices). In other words, a graph is planar if it can be drawn on a flat surface without any of its edges overlapping

or crossing, except where they meet at nodes. This definition applies to both directed and undirected graphs.

13. **Isomorphism**: Isomorphism refers to a one-to-one correspondence between the vertices of two graphs such that the adjacency relation between pairs of vertices is preserved. In simpler terms, two graphs are isomorphic if they are essentially the same in terms of structure, though they might look different in their graphical representation.

    Formally, two graphs G_1 and G_2 are said to be isomorphic if there exists a bijective function $f : V(G_1) \rightarrow V(G_2)$ such that for any two vertices u and v of G_1, there is an edge between u and v in G_1 if and only if there is an edge between $f(u)$ and $f(v)$ in G_2.

2.1.2 Types of Graphs

Graphs can be categorized in various ways based on their properties, structures, and applications. Below is an overview of the primary types of graphs:

1. **Undirected Graph**: An undirected graph is a simple structure consisting of nodes, also known as vertices, connected by edges. In this type of graph, the edges don't have a specific direction. That is, if vertex A is connected to vertex B, then vertex B is equivalently connected to vertex A. Such graphs are commonly used to represent symmetric relationships, for example, friendships in a social network.

2. **Directed Graph (or Digraph)**: Directed graphs, often called digraphs, also comprise vertices and edges. However, the crucial difference is that the edges have a direction. An arrow from vertex A to vertex B signifies a one-way relationship. Digraphs are especially useful in representing prerequisites in a course structure or transitions in a finite automaton.

3. **Weighted Graph**: A weighted graph assigns a specific weight or value to each of its edges. This weight can represent various characteristics such as distance, cost, or any measurable quantity relevant to the problem being addressed. For instance, in mapping out a city's road network, the weights could symbolize the distances or travel times between intersections.

4. **Unweighted Graph**: Contrary to weighted graphs, unweighted graphs treat each edge equally, without any specific value or weight assigned. Such graphs are often utilized in scenarios where only the relationship or connection between nodes is of interest, without any quantitative measure on the edges.

5. **Cyclic Graph**: A cyclic graph contains at least one cycle, which is a closed path in which a vertex is revisited without retracing any edge. Cyclic graphs can represent systems or networks where it's possible to return to a starting point via a unique route.

6. **Acyclic Graph**: An acyclic graph is devoid of any cycles. This means it's impossible to start at one vertex and traverse the graph in such a way that you return to the starting vertex without backtracking. A classic example of an acyclic graph is a tree.

7. **Connected Graph**: In a connected graph, there exists a path between every pair of vertices, ensuring that no vertex is isolated. Such graphs are particularly valuable in scenarios where continuous connectivity is essential (communication networks are one such example).

8. **Disconnected Graph**: As the name suggests, a disconnected graph has one or more vertices that aren't connected to the rest of the graph. In other words, not all pairs of vertices are reachable from each other. Such graphs might represent fragmented or isolated systems.

9. **Complete Graph**: A complete graph is a robust structure where every pair of distinct vertices is connected by a unique edge. In terms of social networks, a complete graph would mean every person knows every other person directly.

10. **Bipartite Graph**: A graph G is called bipartite if its vertex set can be partitioned into two disjoint sets U and V such that every edge connects a vertex in U to one in V. This means that there are no edges that connect vertices within the same set U or within the same set V.

    Formally, a graph $G = (V, E)$ is bipartite if there exists a partition (U, V) of its vertex set V such that for every edge $(x, y) \in E$, either $x \in U$ and $y \in V$ or $x \in V$ and $y \in U$.

11. **Planar Graph**: A planar graph can be drawn on a plane without any edges crossing, except at their endpoints. Such graphs are beneficial in specific design and layout problems, ensuring no overlaps or intersections.

12. **Tree**: A tree is a special kind of graph that's both connected and acyclic. Trees are hierarchical structures commonly used in computer science for data structures such as binary search trees and file systems.

13. **Forest**: A forest is a collection of disjoint trees, meaning it's acyclic but not necessarily connected. Forests can represent multiple independent hierarchies or classifications within a system.

14. **Multigraph**: Multigraphs allow for multiple edges, also termed parallel edges, between the same set of vertices. This type of graph can be useful in scenarios where multiple distinct relationships or connections exist between the same entities.

15. **Simple Graph**: A simple graph is a basic structure where each pair of vertices shares at most one edge, and there are no loops. It's the foundational form of many other graph types and serves as a starting point in many graph theory discussions.

16. **Hypergraph**: A hypergraph generalizes the traditional graph concept by allowing edges, often called hyperedges, to connect any number of vertices, not just two. Hypergraphs can represent complex relationships that don't fit neatly into pairwise associations.

17. **Subgraph**: A subgraph is formed by selecting a subset of vertices and edges from a larger graph, without introducing any new ones. Subgraphs are crucial for analyzing specific portions or aspects of a larger network or system.

18. **Regular Graph**: In a regular graph, every vertex has the same degree, meaning each node connects to an equal number of other nodes. This uniformity can simplify certain analyses and algorithms. Specifically in the context of graph computations, Regular graphs lead to close to uniform work distribution across the computing cores.

2.1.3 Graph Representation

Graphs, as abstract mathematical structures, need to be represented in a tangible form, especially for computational processes. The choice of representation can significantly influence the efficiency of various graph algorithms. The most common forms of graph representation include:

1. **Adjacency Matrix**: This is a 2D array of size $V \times V$ (where V is the number of vertices in a graph). The entry m_{ij} is either 1 (or the edge's weight) if there's an edge between vertices i and j, and 0 otherwise. While this method provides a quick way to check the presence of a specific edge, it can be space-inefficient for sparse graphs as it requires $O(V^2)$ space. Figure 2.1 illustrates a social network graph alongside its representation in adjacency matrix format.

2. **Adjacency List**: For every vertex, a list of its adjacent vertices is maintained. This representation is more space-efficient for sparse graphs. In this method, an array of lists is used, with

the size of the array being equal to the number of vertices. The ith position in the array holds a list of nodes to which node i is connected.

3. **Incidence Matrix**: This is a 2D array where rows represent vertices and columns represent edges. For example, for an undirected graph, the entry m_{ij} is 1 if vertex i is incident to edge j, -1 if i is the edge's terminal vertex, and 0 otherwise. For a directed graph, the entry is -1 for the tail of the arrow (edge) and 1 for the head.

4. **Edge List**: This is a list of pairs (or triples, if weights are present) that represent edges. For instance, an edge from vertex A to vertex B with weight w can be represented as (A, B, w). This representation is particularly useful when the graph structure is more concerned with edges rather than vertices.

The choice of representation often hinges on the specific operations that need to be optimized. For instance, adjacency lists are faster for traversal algorithms, while adjacency matrices can be more suitable for algorithms involving edge lookups or matrix operations.

2.1.4 Graph Transformation

Graph transformation is a powerful technique that focuses on the modification and manipulation of graph structures. It offers a systematic way to derive a new graph from an existing one, serving both as a computational tool and a conceptual methodology to analyze various properties and behaviors of graphs. Common types of graph transformations include:

1. **Subgraph Extraction**: This involves creating a new graph by selecting a subset of vertices and edges from the original graph, usually based on certain criteria or conditions.

2. **Graph Contraction**: This process combines multiple vertices into a single vertex, often to simplify a graph's structure while retaining its fundamental characteristics.

3. **Graph Expansion (or Vertex Splitting)**: This is the reverse of contraction. A single vertex is expanded into multiple vertices, with edges adjusted accordingly.

4. **Edge Contraction**: Two vertices connected by an edge are merged into a single vertex, and the edge is removed. The new vertex retains all edges that the original vertices had, except for the contracted edge.

5. **Line Graph Transformation**: Given a graph, its line graph is another graph representing the relationship between the edges of the original graph. Each vertex in the line graph represents an edge in the original graph, and two vertices in the line graph are connected if their corresponding edges in the original graph are incident on a common vertex.

6. **Dual Graph Transformation**: Applied typically to planar graphs, this creates a vertex in the dual graph for every face in the original graph, and two vertices in the dual graph are connected by an edge if their corresponding faces in the original graph are separated by an edge.

   Graph transformations play a crucial role in a myriad of applications, including algorithm design, network analysis, and optimization problems, by enabling alternative perspectives and simplifications of the original structures.

2.2 Machine Learning on Graphs

Building upon the principles of graph theory just reviewed, we now transition to the applications. Graphs are useful abstractions as then naturally map to a number of machine learning tasks. Graphs are used for non-Euclidean data structures that encapsulate relationships, hierarchies, and patterns, which are difficult to model in traditional data formats. In this section, we delve into the techniques, algorithms, and challenges associated with leveraging graph data for predictive and analytical tasks. This emerging field promises important advancements in domains ranging from social network analysis and recommendation systems to bioinformatics and traffic routing.

2.2.1 Graph Neural Networks

Graph Neural Networks (GNNs) have emerged as powerful tools for learning and processing data structured as graphs. Unlike traditional neural networks that operate on fixed-size vectors, GNNs work directly with graphs, accommodating their non-euclidean nature and inherent irregularities. At the core of GNNs lies the principle of aggregating information from a node's neighbors to iteratively update the node's representation. This process captures both local structures and broader topological features of the graph. Through successive layers, GNNs can accumulate and transform information from increasingly larger neighborhoods around each node. This ability to learn meaningful representations of nodes, edges, or entire graphs has led to their successful application in diverse areas such as social network analysis, molecular chemistry, and recommendation systems,

bridging the gap between the rich expressivity of graphs and the computational capabilities of deep learning.
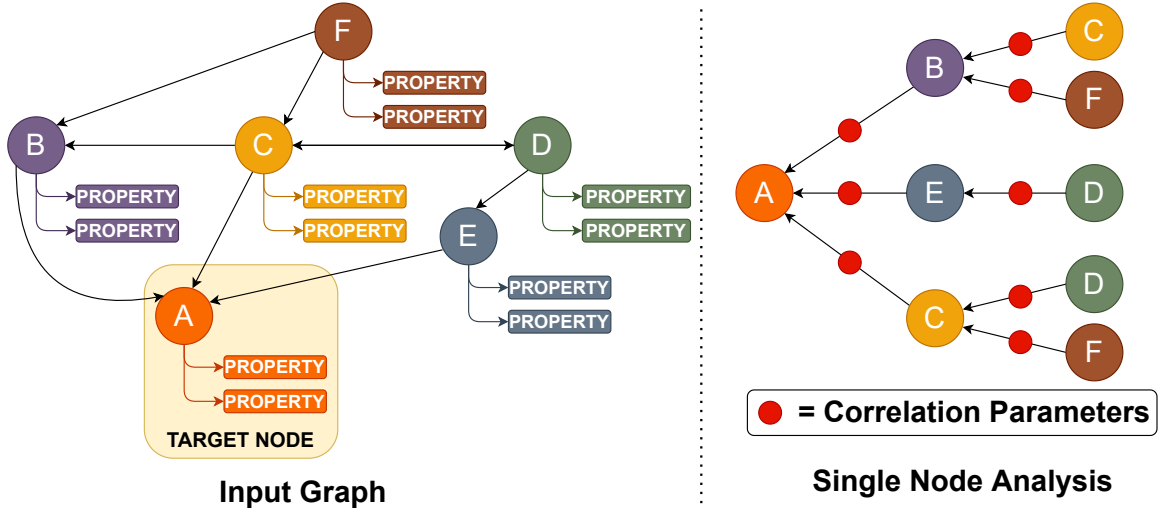
Figure 2.2: Analysis of Graph Neural Networks, demonstrating the propagation of node properties influenced by the graph's topology.

Before a GNN model can make predictions, the model must first be trained. As shown in Figure 2.2, the goal of GNN training is to learn correlation parameters for each node, capturing its relation to the rest of the graph. More specifically, a feature vector for node A relates node A's properties to its neighbors' properties, nodes B, C and E. Each of these neighbors, in turn, has their own feature vectors to relate to their own neighbors. Hence, the properties of node A can be associated with the properties of every other node in the graph. This feature of GNNs is also useful for finding missing properties of nodes in a graph. Similar to DNNs, a GNN can also have multiple layers, with each layer represented by two functions: i) an aggregation function and ii) an update function (i.e., a combination function). As the name suggests, the aggregation function is responsible for collecting or pooling the features of the neighbors for a given node. On the other hand, the update function is responsible for updating each node's feature vectors using Multi-Layer Perceptrons (MLPs). A GNN model can have layers with different aggregation and update functions. The deeper the GNN model, the more information a node has about other nodes that are distant from it in the graph. However, training deeper GNNs is difficult, primarily due to the vanishing gradient problem [80]. As GNNs grow deeper, the gradients become so small that the weights stop getting updated. This property makes it difficult to train the GNN further. To address these challenges, novel GNN architectures have been proposed to enable deeper GNN models [111].

2.2.2 Graph Convolutional Networks

Graph Convolutional Networks (GCNs) represent a key advancement in the domain of graph-based machine learning. These networks are designed to process data structured as graphs, allowing for the consideration of both node features and the graph's inherent structure. Traditional neural networks are ill-suited for graph data due to the irregular and non-Euclidean nature of graphs. In contrast, GCNs leverage the spatial relationship between nodes to propagate and aggregate information through the graph, thus learning a more comprehensive representation of the data.

The propagation rule for a GCN layer can be described as:

$$H_{(l+1)} = \sigma \left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H_{(l)} W_{(l)} \right) \tag{2.2}$$

Where:

$H^{(l)}$ is the matrix of node features at layer l.

$\tilde{A}$ is the adjacency matrix of the graph with added self-connections

$\tilde{D}$ is the diagonal node degree matrix of $\tilde{A}$.

$W^{(l)}$ is the weight matrix for layer l.

σ is an activation function, e.g., the ReLU function.

The key principle of a GCN is the neighborhood aggregation scheme. For each node in the graph, a GCN layer aggregates feature information from its neighbors and possibly itself. This aggregated information is then passed through a transformation (usually a linear transformation, followed by a non-linear activation function). The process can be iteratively run over multiple layers, enabling the aggregation of information from a larger neighborhood at each subsequent layer. The inclusion of this spatial-based information aggregation makes GCNs particularly adept at node classification, graph classification, and link prediction, especially when the structure of the graph plays a significant role in the underlying data distribution.

2.2.3 Graph Isomorphism Networks

Graph isomorphism is a central concept in the field of graph theory, revolving around the study of the structural equivalence between two graphs. Two graphs G_1 and G_2 are considered isomorphic if there exists a one-to-one correspondence (or bijective function) between their vertices, such that the adjacency relationship is preserved. In other words, the graphs are structurally identical, and one can be transformed into the other merely by relabeling the vertices without altering the

underlying connectivity pattern. While the concept sounds straightforward, determining whether two large graphs are isomorphic in an efficient manner remains a challenging computational problem.

The update rule for the GIN can be formulated as:

$$h_v^{(l+1)} = \text{MLP}^{(l)} \left(h_v^{(l)} + \sum_{u \in \mathcal{N}(v)} h_u^{(l)} \right) \tag{2.3}$$

Where:

$h_v^{(l+1)}$ is the feature vector of node v at layer $l + 1$.

$\mathcal{N}(v)$ represents the neighbors of node v.

$\text{MLP}^{(l)}$ denotes a multi-layer perceptron used at layer l.

The GIN introduces an additional learnable parameter to weigh the importance of self-features versus neighbor features. This ensures that the GIN can capture subtle structural details, making it a powerful tool for graph representation learning.

Understanding and recognizing graph isomorphism has profound implications in numerous areas of science and technology. For example, in chemistry, graph isomorphism can be used to determine molecular similarity, as molecules can be represented as graphs where atoms are vertices and bonds are edges. Similarly, in computer science, isomorphic graphs might denote equivalent solutions or states in certain problems. Furthermore, in database search and pattern recognition, determining graph isomorphism efficiently can aid in retrieving or recognizing specific patterns amidst a large dataset. However, due to the complexity of the problem, especially with large graphs, much research has been invested in finding efficient algorithms and heuristic methods to tackle the isomorphism challenge.

2.2.4   Graph Attention Networks

Graph Attention Networks (GATs) mark a significant evolution in graph neural network technology by introducing an attention mechanism that allows nodes to dynamically assign importance to their neighbors' information. This attention-based approach enables the model to focus more on relevant features from a neighborhood, enhancing the adaptability and performance of the network on graph-structured data. GATs address the limitation of conventional graph neural networks, such as GCNs, which treat all neighbors equally during the aggregation process. By in-

corporating attention, GATs can assign a weight based on the influence of each neighbor based on the task at hand, leading to more effective learning outcomes.

The propagation rule for a GAT layer can be described as:

$$h_i^{(l+1)} = \sigma \left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{ij}^{(l)} W^{(l)} h_j^{(l)} \right) \tag{2.4}$$

Where:

$h_i^{(l+1)}$ is the feature vector of node i at layer $l + 1$.

$\alpha_{ij}^{(l)}$ represents the attention coefficient between nodes i and j at layer l, indicating the significance of node j's features to the update of node i's features.

$W^{(l)}$ is the weight matrix for layer l.

σ is an activation function, such as the ReLU function.

The computation of attention coefficients involves a self-attention mechanism where a shared attention function, applicable to all edges, computes the coefficients based on the features of the nodes at either end of the edge. This process allows GATs to perform feature extraction that is both context-aware and adaptive, leading to more nuanced representations of nodes based on their local graph topology.

GATs have demonstrated superior performance on various tasks, including node classification, graph classification, and link prediction, particularly in scenarios where the relevance of neighboring nodes varies significantly. The model's ability to selectively prioritize information makes it highly effective in capturing the complex dependencies characteristic of graph-structured data, thereby pushing the boundaries of what is achievable with graph neural networks.

2.2.5 GraphSAGE

GraphSAGE (Graph Sample and AggregatE) is a novel framework designed to efficiently generate node embeddings for large-scale graphs. Unlike traditional graph neural networks, such as GCNs, that require the entire graph to be processed simultaneously, GraphSAGE introduces a more scalable approach by sampling a fixed-size neighborhood around each node and aggregating their features. This methodology allows GraphSAGE to efficiently deal with graphs of varying sizes and topologies, including those that evolve over time, by learning a function that can generate embeddings for unseen nodes based on their local neighborhoods.

The propagation rule for GraphSAGE can be generalized as:

$$h_i^{(l+1)} = \sigma \left(W^{(l)} \cdot \text{AGGREGATE}^{(l)} \left(\{h_j^{(l)} | j \in \mathcal{N}(i)\} \right) \right) \tag{2.5}$$

Where:

$h_i^{(l+1)}$ is the feature vector of node i at layer $l + 1$.

$\mathcal{N}(i)$ denotes the set of neighbors for node i.

$\text{AGGREGATE}^{(l)}$ is a function that combines the feature vectors of the sampled neighborhood nodes at layer l.

$W^{(l)}$ is the weight matrix for layer l.

σ represents an activation function, such as the ReLU function.

GraphSAGE's aggregation functions can vary, including mean, LSTM, and pooling aggregators, which allows the model to be tailored to specific types of graph data and applications. This flexibility, combined with the efficiency of sampling, makes GraphSAGE particularly suitable for dynamic graphs and scenarios where real-time embedding generation is crucial.

Moreover, by learning to aggregate information from a node's local neighborhood, GraphSAGE can leverage the structural information inherent in the graph, allowing for powerful representations that capture both the features of individual nodes and their relational context within the graph. This approach has proven effective across a range of tasks, including node classification, link prediction, and graph classification, particularly in domains where the graph structure is indicative of underlying patterns or relationships, such as social networks, recommendation systems, and knowledge graphs.

2.2.6 Principal Neighborhood Aggregation

Principal Neighborhood Aggregation (PNA) addresses the need for more nuanced aggregation mechanisms capable of capturing the diverse structural properties within graphs. Diverging from traditional graph neural networks such as GCNs, which primarily utilize simplistic aggregation functions such as sum, mean, or max, PNA introduces a multifaceted approach by integrating several aggregation schemes and a degree-scaling component. This mixture of techniques enhances the model's ability to represent the intricate patterns and relationships inherent in graph-structured data.

The core idea of PNA is to leverage the strengths of multiple aggregation functions simultaneously, thereby improving the feature representation of each node by capturing various aspects

of its neighborhood's structure and feature distribution. The inclusion of a degree-scaling component further refines this process by adjusting the influence of neighboring nodes based on their connectivity, thus providing a more balanced and informative aggregation outcome.

The general update rule for PNA can be encapsulated as:

$$h_i^{(l+1)} = \sigma \left(\sum_{\text{agg} \in \mathcal{A}} \delta_{\text{agg}} \cdot W_{\text{agg}}^{(l)} \cdot \text{agg} \left(\{h_j^{(l)} | j \in \mathcal{N}(i) \cup \{i\}\} \right) \right) \tag{2.6}$$

Where:

- $h_i^{(l+1)}$ denotes the feature vector of node i at layer $l + 1$.

- $\mathcal{A}$ is a set of aggregation functions, such as sum, mean, and max.

- δ_{agg} represents a degree-scaling factor associated with each aggregation function, optimizing the impact of node degrees on the aggregation process.

- $W_{\text{agg}}^{(l)}$ is a weight matrix specific to each aggregation function at layer l.

- σ is an activation function, for instance, the ReLU function.

PNA's design is particularly effective in graphs where nodes exhibit significant variability in their degree distribution. By considering multiple aggregation perspectives and adjusting for node degree, PNA ensures an accurate representation of each node's neighborhood, significantly improving the performance on tasks such as node classification, graph classification, and link prediction.

2.3 Graph Neural Network Frameworks

Support for GNN primitives in popular ML frameworks is increasing. Today, researchers from the ML community are developing libraries in the form of extensions to frameworks such as PyTorch and TensorFlow. The two most popular libraries/extensions that implement customized GNN kernels, as well as provide programming support in the form of APIs, are PyTorch Geometric (PyG) [58] and the Deep Graph Library (DGL) [186]. PyG is an extension based on top of the PyTorch library and so only supports PyTorch. On the other hand, DGL provides support for PyTorch, TensorFlow, and MXNet.

Spektral [72] and Aligraph [221] are two librariesk built on top of TensorFlow, used for GNN training. GraphNets [16] is a GNN framework from Google, supported using TensorFlow as the backend. As PyG and DGL bridge both the semantic and performance gaps when devel-

oping GNN models, they are the most widely used frameworks by both the ML community [214] and architecture community. [118, 198, 199, 213]. Table 2.1 presents a comprehensive comparison between PyTorch Geometric (PyG) and Deep Graph Library (DGL), two prominent libraries for implementing Graph Neural Networks (GNNs). Both libraries are actively maintained, with extensive documentation and a large user community, ensuring robust support for developers and researchers in the field of graph-based deep learning.

Table 2.1: Comparison between PyTorch Geometric (PyG) and Deep Graph Library (DGL)

Feature/Aspect	PyTorch Geometric (PyG)	Deep Graph Library (DGL)
Programming Language	Primarily Python	Primarily Python
Deep Learning Framework	Built on top of PyTorch	Supports both PyTorch and TensorFlow
Ease of Use	High-level API, easy to use for beginners	Low-level C++ API available
Performance	Efficient and scalable	Highly optimized for sparse operations
Models Available	Extensive collection of pre-implemented GNN models	Newer models only optimized for PyG
Community and Support	Large community, active development	Relatively new library with community beginning to grow
Documentation	Extensive documentation with examples	Comprehensive documentation
Graph Types Supported	Supports heterogeneous and temporal graphs	Supports heterogeneous graphs
Scalability	Optimized for single-machine, multi-GPU setups	Designed for distributed training on multiple machines and large graphs
Extensibility	Easy to extend and contribute	Low-level C++ API relatively difficult to extend

2.4 Accelerators: An Overview

Accelerators are specialized hardware components designed to perform specific computational tasks more efficiently than general-purpose processors. The primary motivation behind their design lies in addressing the performance and efficiency bottlenecks encountered in conventional

computing systems, particularly for workloads that require intensive data processing or have unique computational patterns. Accelerators are engineered to offload specific tasks from the central processing unit (CPU), thereby enhancing the overall system performance and energy efficiency.

2.4.1 Unique Advantages of Accelerators

The distinct advantages of accelerators stem from their specialized architecture, which is tailored to execute a specific set of operations. This specialization enables several benefits:

- **Enhanced Performance:** Accelerators can execute certain tasks or algorithms much faster than general-purpose CPUs due to their optimized hardware design for those tasks.

- **Energy Efficiency:** By offloading intensive tasks from CPUs, accelerators can reduce overall power consumption, making them ideal for energy-sensitive applications.

- **Parallel Processing Capabilities:** Many accelerators, just like GPUs, are capable of handling multiple operations concurrently, effectively leveraging the parallel hardware of the accelerator.

- **Customizability:** Accelerators can be customized for the specific needs of an application, allowing for greater flexibility in handling diverse computational requirements.

2.4.2 Advantages and Disadvantages of Designing Accelerators

While accelerators offer considerable advantages, there are also trade-offs involved in their design and deployment.

2.4.2.1 Advantages

- **Highly Efficient for Targeted Tasks:** Accelerators provide optimized performance for specific applications, such as graphics processing, machine learning, and now, graph computing.

- **Scalability:** They can be scaled to handle larger workloads more effectively than general-purpose processors.

- **Innovation:** The development of accelerators drives technological innovation, particularly in fields that require high computing power.

2.4.2.2   Disadvantages

- **Limited Flexibility:** Being specialized, accelerators are not as versatile as CPUs for general computing tasks.

- **Development Complexity:** Designing and implementing accelerators can be complex and resource-intensive.

- **Integration Challenges:** Integrating accelerators into existing systems may require significant architectural changes and software support.

- **Cost:** The development and deployment of accelerators can be costly, especially for cutting-edge designs.

Overall, accelerators represent a critical advancement in computing technology, offering specialized solutions for a range of emerging applications. In particular, their role in graph computing has opened new avenues for handling complex data structures and algorithms efficiently. However, the decision to utilize accelerators must consider the balance between their specialized capabilities and the associated design and integration challenges.

Chapter 3

Related Work

In this chapter, we review the existing body of work on graph computing. Our initial focus is on the examination of prior research concerning the benchmarking of graph-based workloads. Next, we discuss previous work aimed at accelerating graph neural networks.

3.1 Graph Computing Benchmark Suites

Past GPU benchmark suites have provided guidance to GPU architects. To date, GPU benchmarks fall into one of two categories. They either evaluate general-purpose GPU computing capabilities [35, 36, 44, 146, 171, 182], or target assessment of the performance of a specific class of workloads [106, 110, 172]. With the growing popularity of DNN workloads, a new wave of DNN benchmarks have been developed.

Benchmarking Deep Learning Workloads and Workload Characterization: Early DNN benchmark suites explored the execution performance of low-level primitives [51, 134], as well as end-to-end inference and training [2, 41]. Later efforts included a more diverse set of DNN algorithms, including a broader range of network models and commercial efforts. TBD [220] is a DNN benchmark suite proposed by Zhu et al. to study DNN training performance on GPUs. AIBench [63] is an industry-initiated benchmark suite that is focused on industrial AI services. Mattson et al. [124] proposed the MLPerf training and MLPerf inference suites. MLPerf adopts ideas from prior DNN benchmark suites to develop an industry-standard DNN-focused benchmark suite, designed so that new hardware and software optimizations can be evaluated fairly. To date, MLPerf has primarily focused on DNNs. In terms of architectural characterization, Dong et al. [50] looked at the architectural implications of CNN training on a GPU. Mojumder et al. [127] profiled

DNN models trained on an NVIDIA DGX-1 system. However, all these prior workload studies were limited to DNNs that operate on euclidean data (e.g., images, video and speech). In this thesis we develop GNNMark. We specifically aim to bridge this gap, providing the architecture community with an appropriate benchmark suite to study GNN training behavior. We also plan to work with the MLPerf consortium to integrate our GNN models into their training suite in the future.

Workload Characterization for GNNs: GNNs have recently attracted attention from the computer architecture community due to their growing popularity in the machine learning domain. Yan et al. [198] have characterized Graph Convolutional Network (GCN) inference performance, focusing on aggregation and model update phases. Zhang et al. [213] have also characterized the inference performance of GNNs. Their work decomposes GNN inference execution into a Scatter-ApplyEdge-Gather-ApplyVertex (SAGA) pipeline and then analyzes the behavior of each phase. They also present insights on how to efficiently design a GNN accelerator for inference. While a benchmark suite is also created as a part of their study, it is designed primarily for inference and is not available publicly. Most prior GNN studies focused on inference, ignoring the training process that tends to consume a large number of GPU hours. Also, the models evaluated are only designed to process homogeneous graphs. Other related work focused on characterizing GNN inference and designing customized accelerators for that purpose [101, 118, 199]. In contrast, GNNMark includes GNN models that work across a wide range of graph data, including spatio-temporal graphs and heterogeneous graphs. GNNMark also includes multi-GPU implementations of GNNs, making it suitable for research on GNN training behavior targeting GPUs. The applications included in GNNMark can also be used to drive inference studies by first training the models to a target accuracy and then using the pre-trained models to characterize inference. We plan to extend the suite to support inference studies by providing a set of pre-trained models in the future.

3.2   Prior GNN Accelerators

The cause for a critical computational bottleneck within GNN workloads is the presence of irregular memory access patterns. Accelerating GNN workloads for large input graphs, especially those with skewed sparsity patterns, is particularly challenging. Previous accelerator designs have attempted to hide memory bottlenecks by performing high-level and low-level pipelining [100], row-remapping [65], generating two distinct implementations for the aggregation and combination phases [199], and leveraging flexible network topologies [112]. While these enhancements improve performance for specific GNN workloads on predefined datasets, they fall short in terms of gen-

Table 3.1: Comparison of prior state-of-the-art GNN accelerators' support for different graph neural network models. The table indicates the support provided by accelerator for GCN, GIN, GAT, SAGE, PNA, and DGN models

Accelerator	GCN	GIN	GAT	SAGE	PNA	DGN
I-GCN [66]	✔	✔	✗	✔	✗	✗
AWB-GCN [65]	✔	✗	✗	✗	✗	✗
HyGCN [199]	✔	✔	✗	✔	✗	✗
EnGN [118]	✔	✗	✗	✔	✗	✗
GraphPE [7]	✔	✗	✔	✗	✗	✗
GNNerator [170]	✔	✗	✗	✔	✗	✗
GCoD [207]	✔	✔	✔	✔	✗	✗
ReGNN [37]	✔	✔	✗	✔	✗	✗
ReFlip [84]	✔	✔	✔	✔	✗	✗
GROW [86]	✔	✔	✔	✔	✗	✗
FlowGNN [153]	✔	✔	✔	✔	✔	✔

eral applicability across a range of GNN workloads [1]. Table 3.1 illustrates support proposed in prior studies for various GNN workloads. Table 3.2 further extends Table 3.1 by summarizing the optimization techniques incorporated by prior GNN accelerators.

I-GCN [66]: I-GCN is a hardware accelerator designed to improve the performance of GCN inference. One of the primary challenges in accelerating GCNs is the poor data locality and redundant computation arising from the large size, high sparsity, and irregular non-zero distribution of real-world graphs. To tackle these issues, I-GCN employs an online graph restructuring algorithm known as islandization. This algorithm identifies clusters of nodes with strong internal connections, but weak external ones, which improves on-chip data reuse and minimizes off-chip memory accesses.

AWB-GCN [65]: Autotuning-Workload-Balancing GCN (AWB-GCN) is a hardware accelerator specifically designed for speeding up GCN inference. Addressing the challenges of processing large and unbalanced real-world graphs, AWB-GCN employs three hardware-based autotuning techniques: dynamic distribution smoothing, remote switching, and row remapping. These techniques enable the system to dynamically adjust the workload distribution across a large number of processing elements.

HyGCN [199]: HyGCN is an accelerator designed to address the unique computational

Table 3.2: Comparison of various Graph Neural Network (GNN) Accelerators on optimization techniques incorporated

Accelerator	Kernel Fusion	Loop Reordering	Pruning	Bank Mapping	Load Balancing
I-GCN [66]	✗	✗	✔	✔	✔
AWB-GCN [65]	✔	✔	✗	✔	✔
HyGCN [199]	✔	✔	✗	✔	✔
EnGN [118]	✗	✔	✗	✔	✔
GraphPE [7]	✗	✗	✗	✔	✗
GNNerator [170]	✗	✗	✗	✗	✗
GCoD [207]	✗	✔	✔	✗	✔
ReGNN [37]	✔	✗	✗	✗	✔
ReFlip [84]	✗	✗	✗	✗	✔
GROW [86]	✗	✔	✗	✗	✔
FlowGNN [153]	✔	✗	✗	✔	✔

challenges arising from the hybrid execution patterns of GCNs. These patterns comprise a dynamic and irregular aggregation phase, and a static and regular combination phase. The design of HyGCN is motivated by a characterization of GCN execution patterns on an Intel Xeon CPU. The accelerator employs a new programming model to exploit fine-grained parallelism and features two efficient processing engines tailored to handle the irregularity in the aggregation phase and the regularity in the combination phase. These engines are optimized for various levels of parallelism and data reusability.

EnGN [118]: EnGN is an accelerator architecture that addresses the substantial computational and memory overhead present within GNN workloads. EnGN focuses on accelerating the three key stages of GNN propagation by abstracting them as common computing patterns. The architecture employs a ring-edge-reduce (RER) dataflow and corresponding RER PE-array to manage the poor locality associated with sparsely and randomly connected vertices. Additionally, EnGN utilizes a graph tiling strategy to fit large graphs into the accelerator's memory.

GraphPE [7]: The GraphPE architecture incorporates dedicated hardware units engineered to manage the irregular data movement typical of graph computations while also ensuring high computational throughput for GNN models.

GNNerator [170]: GNNerator is an accelerator for GNNs, designed to address the com-

putational challenges arising from the dual nature of GNN operations: dense and regular computations for feature extraction, and sparse and irregular computations for message passing between nodes. GNNerator employs heterogeneous compute engines specifically optimized for these contrasting computational patterns. The paper also introduces the concept of feature-blocking, a dataflow technique that adjusts the trade-off between irregular and regular memory accesses, increasing computational efficiency during both feature extraction and aggregation stages.

GCoD [207]: GCoD is a hardware-software co-designed framework aimed at addressing the computational inefficiencies associated with GCNs when applied to large, sparse and irregular real-world graphs. In terms of algorithmic design, GCoD employs a "split and conquer" training strategy that locally polarizes graph densities, resulting in adjacency matrices with enhanced regularity and thus, achieves good acceleration. On the hardware side, a specialized two-pronged accelerator is developed, featuring separate engines to process denser and sparser graph workloads, thereby further improving utilization and acceleration efficiency.

ReGNN [37]: ReGNN is a GNN accelerator aimed at eliminating computational and communication redundancy inherent in traditional GNNs. ReGNN is built on a hardware-software co-design approach incorporating a dynamic redundancy-eliminated neighborhood message-passing algorithm. ReGNN is a configurable, pipelined architecture adaptable to various GNN variants without compromising accuracy.

ReFlip [84]: ReFlip is a GCN accelerator that aims to improve the overall efficiency of both regular neural network computations and irregular graph analytics. ReFlip employs a unified architecture based on Processing-in-Memory (PIM) [94] and features a crossbar structure. This unified architecture is augmented by novel algorithm mappings that maximize performance by leveraging the inherent parallelism of crossbar structures.

GROW [86]: GROW is an accelerator for Graph Convolutional Neural Networks (GCNs), designed to optimize the two primary stages of GCNs—aggregation and combination—that have distinct dataflow. GROW utilizes Gustavson's algorithm to implement a row-wise product-based sparse-dense GEMM accelerator. By co-designing software and hardware, GROW claims to achieve a balance between data locality and parallelism.

FlowGNN [153]: FlowGNN introduces a dataflow architecture tailored for the acceleration of GNNs that utilize message-passing mechanisms. The FlowGNN architecture is scalable and supports a broad spectrum of GNN models, featuring a configurable dataflow that simultaneously computes node and edge embeddings as well as facilitates message passing, making it universally applicable across different models. A significant advantage of FlowGNN is its ability to perform

GNN inference without any prior graph processing.

LISA [115]: LISA is a compiler-oriented approach to map computations of GNNs on Coarse-Grained Reconfigurable Arrays (CGRA) spatial accelerators. CGRAs, known for their potential to enhance computational performance and energy efficiency, require sophisticated compiler designs to unlock their full capabilities. LISA introduces a solution by leveraging GNNs to analyze and interpret the structural characteristics of dataflow graphs (DFGs), which represent application-specific computations. This analysis facilitates the automatic identification of near optimal mappings for DFGs onto new accelerator architectures, considering both node placement and dependency routing. The integration of a simulated annealing-based mapping strategy, informed by GNN-generated insights, ensures that the mapping process is both efficient and effective. LISA dramatically reduces the time required to generate high-quality mappings for spatial accelerators, thereby accelerating the development cycle and enhancing the performance of computing systems.

Chapter 4

GNN Workload Characterization

Owing to the energy efficiency and high-performance capabilities of GPUs, GPUs are a natural choice for accelerating the training of GNNs. This forms the core motivation to understand the architectural and system-level implications of training GNNs on GPUs. Previously to our work, no benchmark suite existed to examine the architectural implications of GNN training workloads.

In this dissertation , we address this need by presenting GNNMark [14], a feature-rich benchmark suite that encompasses the diversity present in GNN training workloads, datasets and GNN frameworks. Our benchmark suite consists of GNN workloads that utilize various graph-based data structures, including homogeneous graphs, dynamic graphs, and heterogeneous graphs commonly used in a number of application domains that we mentioned in Section 2.2. We use this benchmark suite to explore and characterize GNN training behavior on GPUs. We study a variety of aspects of GNN execution, including both compute and memory behavior, highlighting major bottlenecks observed during GNN training. At the system level, we evaluate multiple metric, including the scalability of training GNNs across a multi-GPU system, as well as the sparsity of data encountered during training. The insights derived from our work can be leveraged by both hardware and software developers to improve both the hardware and software performance of GNN training on GPUs. The contributions of this part of the thesis include:

1. **GNN training-focused benchmark suite:** We deliver an open-source benchmark suite named GNNMark (`https://gitlab.com/GNNMark/gnnmark`), designed to characterize the training behavior of GNNs on GPUs. Our suite includes a diverse set of popular GNN models that the machine learning community has developed. The workloads span seven different application domains and three different types of graph-based data types.

2. **Architecture-level characterization of GNNMark:** We characterize the workloads in GN-NMark, considering their architectural implications during the training process on a GPU. We are the first to provide a detailed execution time breakdown of different operations executed during GNN training and identify the major bottlenecks. We find that these workloads are much more diverse than typical DNN training workloads. GNN execution is highly input data and model dependent. We find that integer operations play a critical role, a factor that has been relatively ignored in DNN training studies on GPUs. We also observe significant sparsity during GNN training. This can potentially be leveraged to train larger graphs on a single GPU. We also consider multi-GPU support in the suite, enabling scaling studies of GNNs across multi-GPU systems.

3. **Recommendations to improve GPU architectures:** We present insights drawn from our detailed characterization and suggest changes to improve GPU architectures and system design so that GNNs can be trained efficiently.

4.1 Motivation for Characterizing GNN Workloads

Deep Neural Networks (DNNs) have revolutionized numerous areas, such as image classification [53, 160], speech recognition [49, 141] and autonomous systems [121, 126]. Notable DNN architectures such as Convolutional Neural Networks (CNNs) [61] and Transformers [183] primarily operate on Euclidean data. This type of data, inherently 1D or 2D, includes images and speech datasets [89]. Yet, much of the data we encounter in the real world is non-Euclidean in nature [23], encompassing structures of molecules, social networks, sensor systems, and manifolds. Traditional DNNs, designed for Euclidean data, often fall short in efficiently processing non-Euclidean data due to challenges in directly applying operations, such as convolutions [23, 161].

To bridge this deficiency, GNNs [102, 193] have been developed, specializing in non-Euclidean data training. For instance, Pinterest employs a GNN model, PinSAGE [205], for its recommendation algorithms, while Twitter researchers utilize GNN models with temporal graph data [152]. Similarly, the Drug Repurposing Knowledge Graph (DRKG) [88] adopts GNN models to research the applications of existing drugs for novel diseases.

With GPUs establishing themselves as the go-to platform for DNN and GNN training due to their advanced capabilities, many leading GNN frameworks, for example, the Deep Graph Library [215] and PyTorch Geometric [58], have integrated GPU training support. As GNNs con-

tinue to surge in popularity, there's a pressing need for optimizing GPU platforms to train them. Thoroughly analyzing GPU behavior during GNN training is paramount. By dissecting the myriad of GNN operations and their execution during training, GPU architects can pinpoint and address performance bottlenecks. Aspects such as GNN training scalability on multi-GPU setups and the presence of data sparsity during training can be tapped into for training large graphs, especially those surpassing a single GPU's memory capacity [119, 149, 159]. A thorough analysis of GNN training workflows will enhance our understanding of the computational and memory constraints associated with running GNN workloads on GPUs.

4.2   Prior Work on GNN Characterization

Prior work on characterizing GNNs has focused primarily on the inference behavior for GCNs [198] or targeted a limited set of GNN models [213]. Both model and dataset diversity [213] have not been considered by these studies. By dataset diversity, we mean different types of graphs, including homogeneous, heterogeneous, knowledge, and dynamic graphs (explored in detail in Section 4.3). Model diversity implies different types of GNN models, such as Graph Transformers, Spatio-Temporal GNNs, and LSTM based GNNs. In previous benchmarking efforts, GNN inference has been the primary target for characterization studies [198, 213]. Popular benchmark suites for DNN training, such as the MLPerf Training Suite [123], Training Benchmarks for DNN (TBD) [220], DNNMark [52], Fathom [2], and DawnBench [41], do not consider GNNs as part of their workloads and deal exclusively with DNNs that deal with euclidean data. To comprehensively characterize the execution behavior of GNN training on GPUs, we need a benchmark suite that includes diverse GNN models that are trained on diverse datasets. Currently, no such benchmark suite exists. To fill this gap, we develop **GNNMark**, a collection of representative workloads that can be used by the computer architecture community to study the execution of GNNs on GPUs. We then analyze the workloads in the GNNMark benchmark suite, specifically focusing on their behavior during GNN training on a GPU. Apart from the fact that prior GNN workload characterization studies primarily focused on inference, we chose to focus on training, given that GPUs remain the best platform in terms of performance for GNN training.

4.3 Input Graph Types

GNNs have evolved over time, and we find that each variation of GNN is typically associated with a distinct form of graph data [193]. In our examination of GNNs, we identify three primary classifications of graph data:

1. **Homogeneous Graphs:** A homogeneous graph contains nodes and edges of a single type. For example, social network graphs are typically homogeneous, where each node represents a user, and an edge can represent if that one user follows another. Homogeneous graphs can be directed (e.g., following a user on Twitter) or undirected (e.g., adding a friend on Facebook). Another notable collection of homogeneous graph datasets that are used to evaluate GNN models are citation datasets (e.g., Cora, PubMed, Citeseer) [102].

2. **Heterogeneous Graphs:** A heterogeneous graph contains nodes and edges of multiple types. A widely used form of a heterogeneous graph is found in recommendation generation scenarios. For example, in a dataset designed to recommend music to users, the graph will consist of two types of nodes: i) music nodes and ii) user nodes. The edges will correspond to different interactions between the user and a music piece. In addition, edges may contain additional information such as ratings or like/dislike attributes. Knowledge graphs that are used to model relations between an object and entities are another form of a heterogeneous graph – e.g. when users search for a famous celebrity on Google (an object).

3. **Dynamic Graphs:** A dynamic graph is a special type of graph where the graph itself, as well as its properties, can evolve over time. Many real-world graphs, such as social-network graphs [152], traffic data graph [208] and communication-network graphs, are dynamic [56]. Note that dynamic graphs can be either homogeneous or heterogeneous. For example, if we take a homogeneous social network graph, where nodes represent people and edges represent whether there is a relationship, the number of relations a person has or the relations between two people can change over time. Another common use case of dynamic graphs is to model traffic data as a dynamic graph and use it for traffic forecasting and prediction [48].

4.4 Benchmark Suite Design

Characterizing the behavior of GNN training on a GPU requires a set of representative workloads to cover the wide variety of GNNs [193, 214]. The variants should include GNNs

Abbv	GNN Model	Application Domain	Graph Input Type
PSAGE	PinSAGE	Recommendation	Heterogeneous Graph
STGCN	Spatio Temporal GCN	Traffic Forecasting	Dynamic Graph
DGCN	Deep GCN	Molecular Property Prediction	Homogeneous Graph
GW	GraphWriter	Text Generation	Heterogeneous Graph
KGNN	k Graph Neural Networks	Protein Classification	Homogeneous Graph
ARGA	Adverserially Regularized Graph Autoencoder	Node Clustering	Homogeneous Graph
TLSTM	Tree Long Short-Term Memory Networks	Sentiment Classification	Homogeneous Graph

Table 4.1: Workloads in GNNMark Benchmark Suite.

used across multiple application domains, including recommendation systems, classification of molecules, traffic forecasting, etc. The representative suite should also include models that consider different classes of real-world graphs, including knowledge graphs, heterogeneous graphs, and dynamic graphs. In addition, multi-GPU GNN training should be supported to evaluate the efficacy of training GNNs on multi-GPU systems.

To satisfy all the above-mentioned criteria, we offer GNNMark, a benchmark suite designed for studying the behavior of GNN training on GPUs. Similar to benchmark suites that target DNN training, such as TBD [220] and MLPerf [123], we curate our benchmark suite from open-source publicly available implementations of GNN models. As PyTorch Geometric (PyG) and Deep Graph Library (DGL) are the main frameworks employed for developing GNN models by the ML community, we use models developed using these frameworks. Since both of these frameworks support PyTorch, we have chosen models developed in PyTorch. The specific models chosen for this suite, along with their associated application domains and datasets, are summarized in Table 4.1. Below, we provide more details about each GNN model.

PinSAGE: GNNs that operate on heterogeneous knowledge graphs can be used for recommendation tasks. These are commonly used in social networks. PinSAGE [205] is one such GNN model that has been developed at Pinterest. Since the original PinSAGE model is not publicly available, we use the implementation that has been published by the developers of DGL. PinSAGE is an improvement upon the GraphSAGE model [74] for training on large graphs. It uses a *random walk* mechanism [188] during aggregation to identify the importance of a node in the graph without

Abbv	Datasets	# Node	# Edge
PSAGE	Nowplaying (NWP) [209]	22.9M	1.9M
	Movielens (MVL) [75]	1.9M	9.7K
STGCN	LA [208]	207	325
	PEMS_Bay (PEMS) [208]	1722	2694
DGCN	MOLHIV [83]	1.04M	1.1M
	MOLTOX [83]	145K	151K
GW	AGENDA [105]	885K	2.57M
KGNN	Proteins (PROT) [96]	43K	162K
ARGA	Cora [202]	2K	10.5K
	CiteSeer (CSEER) [202]	3.3K	9.2K
	PubMed (CSEER) [202]	19.7K	88.6K
TLSTM	Stanford Sentiment Treebank (SNTM) [164]	318K	310K

Table 4.2: Workloads in GNNMark Benchmark Suite.

the need to process the entire graph. This effectively allows a user to train a model on graphs that do not fit in GPU memory.

Spatio-Temporal Graph Convolutional Network: Traffic forecasting is an important problem that falls into the domain of time-series prediction and uses dynamic graphs. This task is highly relevant for use in urban areas where traffic control and guidance are required. Solving this problem using conventional Euclidean-based DNNs is challenging because of the nonlinearity involved in traffic data [133]. One approach to deal with nonlinearity is to represent the problem as a graph and then apply depth-wise convolutions on the graph. Spatio-Temporal graph Convolutional Networks (STGCN) [208] represent one such model that has been proposed to solve the problem of traffic forecasting. We include an STGCN to represent a GNN model that deals with dynamic graphs.

DeepGCNs: One of the key challenges with the original GCN models, such as the one proposed by Kipf and Welling [103], is that increasing the depth of the model does not improve the accuracy of the model. This is due to the vanishing gradient problem [80], which has made implementing deep GCNs challenging. Therefore, researchers have developed mechanisms to train deeper GCNs [111], using ideas borrowed from DNN research, such as residual layers and skip-connections used in models such as ResNet [78]. DeepGCN is a novel GCN architecture that

allows GCNs to have more layers. Additional layers in a GCN can significantly improve training accuracy [111], so we include it in our study. Specifically, we use a DeepGCN model and train it to perform graph property prediction, a common task in molecular property prediction.

GraphWriter: Automated generation of text from a knowledge graph to form meaningful and coherent sentences is an open and challenging problem [90]. Text encoding models, such as the popular Transformer model [183], cannot be directly applied to a knowledge graph as they do not work with non-euclidean data. Therefore, ML researchers have developed GNN-based Transformer models for this task. Graphwriter [105] is one such novel GNN-based Transformer model designed to operate on knowledge-graphs for text generation.

k-GNNs: Most GNN models are one-dimensional in nature and cannot effectively capture any higher-order information, such as the properties of subgraphs, within the graph. As a result, they fail the graph isomorphism test proposed by Weisfeiler and Lehman [189] (WL algorithm). The WL algorithm is a test used to determine the expressiveness power of a GNN by testing if an algorithm is able to distinguish whether two graphs are isomorphic or not. Two graphs are said to be isomorphic if they have the same number of vertices, edges, and connectivity. Therefore, researchers have developed higher-dimensional hierarchical GNNs, called k-GNNs (where the k stands for the dimension), which can capture properties of subgraphs [128]. This enables GNNs to perform close to the k-WL graph isomorphism test [128]. We include two variants of k-GNNs, (KGNNL and KGNNH to denote a lower and higher dimensional version of k-GNN, respectively) and use them to perform classification of protein molecules. The primary reason we include this workload in our suite is to study how application characteristics and behavior change as we move towards higher-dimension GNNs.

Adversarially Regularized Graph Autoencoder: Generative Adversarial Networks (GANs) are gaining popularity due to their ability to learn with limited amounts of data [144]. Due to this property, GAN-based architectures are also being explored for GNNs. An Adversarially Regularized Graph Autoencode (ARGA) [143] is one such GNN-based GAN model that is proposed for graph embedding. ARGA has an encoder-decoder architecture where the encoder is trained to form a compact representation of a graph, and the decoder is trained to generate the graph structure. The model is designed to perform node clustering, which is an unsupervised learning task, on real-world graphs. ARGA employs this encoder-decoder architecture within a GAN framework, so that it can successfully learn the low-dimensional features of the graph from the high-dimensional graph features. This process is referred to as graph embedding [28]. We include ARGA as a representative GAN-based GCN to further increase the diversity of our benchmark suite. We train

ARGA to perform node clustering on real-world homogeneous graphs, such as Cora, PubMed, and CiteSeer [103].

Tree-LSTM: Sentiment classification is an important task in the Natural Language Processing (NLP) domain. Tree Long Short-Term Memory Networks (Tree-LSTMs) [173] are one group of models developed for this task. In contrast to the linear model used in an LSTM, Tree-LSTMs use a tree-structured network topology and can outperform linear LSTMs in the sentiment classification task [87, 173]. The Tree-LSTM method implemented in DGL uses the idea of batching. The basic idea of batching is to collect smaller graphs that are part of the dataset and convert them into a batched larger graph. We include the Tree-LSTM model in GNNMark to study how batching multiple small graphs to a larger graph impacts the behavior of an application.

4.5 Profiling Methodology

4.5.1 Experimental Platform

To demonstrate the utility of GNNMark, we use an NVIDIA V100 [139], a commonly used GPU for running neural network training. V100 is part of the NVIDIA Volta family of GPUs. Our test system is equipped with an Intel(R) Xeon(R) CPU E5-2630 CPU that operates at a frequency of 2.4GHz. The GPU has 80 Streaming Multiprocessors (SMs) and is rated to deliver 14 TFLOPS of single-precision performance. The GPU memory uses HBM2 with 16 GB capacity and bandwidth of 900 GB/s. The combined L1 cache/shared memory/texture cache has a capacity of 128 KB and is private to each Streaming Multiprocessor (SM). The L1 memory is backed by a 6.14 MB L2 cache, which is banked and shared across all SMs.

For our multi-GPU experiments, we use 4 V100 GPUs on a node equipped with Intel(R) Xeon(R) E5-2686 v4 2.4GHz CPUs, hosted on Amazon AWS EC2. Each GPU is interconnected using NVIDIA NVLink technology, providing a total of six links, for an aggregate bandwidth of 300 GB/s. Both the single-GPU and multi-GPU systems used in our experiments run CUDA 10.2, cuDNN 7.6.5, and PyTorch 1.5.0. The workloads included in GNNMark use either DGL version 0.5.2 or PyTorch Geometric 1.6.1.

Since multi-GPU training has been shown to improve the performance of DNN training [127], we also look at how well GNN training can scale across multiple GPUs. GNN training can be sometimes be limited by GPU memory capacity, especially given the continual growth in the size of the input graph [91]. One approach to counter this problem is to compress the data trans-

ferred from the CPU to the GPU and store the compressed data in GPU main memory. This is only possible if the data transferred is highly sparse [149]. Therefore, we also characterize sparsity levels of the data transferred between the CPU and GPU during GNN training in our suite.

4.5.2 Profiling Tools

We use several tools for collecting the metrics of interest. For the kernel-level characteristics, such as cache statistics and comparisons between compute versus memory behavior, we use the NVIDIA nvprof profiler (version 10.2) [22]. Similar to DNNs, GNNs typically launch the same kernel many times during training. Therefore, when profiling and collecting hardware performance counters using nvprof, we profile the same kernel for either fifty kernel invocations or for one epoch, whichever is shorter. However, nvprof does not provide any mechanism to collect the memory divergence behavior of a workload. Therefore, we use the NVBit framework [184] (version 1.4), which is a binary instrumentation tool to collect the memory divergence information at a kernel level. To collect the sparsity of the data transferred from the host to the device during GNN training, we modified the PyTorch source code to collect this information.

4.5.3 Metrics of Interest

Characterizing the behavior of GNNs requires an understanding at both the architectural level and the system level. In this work, we profile and collect the following metrics:

1. **Ratio of time spent in different operations:** Prior work on classifying the phases of GNN execution have categorized GNN execution into two phases: i) aggregation and ii) update phases [198]. While classifying into these two phases is beneficial for machine learning purposes, we believe that architectural studies can be guided by a lower level of abstraction (i.e., operations), which has been proposed by Adolf et al. [2]. In our profiling experiments for GNN, we observe commonly used operations across various GNN workloads, such as sparse matrix GEMM operations (spgemm), scatter and gather operations, reduce operations, embedding operations, index selection operations, sorting operations, and element-wise operations. These operations may be embedded within one or multiple kernels within GNN training workloads. Understanding the time spent in these different operations and how they vary across different datasets for the same model can shed insight into where the majority of the execution time is actually spent during GNN training on a GPU.

2. **FLOPS and Arithmetic Intensity Analysis:** To understand how well the GPU can handle GNN training, it is important to analyze the arithmetic intensity and count the number of floating point operations (FLOPS). Arithmetic intensity (AI) is defined as the ratio of the total number of floating point operations performed to the ratio of data transferred (in bytes). AI can be used to gauge the data reuse of an algorithm. A higher AI is better since it implies more computations are performed for every byte of data. Analyzing the FLOPS vs AI shows whether a workload is mainly compute or memory bound.

3. **Stall Analysis:** To improve the performance of GNNs on GPUs, GPU application developers need to have an understanding of major stalls incurred during GNN training of different GNN workloads. Such an understanding of stalls at an operation level can be helpful to understand the performance of each aforementioned operation.

4. **Cache behavior:** While not as important as they are for CPUs, caching can still benefit GPU applications with high spatial and temporal locality. Therefore, having a basic understanding of the hit rates of different levels of the cache is important. Another key characteristic relevant to caches is memory divergence. Memory divergence demonstrates the scattered memory access pattern of a given operation. The memory divergence of a single transaction is calculated by the number of unique cache lines that are touched by a warp. For example, if each of the 32 threads in a warp accesses a different cache line, then the divergence is 32. A scattered memory access pattern i.e., where threads in a warp end up accessing different cache lines, is detrimental on GPUs as the memory transactions cannot be coalesced. This, in turn, leads to serialization and can hurt performance. It is well known that memory divergence hurts the performance of typical graph workloads such as Breadth First Search and PageRank [35]. Therefore, it is essential to understand the level of memory divergence within a broader class of GNN workloads.

5. **Sparsity during GNN training:** Sparsity of the data that is transferred between the CPU and the GPU during training can be leveraged to use optimizations such as DMA compression as proposed by Rhu et al. [149]. Therefore, in this work, we also aim to understand the sparsity and compressibility during GNN training.

Figure 4.1: Execution breakdown, reported as the percent of total execution time, for individual operations across the different workloads of GNNMark.

4.5.4   Multi-GPU Implementations

We also include multi-GPU versions of each workload in GNNMark to enable users to study the scalability of GNN training on multi-GPU systems. The multi-GPU implementations are built on the PyTorch `Distributed Data Parallel` (DDP) method to train GNNs across multiple GPUs, exploiting data-level parallelism. In practice, DDP has been shown to scale well on up to 256 GPU nodes [114] for DNN training.

4.6   Benchmarking Results

4.6.1   Execution Time Breakdown

We start our analysis by breaking down the time spent in the different GNN operations across the different workloads in our suite. Similar to DNNs [2], GNN training can be broken down into layers or operations. Prior work divided GNN training into two phases: i) an aggregation phase, and ii) an update phase [198]. While this division is appropriate when looking at GNNs from an machine learning perspective, we believe that deeper insights are needed to fully characterize their behavior. Therefore, we work at the abstraction level of individual operations [2].

We identify a common set of operations performed during GNNMark execution. These operations include general matrix multiply (GEMM), sparse matrix-matrix multiplication (SpMM), convolutions, scatters, gathers, reductions, index selection, sorting, and element-wise operations. Element-wise operations are operations that operate on individual elements of a tensor and perform operations such as multiplication of all elements in the tensor by a scalar, changing the sign of all elements in the tensor, or adding two tensors of similar dimensions.

Figure 4.1 shows a breakdown of the percentage of time spent in individual operations across the different workloads of GNNMark. Figure 4.1 illustrates the percentage breakdown of operations varies significantly across workloads. For instance, STGCN, a spatio-temporal GNN, is dominated by 2D convolution operations (60% on average), while DGCN is dominated by element-wise operations (31% on average).

The execution time breakdown across operations in a GNN differs greatly from the mix in a typical DNN. Across all the workloads, we observe that only 25% of the execution time is spent executing GEMM and SpMM operations. This is in stark contrast to the mix of operations commonly found in DNN workloads, where GEMM (convolutional layers and fully-connected layers) dominate the execution [51]. We find that GNN training also differs significantly from GNN inference workloads [198], where GEMM operations are reported to consume more than 50% of the execution time.

Other common operations, such as sorting, index selection, reductions, and scatter-gather operations, account for 20.8% of the total execution on average. These operations are primarily used in the graph's aggregation phase, where the nodes exchange information with one another before updating the feature vectors.

PSAGE, when trained on the MVL dataset, spends 20.7% of its execution on sorting and only 7.0% of the time on reductions, whereas ARGA (using the Cora data) spends 23% on reductions and only 6.1% on sorting. This great diversity and variety of tasks in GNN training present challenges to architects designing customized accelerators for GNN training, given that accelerators are typically designed to optimize only for a single set of operations.

In contrast to typical DNN workloads, GNN workloads tend to be more input data-dependent. For PSAGE, the percentage of time spent in element-wise operations is much higher when training on the (NWP) dataset (78%), versus training on the (MVL) dataset (36%). This is because, when training on the NWP dataset, the feature vectors are $10\times$ larger than when training on the MVL dataset. As element-wise operations operate on each value of the input feature vector, the time spent executing these operations becomes more dominant when graphs with larger input features are used.

GNN Execution Characteristics: Our performance analysis shows that GNN training workloads exhibit more diverse behavior as compared to DNN training workloads. Each model's characteristics can differ vastly from others. Even the same GNN model can exhibit different characteristics depending on the input graph type. In addition, execution hot spots are no longer limited to convolu-

tion and GEMM operations. We find operations such as reductions, scatter, gather and sorting also need to be optimized. The solution of attaching a single-purpose accelerator to primarily accelerate GEMM operations [147] during DNN training may not work well for GNN training.

4.6.2 Instruction Mix and GFLOPS/GIOPS Analysis

Another aspect of GNN training behavior is the dynamic instruction mix present in different workloads. As shown in Figure 4.2, integer instructions play a larger role than floating-point instructions across all workloads. On average, 64% of the executed instructions are integer (int32) instructions, whereas only 28.7% are single-precision floating point (fp32) instructions. The only workload where this trend is reversed is in GraphWriter (`GW`). This is because, in `GW`, a majority of the time is spent on GEMM and SpMM operations (as seen in Figure 4.1), which work on fp32 data. While improving the performance of fp32 instructions has received much attention, int32 instructions have not received the same. Given that int32 instructions dominate GNN execution during training, improving the performance of integer math on a GPU is a critical factor when trying to accelerate GNN training.

Figure 4.3 presents the number of GFLOPS and GIOPS executed by our workloads in GNNMark. We observe that the average GFLOPS rate is 214 GFLOPS, and the average GIOPS rate is 705 GIOPS. The observed average GFLOPS rate is much lower than the theoretical max GFLOPS of the V100, which is 14 TFLOPS for fp32 arithmetic [139] (the V100 specs do not mention the peak theoretical GIOPS. We believe it to be close to the peak theoretical GFLOPS). `GW` has the highest fp32 performance of 1.99 TFLOPS. Being a transformer-based ML model, `GW` can effectively use most of the parallel resources on a GPU [183]. We also observe that, while graph batching has been proposed to improve performance in DGL, `TLSTM` is still able to achieve only 74 GFLOPS.

The average IPC measured across all the workloads was found to be 0.55, which reflects the memory-bound nature of the workloads. When comparing the GFLOPS and GIOPS of different operations, we observe that the GEMM operations typically have a higher GFLOPS (in the mid 300s) as opposed to other operations, such as reductions, scatters, and gathers that have lower rates (in the 100 GFLOPS/GIOPS range) suggesting a very low overall GPU utilization. Given that these operations can dominate the execution time, it is important for both hardware and software developers to focus on improving the performance of these operations.

Instruction Set Usage Summary: Our analysis reveals that, during GNN training, execution is dominated by integer operations. Thus, to accelerate GNN training on either GPUs or accelerators, int32 arithmetic performance will be key. The overall performance in terms of GFLOPS/GIOPS for GNNs is relatively low compared to the peak performance of the hardware. This suggests that GNN training is primarily memory-bound. Given that operations such as reductions, scatters, gathers, and sorting can occupy a good chunk of the execution time during GNN training, it is important for both hardware and software developers to focus on improving the performance of these operations.

4.6.3 Stalls and Cache Analysis

Developing a comprehensive understanding of major stalls in the GPU hardware during GNN training can help guide architectural design decisions when tuning the performance of these workloads. Given that caches can greatly improve the performance of GPU applications, it is also important to look at their efficiency in the context of GNN training. In Figure 4.4, we provide a distribution of different types of stalls observed in GNN training. We find that execution is stalled primarily due to *Memory Dependency*, *Execution Dependency*, and *Instruction Fetch*. The high percentage of *Memory Dependency* stalls (34.3% on an average) suggests that the memory subsystem is inefficient in serving data read requests to the GPU cores. From Figure 4.5, we observe that GNN workloads have an extremely low L1 D-cache hit rate on the V100 (a mere 15%, on average), which is the primary reason for these stalls.

We also analyze the impact of divergent load instructions. The load instructions associated with a warp are considered divergent if they access more than one cache line (a line is 128B on the V100). Memory divergence can impact the performance of typical graph workloads, such as Breadth First Search and PageRank [35]. Therefore, it is important to characterize the degree of memory divergence present during GNN training.

Of all the load instructions, we observe 32.5% of load instructions exhibit divergence across different GNN training workloads. This percentage is large and is highly correlated with the resulting low L1 D-cache hit rates. While the larger L2-cache (6MB) on the V100 fares significantly better with a 70% hit rate on average, the inability of the L1 D-cache to effectively hold the working set can put pressure on the L2 cache to satisfy the memory requirements. Across the different operations, we observe that GEMM, SpMM, and GEMV have poor locality (i.e., a low L1 D-cache
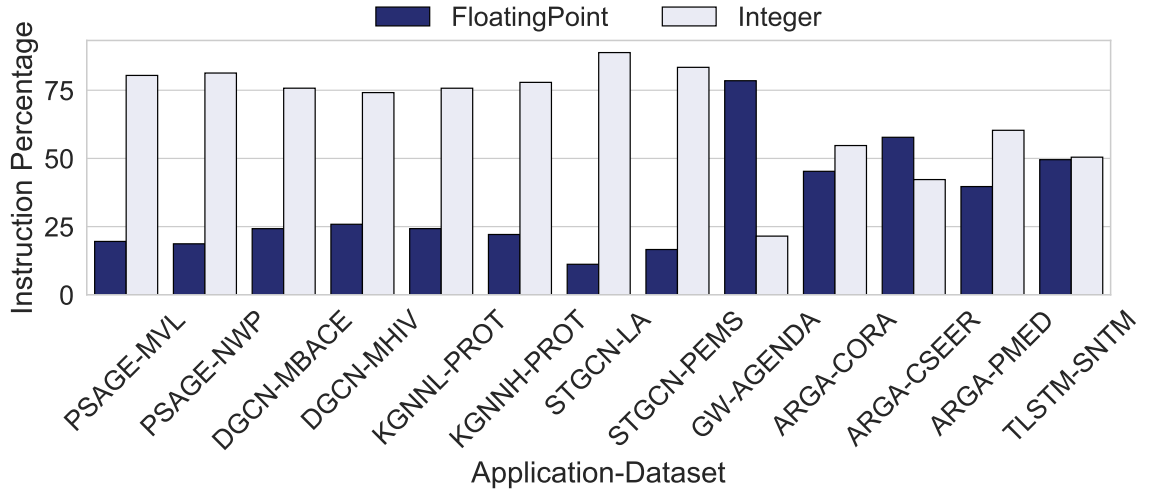
Figure 4.2: Breakdown of fp32 vs. int32 instructions across the different workloads in GNNMark.
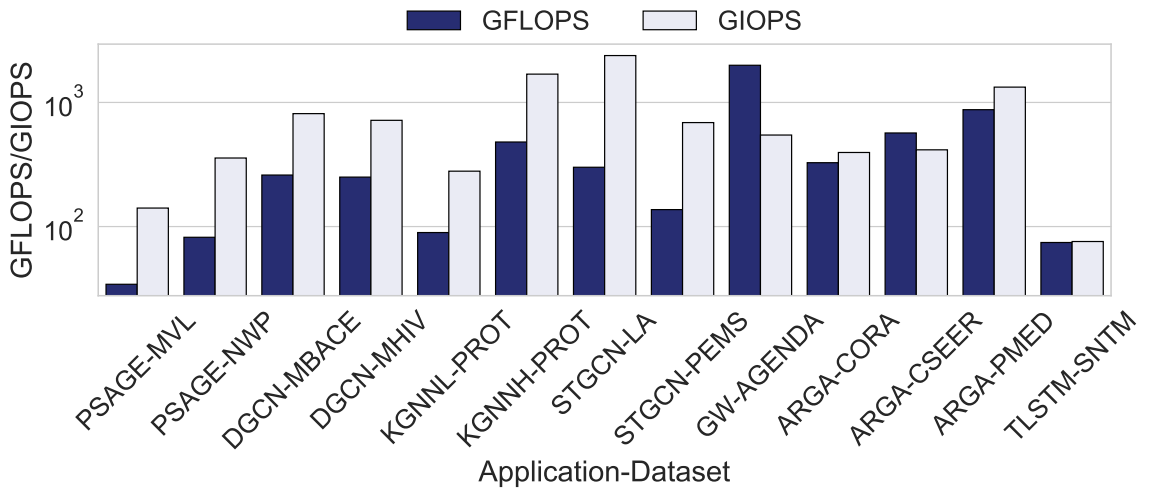
Figure 4.3: GFLOPS and GIOPS across the different workloads in GNNMark.

hit rate, less than 10% on average). The L1 D-cache hit rates of other operations, such as indexing, scatters, gathers and sorting, are also low (below 15%, on average).

The high percentage of *Execution Dependency* stalls (i.e., 29.5% of the stall cycles on an average) points to the fact that, across the entire set of workloads, there are many dependencies between instructions in a warp, which results in low instruction-level parallelism. Microarchitectural enhancements to support out-of-order execution in the GPU pipeline [70] can potentially accelerate GNN training.

Surprisingly, *Instruction Fetch* stalls are also significant (21.6% on average). This is due
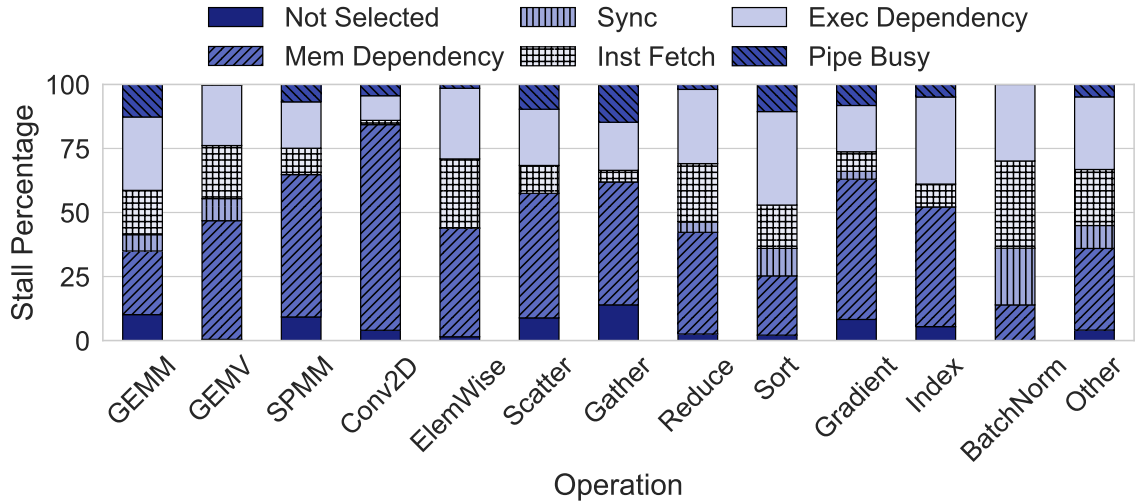
Figure 4.4: Stall breakdown across operations in GNNMark.

to two reasons. The first is that the instruction cache is ineffective in caching all the instructions. Although the V100 architecture has a new 12KB L0 I-cache that is backed by a larger 128KB L1 I-cache, it seems to not be highly effective in caching all instructions during kernel execution. The second reason is loop unrolling techniques [131], which are used to improve the performance of a GPU kernel, can negatively impact the instruction cache hit rate and increase the stalls due to instruction fetching [45].

In Figure 4.4, it is evident that scatter and gather operations, along with index selection operations, exhibit a higher frequency of stalls in comparison to GEMM, particularly for commonly utilized GNN operations (notably, Conv2D for STGCN and BatchNorm for DeepGCN). The primary reason for this is the irregular memory access patterns demonstrated by both scatter and gather, as well as index selection operations, which lead to memory dependencies.

Takeaways: Our analysis of the stalls during GNN training shows that stalls due to *Memory Dependency*, *Execution Dependency*, and *Instruction Fetch* can be significant. While GPU architecture research has focused on removing the first two types of stalls, improving the performance of instruction fetching has been neglected. Therefore, architects and compiler developers should focus on developing techniques to improve instruction fetch to optimize the performance of GNN training.

GNN training also suffers from a high degree of L1 D-cache misses and a significant number of divergent load instructions across all operations. The extremely high L1 D-cache miss rates suggest that caching is not effective for GNN workloads. We envision two potential solutions
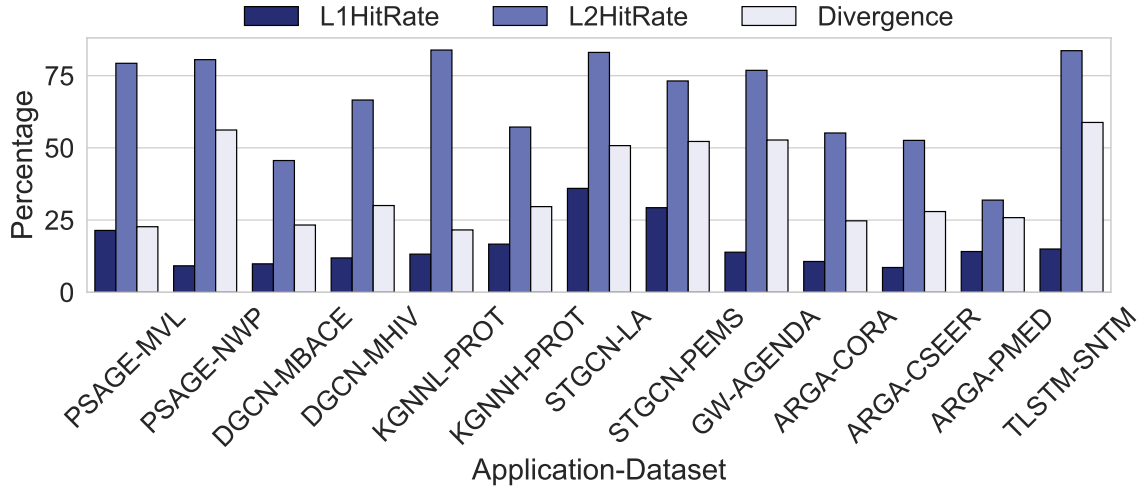
Figure 4.5: L1-D and L2-cache hit ratios, and divergent load ratios for GNNMark workloads.

to alleviate this problem. The first is to employ half-precision-training for GNN training, which uses only 16-bit data instead of 32-bit data, thus can significantly reduce the L1 D-cache miss rates. Alternatively, L1 cache bypassing solutions [194, 195] can be explored to alleviate this problem. Among all the load instructions, 32.5% exhibit divergence across various GNN workloads.

4.6.4   Sparsity during GNN training

Training sparsity refers to the zero values (as a percentage of all values) that are transferred during CPU-to-GPU memory copies during the GNN training process. For characterizing the average sparsity, we report the percentage of zero values observed in CPU-to-GPU data transfers during GNN training. From Figure 4.6, an average sparsity of 43.2% was observed during GNN training. This suggests that compression techniques could be employed. Rhu et al. [149] proposed using compression to alleviate the problem of training large DNN models on a GPU. While GNN models are smaller than conventional DNN models (e.g., Resnet-50 is 50-layers deep, whereas most GNNs today have fewer than 10 layers), the input graph can occupy a significant portion of GPU memory (up to 90% in our experiments). While the machine learning community has proposed sampling the graph to address this problem [205], there are situations where training on the whole graph has been shown to provide better accuracy [91]. We suggest compressing the data in GPU memory to facilitate training on large graphs.

We can also observe a predictable pattern in the data sparsity (from Figure 4.7), providing

opportunities to apply adaptive compression algorithms [176]. As the sparsity values change during training, the GNN training framework may need to exploit different compression solutions and formats that work the best for a specific sparsity level.

Looking at the average sparsity for `PSAGE` in Figure 4.6, we can conclude that training sparsity is a function of both the model and the input graph. When using the MVL dataset, the average sparsity is 22%, but it reduces to 11% when training on the NWP dataset.

In terms of models, since many GNNs such as GraphTransformer, DeepGCNs and ARGA use activation functions such as ReLU and PReLU in their layers, they produce highly sparse data. We suggest applying compression to take advantage of this sparsity. The result will be that we can train larger graphs on a single GPU. We plan to pursue this path in later work in this thesis.

Takeaways: Training on graphs that are larger than the size of GPU memory is a challenging problem. Thus, exploiting the high degree of sparsity present in GNN workloads by using compression techniques can begin to address this problem.

4.6.5   Scalability of GNN training using multi-GPU systems

Using the multi-GPU implementations that we developed for the GNNMark workloads using PyTorch DDP, we evaluate the strong scaling characteristics of the workloads in GNNMark. We train all our models for five epochs (we observe similar performance across all epochs) and report the average `time-per-epoch`, an approach used in previous work [127], to understand the performance of DNN workloads on multi-GPU systems. We do not evaluate ARGA, as the application inherently sends the entire graph to the GPU as a part of its training process, and therefore, distributing the same graph across multiple GPUs does not help. The first thing we can clearly observe from Figure 4.8 is that not all workloads benefit from multi-GPU training. While DGCN, STGCN, and GW show considerable performance gains, the same does not hold true for the other applications. TLSTM does not benefit from multi-GPU training. Given that this is an LSTM-based GNN model with low computational GFLOPS/GIOPS intensity, the application is unable to take advantage of the additional computing power offered by multi-GPU systems. For PSAGE, we observe performance degradation when scaling across multiple GPUs. This is primarily because the PSAGE implementation in DGL uses a batch sampling mechanism, which is not compatible with PyTorch DDP. As a result, the training data gets replicated across multiple devices, and this replication results

49
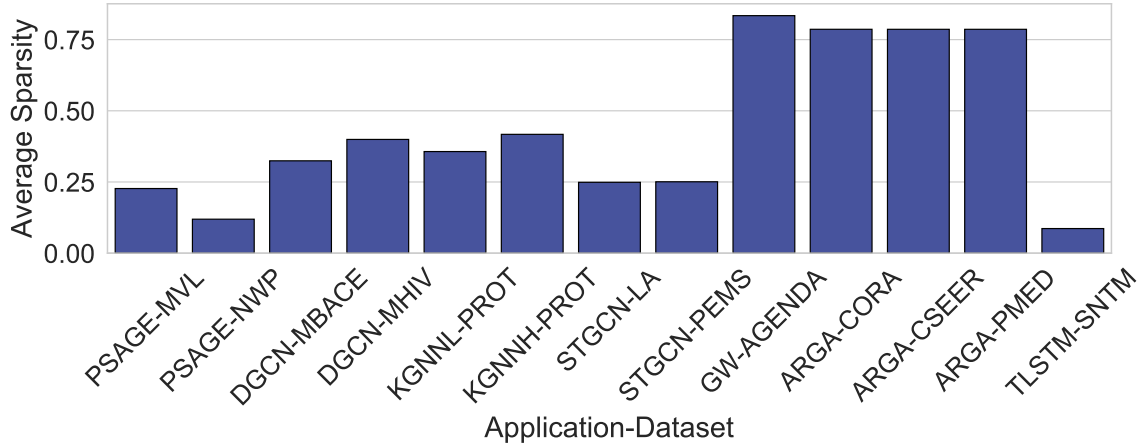
Figure 4.6: Average sparsity in the data transferred from CPU-to-GPU during GNN training in GNNMark workloads.

in redundant computation and unnecessary communication, which in turn hurts performance.

Takeaways: Multi-GPU systems do not always benefit GNN training. Therefore, ideas such as topology-aware scheduling and fine-grained graph partitioning that have been proposed by researchers in graph-centric GNN frameworks, such as ROC [91] and NeuGraph [120], should be adopted by high-level frameworks, such as PyG and DGL, to enable more efficient GNN training. Currently, these frameworks are not open source, and hence, we cannot evaluate them for the GNNMark workloads.

4.7 GNNMark Summary

In this dissertation , we present GNNMark, a diverse benchmark suite of GNN workloads designed for the characterization of GPU performance. To the best of our knowledge, we are the first to propose a GNN training focused benchmark suite for the architecture community. We use GNNMark to perform a detailed characterization of GNNs to understand the architectural implications of training on GPU systems. Our work provides novel insights that show the major architectural bottlenecks in GNN training and suggests how they can be potentially addressed.
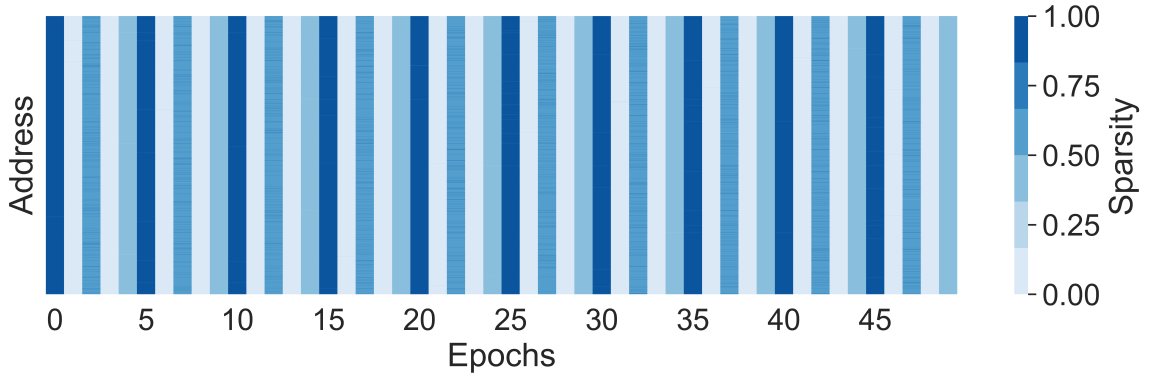
Figure 4.7: Sparsity heat map for DeepGCN when running on the MOLHIV dataset.
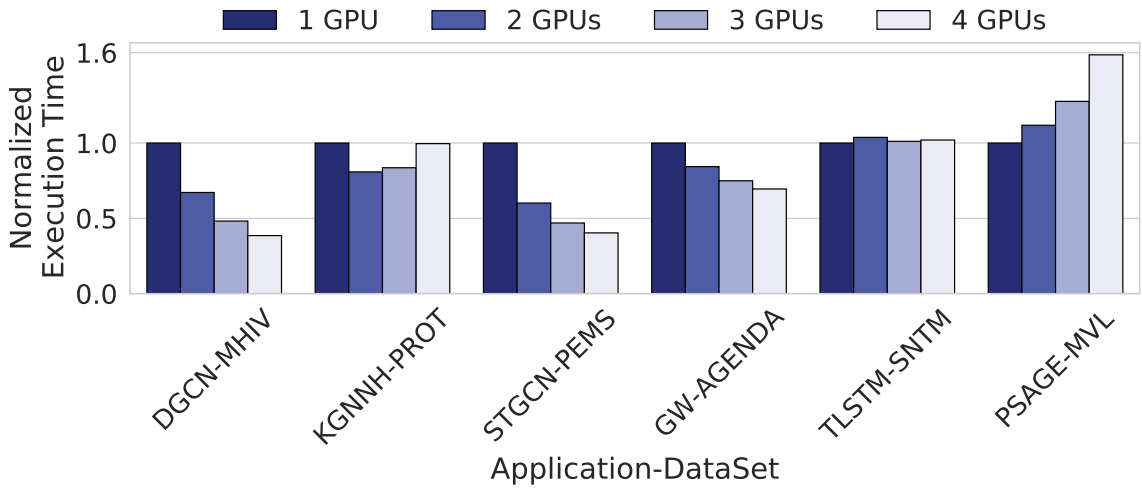
Figure 4.8: Multi-GPU performance scaling.

A single GNN model can exhibit different characteristics based on the input graph. We observe that unlike DNNs, GEMM and convolution operations are less dominant in GNN execution. Instead, integer operations required for graph processing can dominate execution, suggesting that improving the performance of integer math is paramount. A high degree of instruction fetch stalls shows that the instruction cache on the GPU can limit GNN performance. Finally, we also report on the training sparsity and strong scaling characteristics of GNN training using our suite.

51

Chapter 5

Algorithmic Strategies for GNN Acceleration

Graph Neural Networks (GNNs) are characterized by their computational structure, which involves dense matrix operations in the combination phase and sparse matrix operations in the aggregation phase. This chapter focuses on the acceleration of the combination phase through the development of an accelerator specifically designed for Sparse General Matrix-Matrix Multiplication (SpGEMM). Subsequently, in the following chapter, we introduce an accelerator designed to efficiently manage both the sparse (combination) and dense (aggregation) computational phases of GNN workloads, thereby establishing a comprehensive and versatile GNN accelerator.

Optimizing the performance of multiplications that operate on sparse matrices is challenging, especially given the associated irregular memory access patterns, resulting in load imbalance on today's parallel architectures. Given that the input matrix data possesses low temporal and spatial locality, this leads to inefficient cache usage and pipeline stalls. Modern-day CPUs and GPUs struggle to produce scalable performance when executing sparse matrix multiplication workloads. CPUs fail to monetize on the parallelism present in such workloads, while GPUs struggle to balance tasks across their thousands of hardware threads. While a number of sparse matrix formats have been proposed to better handle the sparsity, we lack a single format that works over a range of different sparsity patterns. Given the growing popularity of sparse datasets in emerging applications, we need to explore how we can leverage a novel architecture to accelerate SpGEMM workloads.

As part of this dissertation , we explore a novel distributed memory SpGEMM implementation. We specifically target this work for a custom accelerator. Our approach improves perfor-

mance by mapping the computation of row-wise products, hashtables, and on-chip accumulation to the accelerator's scratchpads. We provide a new set of performance metrics for this class of workload and demonstrate their utility using a suite of micro-benchmarks run using synthetic, as well as real-world, datasets. We also introduce a new Memory-aware Aligned Parallel Compressed Sparse Row matrix storage format called MAP-CSR to further accelerate local memory accesses. Running on a custom graph-based accelerator, we are able to achieve consistent speedup over MKL and provide insights on the scalability of our implementation.

5.1 Motivation for SpGEMM kernel acceleration

Multiplication of two sparse matrices (i.e., a SpGEMM kernel) is commonly found in many emerging workloads. Some examples of popular algorithms that need to process sparse matrices include:

- Scientific computations: algebraic multigrid solvers (AMG) [11], volumetric mesh analysis [130] and linear-scaling electronic structure computations [21];

- Graph-based computations: triangle counting [46], path planning [138], community detection [54], breadth-first-search [26], recommendation systems [136], graph neural networks [14], label propagation, network packet routing [177], graph centrality measures, and graph contractions [5].

SpGEMM plays a pivotal role in applications for controlling epidemics. It serves as an essential kernel in calculating centrality measures for airports [5, 165], and offers crucial guidance for directing vaccine distribution to key cities [82]. The world exposure graph, plotted in Figure 5.1, is recursively generated using the airport network dataset. We incorporate this dataset (with a sparsity of 99.63%) in the analysis of our SpGEMM kernel implementation on the custom accelerator.

Modern trends in Big Data have witnessed an increase in data sparsity, along with an increase in data set size. In 2021, Facebook claimed they had 2.89 billion monthly active users, with studies suggesting an average of 338 friends per user [163]. The resulting adjacency matrix of the Facebook user graph would approach 99.99% sparsity. Graph analytics of such highly sparse datasets push the limits of the current computing infrastructure and expose innate problems exhibited by traditional architectures. Sparse graph workloads are dominated by highly irregular and uncoalesced memory access patterns.
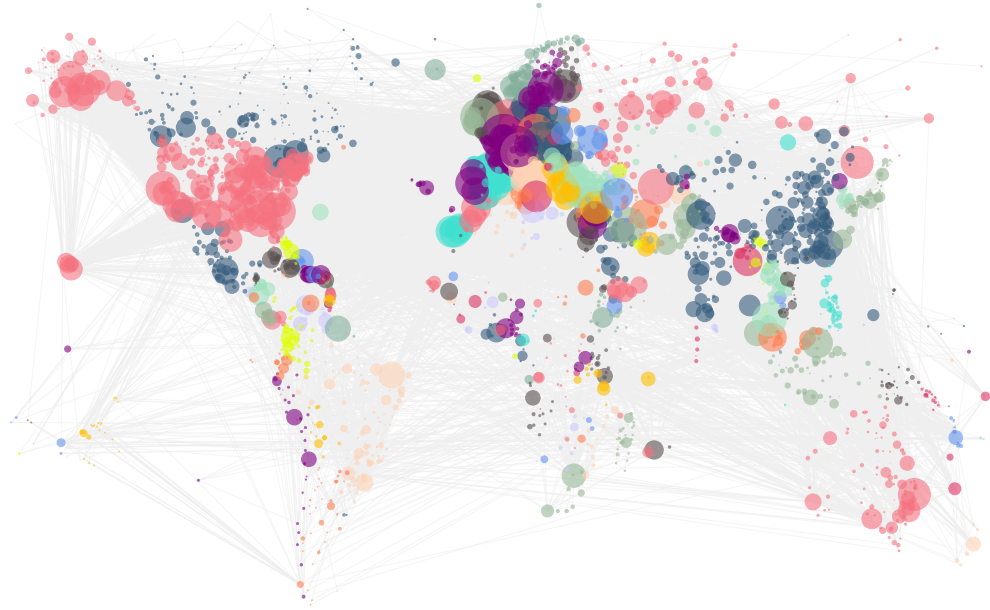
Figure 5.1: World exposure graph centrality, as generated using an SpGEMM kernel (with nodes as cities, node size proportional to Maximal Frontier Betweenness Centrality (MFBC), edges as air travel corridors, and colors representing countries)

Current multi-core CPU architectures, given their limited number of compute units (i.e., cores), fail to capitalize on the fine-grained parallelism present in these workloads. Single Instruction, Multiple Data (SIMD) style GPU architectures struggle to evenly distribute tasks among their threads, leading to under-utilization of hardware. In this work, we present an SpGEMM algorithmm called Sparse Matrix Atomic Scratchpad Hashing (SMASH), tailored to a custom multi-threaded accelerator. This accelerator provides a novel MIMD-style architecture based on simple in-order cores. *SMASH*, a state-of-the-art SpGEMM kernel implementation, provides $1.6\times$ average speedup over MKL with synthetic datasets, a $1.29\times$ average speedup over real world datasets and a $1.04\times$ average speedup over real world datasets as compared to an A100 GPU.

To summarize, the key contributions of this part of the thesis work include:

1. We characterize the challenges faced while developing efficient SpGEMM kernels. To this end, we perform an analysis of various sparse matrix multiplication methods and evaluate the advantages and limitations of each..

2. We present a new sparse matrix storage format called Memory Aligned Parallel - Compressed Sparse Row (MAP-CSR), which allows us to compute each row of the sparse matrix in par-

allel. MAP-CSR improves the efficiency of memory accesses, ensuring memory-aligned storage of each row. Our MAP-CSR implementation is able to improve the performance of SpGEMM by $1.58\times$.

3. Finally, we present Sparse Matrix Atomic Scratchpad Hashing (SMASH), an efficient SpGEMM kernel implementation that leverages distributed memory on a custom accelerator. We provide three different versions of SMASH, with iterative improvements, each capitalizing on a different feature of the underlying architecture.

5.2 Background on SpGEMM

The SpGEMM kernel operation generally consists of two distinct phases, each with its own computational requirements and challenges:

1. **The Multiplication Phase**: In this initial phase, the algorithm performs element-wise multiplication between corresponding non-zero elements of the sparse matrices involved. Given the sparse nature of the matrices, the algorithm needs to identify matching elements efficiently. This often involves complex data structures like compressed sparse row (CSR) or compressed sparse column (CSC) to store only the non-zero elements along with their indices. The computational complexity in this phase is primarily determined by the number of non-zero elements in the matrices.

2. **The Accumulation Phase**: Following the multiplication of elements, this phase focuses on summing up the products to generate the final sparse matrix. This involves aggregating values that are multiplied with the same index, essentially condensing them into a single entry in the resulting matrix. The challenge here is to perform this aggregation in an efficient manner, especially when the product matrix has fewer zeros, i.e., is less sparse than the input matrices. Optimizations often target reducing memory access latency and improving data locality in this phase.

Variations in the implementations of these two phases give rise to various SpGEMM algorithms. There are four methods to compute the first multiplication phase, as shown in Figure 5.3 and Figure 5.2. Each method exhibits different memory access patterns and provides varying degrees of parallelism.
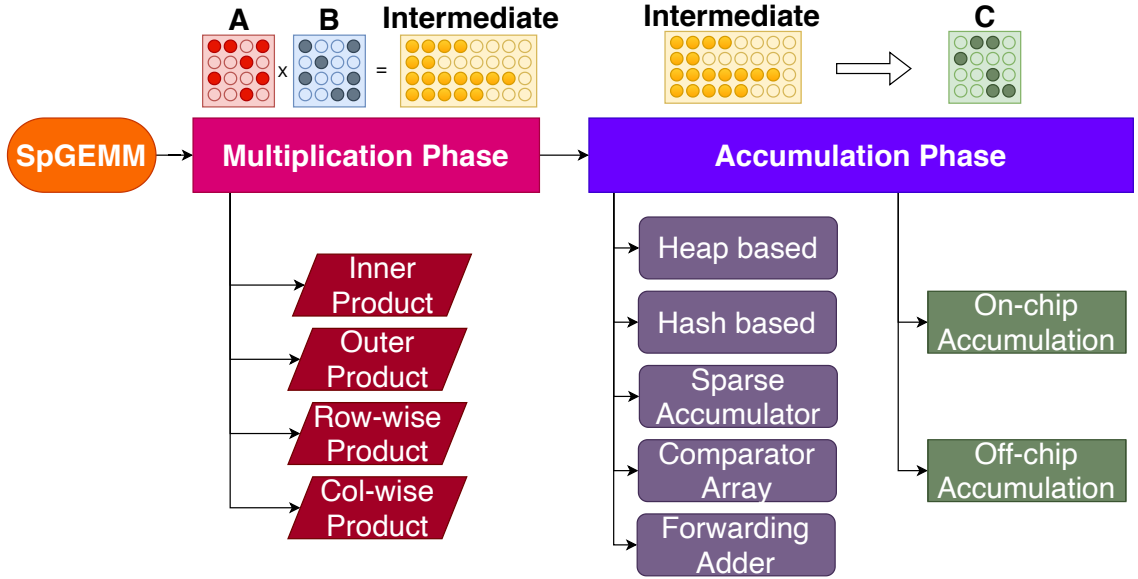
Figure 5.2: Methods of implementing the two distinct phases of SpGEMM kernel

While the inner product multiplication computes output matrix elements directly, its performance is crippled by poor input reuse. On the other hand, the outer product multiplication suffers from poor output locality arising from the endless batches of partial product matrices generated [212]. In our work, we incorporate row-wise multiplication, owing to the massive parallelism exposed by this method. Row-wise multiplication also does not suffer from the memory bloat problem when dealing with a large number of intermediate partial products [10].

The next phase, called the accumulation phase, can be distinguished based on the underlying data structures. Examples of accumulation techniques include heap-based [8], hash-based [132], sparse accumulator (SPA) based [67], comparator array based [212], and Forwarding Adder Network (FAN) based [147] to name a few. Depending on the memory hierarchy used for accumulation, this phase can be further classified into on-chip and off-chip accumulation.

This work presents SMASH, a scalable sparse matrix multiplication kernel based on the row-wise multiplication method. SMASH incorporates on-chip, hash-based accumulation to lower redundant memory accesses. We implement three different versions of this kernel on a custom graph accelerator.
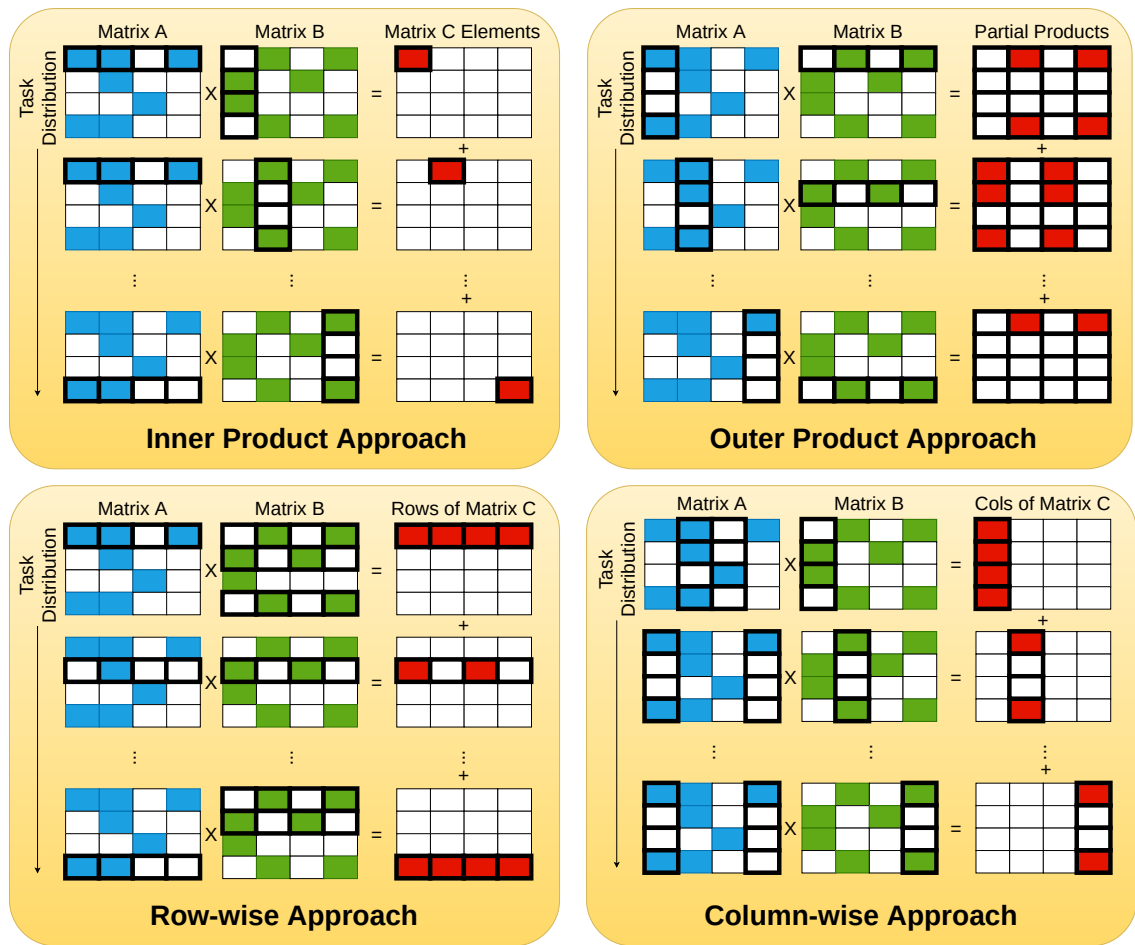
Figure 5.3: Matrix Multiplication Methods

5.3   MAP - CSR Storage Format

The traditional method of storing sparse matrices is CSR [162] (see Figure 5.4), which is memory efficient as it only stores $n + nnz$ elements instead of n^2 (where n is the dimension of a square matrix and nnz is the total number of non-zeros). But what it gains in memory efficiency, it lacks in exposing parallelism. For example, while writing to a CSR matrix, the rows are required to be written sequentually. If the data needs to be written in parallel, using the CSR format requires knowledge of the number of non-zeros in all rows in advance to allocate memory preemptively.
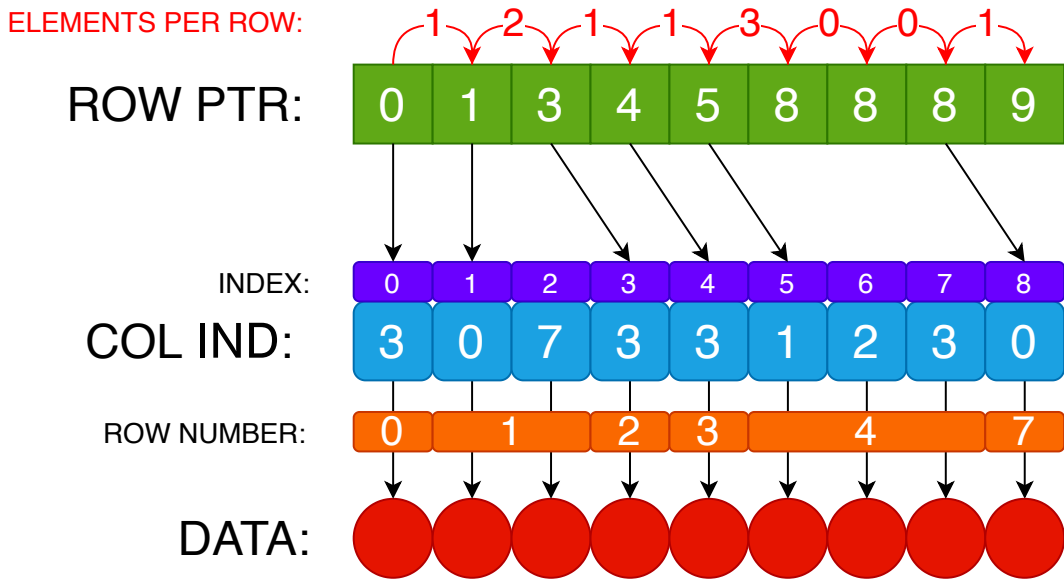
Figure 5.4:   Conventional CSR Format

The Conventional CSR format allows only sequential write operations, which poses a considerable challenge to implement SpGEMM kernels that scale on multi-node systems. It also introduces many synchronization operations, leading to performance degradation. We introduce a novel matrix storage format called MAP-CSR [155], that is designed to scale well on large-scale distributed systems.

5.3.1   MAP-CSR Implementation

Instead of the traditional 3-array storage format used conventional CSR (refer to Figure 5.4), MAP-CSR utilizes a 5-array storage (see Figure 5.5) as follows:

1. Elements per row array: Stores the number of non-zero elements present in each row.

2. Row pointer array: Points to the start of each row (stores the offset to the start of each row in the column pointer and the data array).

3. Replicator array: Similar to the row pointer array, but points to the replica of rows in the column pointer array and the data array.

4. Column pointer array: Stores the column indices of elements in each row.

5. Data Array: Stores the value of each element.
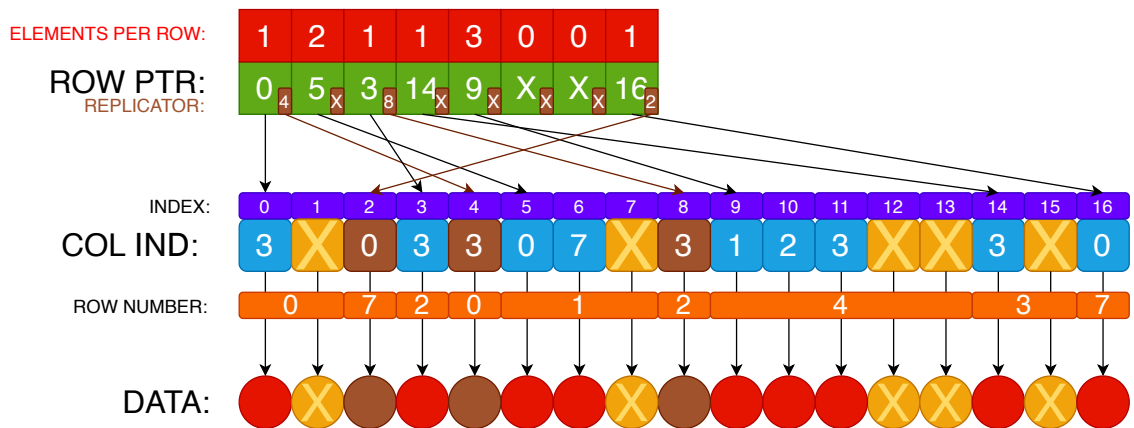
Figure 5.5: MAP-CSR Format

Using a 5-array storage format allows us to write the rows of the matrix in a random order, as compared to the sequential order imposed by Conventional CSR. In addition to storing rows in a random order, our approach also allows padding rows with zeros. This enables us to store rows in specific memory banks in main memory. Data accesses to different memory banks can have different latencies depending on which core is making the request. The ability to select a memory bank for storing specific rows allows for the optimization of memory access latency, particularly for rows that are accessed frequently.

A similar approach of storing sparse matrices was incorporated by Buluc et al. [25], where they stored rows in random order. With MAP-CSR, we add another feature called the *Replicator array*. This array, as the name suggests, allows rows to be replicated multiple times in the column pointer and data array. For example, row 7 is replicated in Figure 5.5. The row pointer array points to offset 16, the location where one copy of the row is located, and the replicator array points to offset 2, where a second copy of the row is present. There might even exist more than two copies of each row, in which case, the replicator array for each core points to their respective offsets with low

latency. To compare memory requirements of the MAP-CSR format with the memory consumed by the Conventional CSR format, we can compute the replication ratio $\Re$ as follows:

$$\Re \approx \frac{nnz + nnz' + nz_{pad}}{nnz} \tag{5.1}$$

where nnz represents number of non-zeros, nnz' represents the number of non-zeros from the replicated rows, nz_{pad} denotes the number of zeros used for padding and $\Re$ is the replication ratio, where $1 \leq \Re < \infty$.

We benchmark our SpGEMM implementation using the MAP-CSR storage format and compare it to the performance of a vanilla CSR (traditional CSR) storage format. Figure 5.6 provides information on the replication ratio (Equation 5.1) for each dataset, as well as the speedup obtained by using MAP-CSR as compared to the CSR storage format. A higher ratio denotes a larger memory footprint, hence we aim to lower the replication ratio. On average, we obtain a $1.582\times$ speedup by utilizing MAP-CSR, as compared to using a CSR storage format, with an average replication ratio value of 3.169.
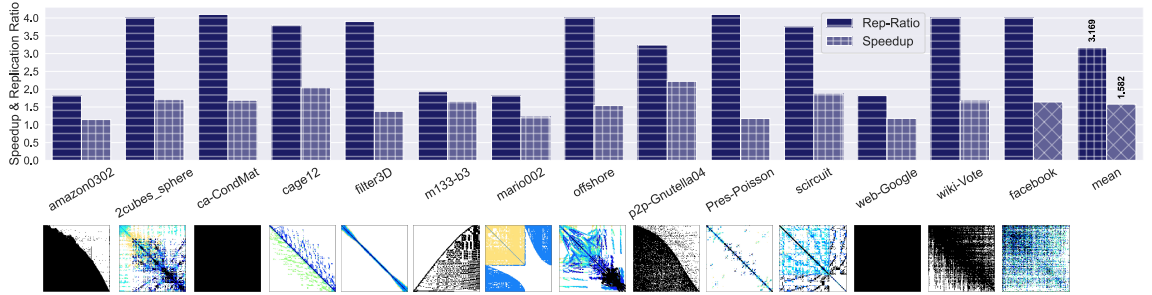
Figure 5.6:   Replication Ratio (lower is better) and Speedup (higher is better) of SMASH using MAP-CSR v/s CSR storage format on real world datasets.

5.3.2   MAP-CSR Advantages

MAP-CSR offers many advantages over conventional CSR, namely:

1. Allows rows to be stored in a random order.

2. Allows zero-padding to be aligned on memory banks.

3. Allows for rows to be replicated for faster reads.

4. Allows for rows to be prefetched, as they are isolated in banks.

60

5.3.3   MAP-CSR Limitations

While MAP-CSR offers major benefits, it also has certain shortcomings, which we address here. While replication of rows provides faster memory read transactions (data can now be fetched from memory with relatively lower latency), this affects the writing mechanisms. Writing to a replicated row requires writing to the original copy of the row (as pointed to by the row pointer) and requires the replicated copies to be invalidated.

Conversion of the Conventional CSR to MAP-CSR is associated with both compute and memory overheads. Memory overhead, as discussed before (refer to Equation 5.1), poses a $\Re$ times increase in the memory footprint. The computational overhead of converting to MAP-CSR is associated with the replication of rows. Replication of rows requires re-computing the indices of the row pointer in MAP-CSR format, which is a compute-intensive process. Despite the necessary overhead associated with MAP-CSR, we were able to obtain an average speedup of $1.58\times$ over conventional CSR, achieving an average replication ratio of 3.17 for the SpGEMM workload.

5.4   SMASH Kernel

One of the key design choices for our SpGEMM kernel implementation was to select one of the four general matrix-multiplication approaches (shown in Figure 5.3). The inner product approach faced issues due to the cost of index-matching and low temporal reuse [142]. The outer-product approach generated a large number of intermediate partial products, demanding high on-chip memory requirements. Neither of these choices provides any benefit when multiplying extremely sparse matrices.

Our novel implementation of the SpGEMM kernel is based on a row-wise product method called SMASH [155]. Our method exploits high data reuse behavior [147] and minimizes the number of input matrix reads, while still maintaining low on-chip memory usage. SMASH incorporates on-chip memory to store intermediate results and leverages the atomic instructions to accumulate these partial products.

In this dissertation , we present a set of successive improvements, resulting in three versions of SMASH [155] (overview of SMASH architecture shown in Figure 5.7). In each version we identify the remaining bottlenecks, and then optimize our algorithm to mitigate them in the next version. Each SMASH implementation targets a specific performance bottleneck on the custom accelerator architecture.
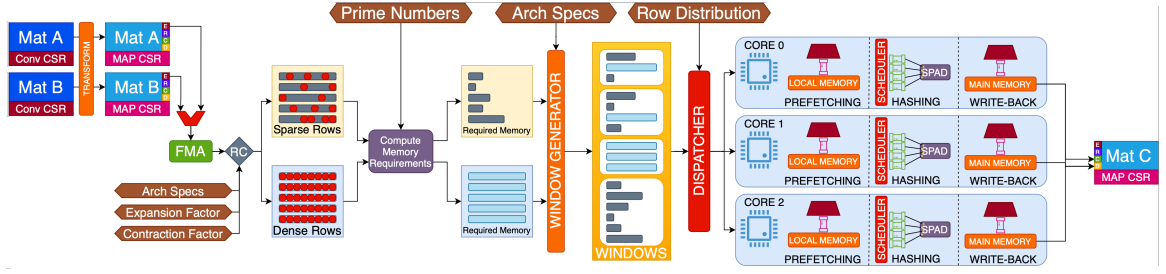
Figure 5.7: The SMASH architecture.

The following subsections describe our implementation of SMASH, along with three different optimizations. Similar to Gustavson's two-phase matrix multiplication algorithm [73], our SMASH implementation is characterized by two phases:

1. Memory computation phase

    (a) Matrix Read

    (b) Compute Memory Requirements

    (c) Window Generation

2. Product computation phase

    (a) Prefetching

    (b) Hashing

    (c) Write-back

5.4.1 Memory computation phase

Analogous to Gustavson's two-phase matrix multiplication approach [73], the first phase of SMASH determines the memory required for the output matrix C, as well as the on-chip memory requirements of the intermediate products. For evaluation purposes, we do not include the time consumed in the memory computation phase while considering speedup over other architectures. Evaluation methodologies are further discussed in Section 5.5. This phase can be further decomposed into three tasks.

1. **Matrix Read** - Our SMASH SpGEMM implementation starts off with reading input matrices A and B, both of which are presumed to be in a conventional CSR format.

2. **Compute Memory Requirements** - After reading the input matrix arrays in CSR format, we compute the required size of memory required to store the output matrix by counting the total FMA operations per row. We compute the maximum number of non-zeros for every row of output matrix C using Gustavson's two-step algorithm [73]. In this dissertation , we refer to this term as Floating-Point Multiply-Add (FMA). The computation of FMAs per row has a computational complexity of $\mathcal{O}(n)$, where n is the size of the input matrix.

3. **Window Generation** - Once the memory requirements of each row are computed, this phase then classifies each row of the output matrix C as either dense or sparse. We then group multiple rows together into a single window that can be dispatched as a task to computing core. This process of classifying and grouping rows into windows is characterized by two parameters:

   (a) Contraction Factor (CF): Decides if a row of output matrix C can be classified as dense or sparse. If $\frac{FMA}{CF} > threshold$, then the row is classified as dense, else it is classified as sparse. The $threshold$ value is a function of $scratchpad$ size and matrix density.

   (b) Expansion Factor (EF): This is used to determine the memory requirements of sparse rows, where the memory requirements are equal to the higher prime number closest to the value $FMA \times EF$.

A dense and a sparse row is evaluated differently in SMASH during the hashing phase. A sparse row will be allocated less memory than the max size of the row, as a dense row will follow a $1 : 1$ mapping and will be allocated memory equal to the max size of the row. Once the classification of rows is complete, this phase groups multiple rows together into a single window, such that the intermediate partial products can fit on the on-chip memory (i.e., scratchpad). At this phase of window generation, the input matrices are converted from conventional CSR format to the new MAP-CSR format. The MAP-CSR storage format allows for each window to consist of rows in a non-sequential order, permitting greater flexibility for this phase to generate windows. An evenly spread mix of sparse and dense rows are packed together and shipped to the next phase for computation. Every individual compute core processes its own window independently, regardless of the status of other windows. This allows us to assign windows to compute cores in a random order and oversubscribe windows.

5.4.2 Product Computation Phase

5.4.2.1 Prefetching

Each window generated in the previous phase is scheduled on a computing core for generating intermediate products. The prefetching phase preemptively copies the input matrix rows that are required by each compute core to their respective local memory bank. This phase of Prefetching is only possible due to the "replicator" property of MAP-CSR, which allows rows to be duplicated multiple times across each computing core.

5.4.2.2 Hashing

This phase involves the multiplication of input matrix elements, required to compute the intermediate partial product. After the partial product is computed, it needs to be stored and merged. Merging partial products is a memory-intensive process requiring scanning through arrays to match indices. In addition, on multi-threaded architectures, this class of operations needs to be synchronized to avoid data races and ensure atomicity. Among the multiple solutions available to store and merge data, we opt for hashtables. SMASH utilizes row-partitioned hashtables to store and merge partial products. The use of hashtables avoids the use of expensive index matching, while allowing us to merge partial products on the fly.

We utilize hashtables to store intermediate partial products (on the on-chip memory). In the hashing phase, a global hashtable is created in the Scratchpad (SPAD) (the on-chip memory). A single row is allocated to one thread of each compute core in a round-robin fashion. Each element of the row from the first matrix is multiplied with an entire corresponding row of the second matrix (Equation 5.2 and 5.3, where C is the output matrix, A and B are input matrices, and N is the size of the matrix). This leads to the creation of partial products. These partial products are hashed into the SPAD using prime-modulo hashing.

$$C[i, :] = \sum_{k=0}^{N} A[i, k] * B[k, :] \tag{5.2}$$

$$u \otimes v = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & u_1 v_3 \\ u_2 v_1 & u_2 v_2 & u_2 v_3 \\ u_3 v_1 & u_3 v_2 & u_3 v_3 \\ u_4 v_1 & u_4 v_2 & u_4 v_3 \end{bmatrix} \tag{5.3}$$

In prime-modulo hashing, we hash the intermediate products on the SPAD by indexing each to the closest highest prime number, as computed in the Window Generation phase. The use of prime numbers allows us to reduce the number of hash collisions. A prime-modulo hash of intermediate products can result in three outcomes:

- **Hash Insert**: This routine is executed when a hashed index finds an empty location on SPAD. The column index value and data value of the intermediate product are stored on the SPAD at the hashed index location.

- **Hash Update**: This routine is executed when the hashed index does not find an empty location on SPAD, but the column index value of the current intermediate product matches with the one present on SPAD. In this case, the SPAD data value is updated with the sum of new data value and existing data value. This routine is also called the merging of partial products.

- **Hash Collide**: This routine indicates that the hashed index did not find an empty location on SPAD, nor did the column index match the existing column index value on SPAD. In this case, the algorithm probes for a new location using quadratic probing to find the next available empty location on SPAD for hash insertion.

The pseudo-code for the entire hashing phase is shown in Algorithm 1.

5.4.2.3   Write-back

The write-back phase moves the partial products from the hashtable to their final output matrix, stored in DRAM in the MAP-CSR format. The use of the MAP-CSR storage format allows us to asynchronously move rows of the output matrix C in non-sequential order from the SPAD to their final DRAM memory location.

The SMASH implementation discussed so far is considered as our "base" implementation. We iteratively add three more optimizations on top of this base implementation, addressing key performance bottlenecks observed during each implementation.

5.4.3   SMASH Version 1: Atomic Hashing

A row-wise product method multiples each element of the first input matrix with an entire row of the second input matrix, generating a row of partial products of the output matrix. These partial products are then merged to form the output matrix elements using a hashtable. This is one

---

Algorithm 1: SMASH HASHING

```
        // READ PHASE
 1  while Till you reach end of window do
        // Atomically distribute work to each thread
 2      token ← Each thread will receive one unique token
 3      if token_id % 2 = 0 then
 4          row_begin ← A_col_ptr_copy_1[token_id/2]
 5      else
 6          row_begin ← A_col_ptr_copy_2[token_id/2]
 7      end
 8      row_end ← A_col_ptr[token_id/2 + 1]
 9      for i ← Iterate from row_begin to row_end do
10          if Check if we are within our assigned window then
11              col_begin ← B_row_ptr[token_id/2]
12              col_end ← B_row_ptr[token_id/2 + 1]
13              if token_id % 2 = 0 then
                    // Hash EVEN Section
14              else
                    // Hash ODD Section
15              end
16          end
17      end
18  end
19  A_col_ptr_copy_1 and A_col_ptr_copy_2 will now reflect new positions
```

---

of the disadvantages of using a row-wise product method. The intermediate results (i.e., partial products) need to be stored and merged into the output matrix atomically. The base version of SMASH only allowed a single compute core to work on each row of output matrix C, avoiding data races. This leads to a lower degree of parallelism in each window, as the maximum number of compute cores concurrently working on any window depends on the number of rows in that window. We overcome this obstacle with our first version $V1$ of the SMASH kernel by using atomic hashing. We make use of atomic compare and exchange instructions and atomic fetch and add instructions, enabling us to use multiple cores simultaneously to produce a single output row of matrix C. Optimizing with atomic instructions leads to a $2.48\times$ speedup over the base SMASH implementation for synthetic datasets.

5.4.4 SMASH Version 2: Tokenization

SpGEMM workloads, when working with extremely sparse matrices that possess a highly irregular non-zero distribution, experience load imbalance on multi-core architectures. Although our implementation is not immune to the effects of such irregular sparsity patterns, we aim to reduce the performance impacts of load imbalance with an on-the-fly row scheduler that is based on the

---

Algorithm 2: SMASH HASHING Even and Odd Section

---

1 **for** $k \leftarrow$ *Iterate from col_begin to col_end* **do**
  // Multiply element from mat_A with that from mat_B
  and store its tag and value
2 | $tag \leftarrow X$ coordinate from mat_A element and Y coordinate from mat_B
  element
  // Hash the Tag
3 | $tag \leftarrow tag \% prime\_modulo$
4 | **if** $SPAD\_tag[tag] = EMPTY$ **then**
5 | | $SPAD\_tag[tag] \leftarrow tag$ // Store Tag on scratchpad
6 | | $SPAD\_val[tag] \leftarrow value$ // Store Value on scratchpad
7 | **else**
8 | | **if** $SPAD\_tag[tag] = tag$ **then**
9 | | | $SPAD\_val[tag]+ = value$ // Accumulate Value
10 | | **else**
   | | | // Probe for empty space on Scratchpad
11 | | **end**
12 | **end**
13 **end**
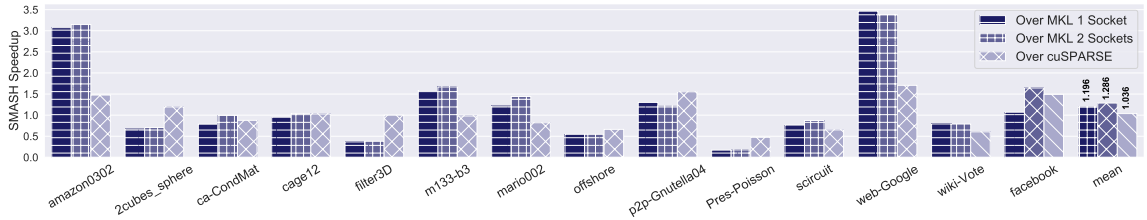
Figure 5.8: Speedup of SMASH over MKL using 1 CPU, 2 CPUs, and over cuSPARSE using an A100 GPU

classic producer-consumer model.

We tackle this issue by adding a dynamic work scheduler layer into our hashing phase. Instead of statically assigning rows to threads in a round-robin fashion, we adopt a Producer-Consumer for model row allocation. The dynamic row allocation works as follows:

1. Generate two tokens for every row present in the window.

2. Each compute core polls for a single token. Thus, every row is allocated 2 compute cores.

3. The 2 compute cores start hashing the row. The first core starts from the beginning of the row and hashes the first half of the row (i.e., the even section). The second thread applies the same steps over the second half of the row (i.e., the odd section).

4. Partial products from both threads are hashed into a common hashtable, stored in the SPAD memory.

5. When all of the tokens have been polled, the window execution is completed.

Despite the overhead of polling tokens, tokenization produces a $1.5\times$ speedup over static allocation, as it achieves a near-perfect distribution of workload across threads. More details of the performance benefits are presented in Section 5.5.

5.4.5 SMASH Version 3: Pipelining

Previous versions describe how *atomic hashing* exposes more parallelism and how *tokenization* balances workload across compute core. Next, we describe an optimization to increase resource utilization, where we adopt pipelining at the phase level. We divide the local memory and SPAD into 2 equal parts. In the first iteration, the first window is processed by the prefetching phase. In the next iteration, that hashing phase processes the previously prefetched window and the prefetching phase works on the next upcoming window. Once the hashing phase is completed, the write-back phase starts moving data out of the SPAD. The hashing phase starts working on the next window, preemptively loaded by the prefetching phase (Figure 5.9). As compared to our previous
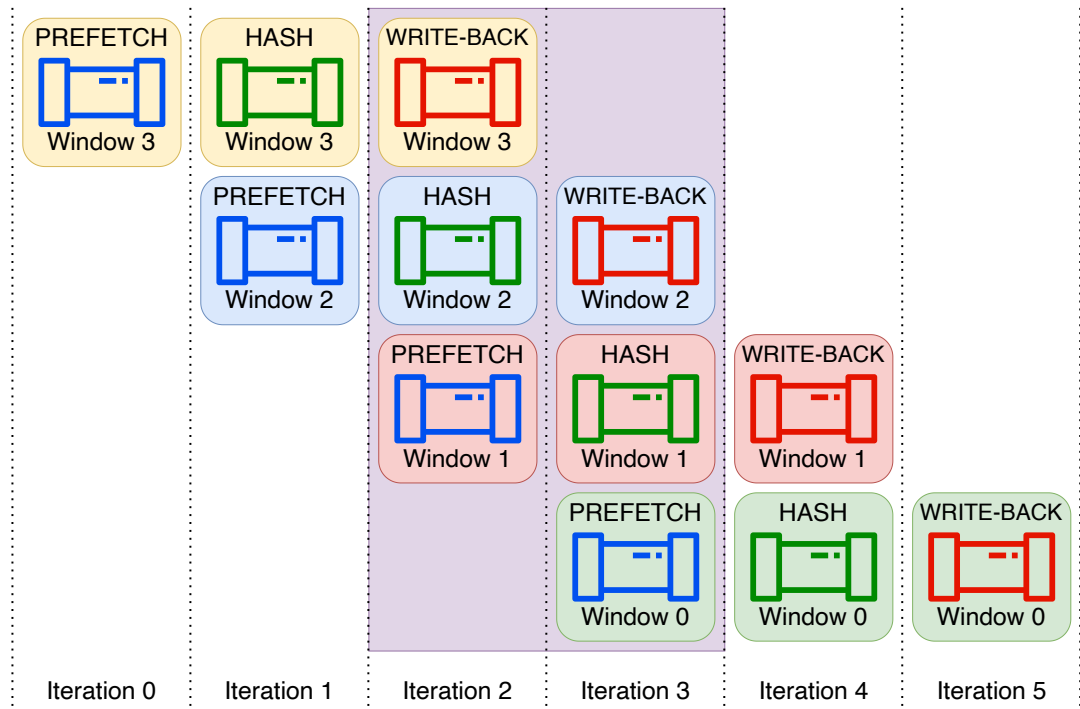
Figure 5.9: SMASH Pipeline Stages

SMASH versions, where at each stage only one phase is active, this phase enables all three phases

to be active simultaneously. This comes at the cost of increased resources required in terms of local memory and SPAD memory. Despite the increased resource utilization, with SMASH $V3$ pipelining, we were able to obtain a $1.3\times$ speedup as compared to SMASH $V2$, for synthetic datasets.

5.5 SMASH Evaluation

We designed SMASH, a SpGEMM kernel implementation, to expose the performance improvements provided by the custom accelerator architecture. We compare its performance to Intel's MKL implementation on a dual socket server (with Intel Xeon E5-2630), as well as against NVIDIA's A100 GPU.

We evaluate the performance of our SMASH SpGEMM kernel implementation on synthetic, as well as real-world, datasets. For synthetic datasets, we chose the RMAT, as it produces datasets possessing a non-zero power-law distribution [33, 168], making it harder to find patterns in the non-zero values.
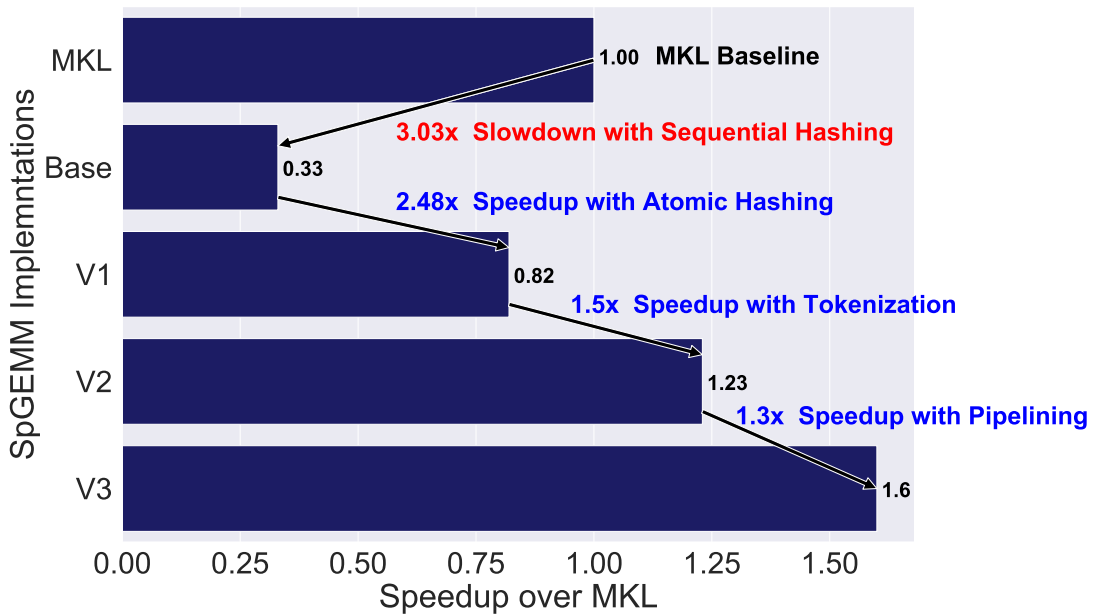
Figure 5.10: Speedup over MKL, as compared to different versions of SMASH exploiting various architectural features.

For real-world datasets, we experimented with datasets from the Stanford Network Analysis Platform (SNAP). For all our experiments, we compare the performance of a single CPU core

to a single compute core, a single Intel CPU with 8 cores, to 8 custom accelerator cores, a dual-socket server with 16 compute cores, and an A100 GPU to a system with 64 compute cores. For our first experiment, we compare the speedup obtained for our SMASH implementations on 16 compute cores, and compare against Intel MKL's performance on a dual-socket server, as seen in Figure 5.10. Our base implementation of SMASH ends up with $3.03\times$ slowdown, but after iterative optimizations, exploiting various architectural features, we end up with an average speedup of $1.6\times$ over MKL for synthetic datasets.

Next, we focus on the "Tokenization" optimization. Tokenization of the hashing phase led to better workload balance between threads. With tokenization, almost all threads have near-perfect utilization, leading to an average $1.5\times$ speedup over the non-optimized version. We next analyzed the performance improvements provided by pipelining. Ideally, if a workload is divided into 3 stages of a pipeline, the highest speedup achievable is 3. For this case to hold true, the work across each stage of the pipeline would need to be completely balanced. In our case, the prefetcher consumes 12.1% of cycles, the hashing phase consumes 64.8% of cycles, while the writeback phase consumes 18.8% of overall cycles, as shown in Figure 5.11. Despite this imbalance of cycles taken by each of the phases, we obtained a $1.3\times$ speedup over the non-pipelined version of SMASH for the synthetic datasets. We also performed scaling experiments by measuring the performance improvements
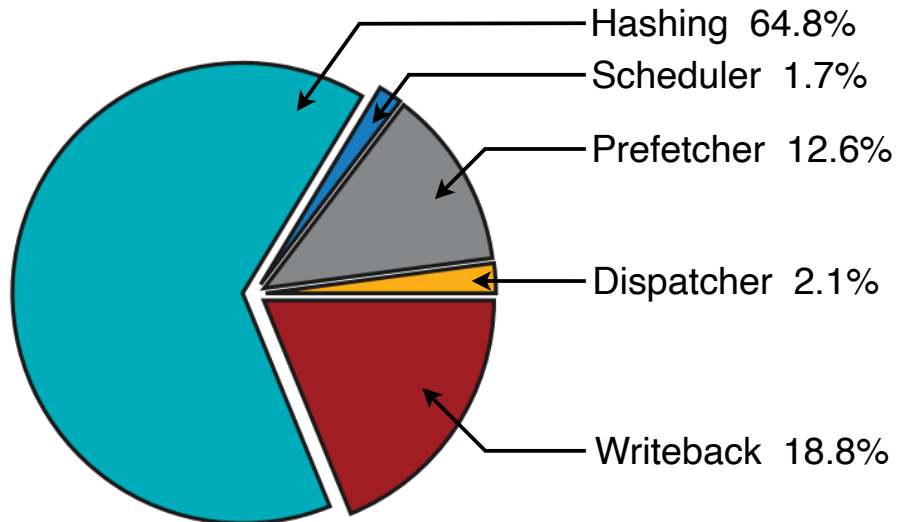
Figure 5.11: Cycle consumption breakdown of SMASH phases

as a function of both the number of cores, as well as the matrix density (this experiment utilizes

the synthetic dataset). Figure 5.12 plots the density of the input matrix on the X-axis and the number of cores utilized on the Y-axis. Each number in this heat map is representative of the speedup acquired by SMASH V3 over MKL with the same number of cores. This plot indicates that SMASH outperforms the MKL implementation for sparse matrices at higher core counts. Finally,

Core Count	8	16	32	64	128	256	512	1024
16	1.4	1.2	0.96	0.66	0.6	0.38	0.41	0.43
15	1.3	1.1	0.9	0.62	0.56	0.4	0.39	0.43
14	1.2	1.1	0.88	0.6	0.55	0.4	0.36	0.47
13	1.2	1	0.82	0.56	0.49	0.3	0.33	0.4
12	1.2	0.97	0.8	0.54	0.52	0.33	0.31	0.35
11	1.1	0.91	0.76	0.51	0.47	0.37	0.35	0.36
10	1.1	0.88	0.74	0.5	0.48	0.34	0.32	0.33
9	0.95	0.81	0.66	0.45	0.42	0.24	0.27	0.31
8	1.5	1.3	1	0.69	0.68	0.42	0.52	0.49
7	1.5	1.2	1	0.69	0.65	0.4	0.38	0.5
6	1.4	1.2	1	0.68	0.39	0.36	0.37	0.4
5	1.6	1.2	0.95	0.63	0.41	0.33	0.38	0.43
4	0.89	0.79	0.66	0.45	0.33	0.33	0.35	0.37
3	0.87	0.8	0.67	0.45	0.36	0.34	0.37	0.37
2	0.84	0.8	0.67	0.45	0.34	0.34	0.35	0.35
1	0.82	0.78	0.67	0.43	0.31	0.32	0.34	0.33

Average number of non-zeros per row

Figure 5.12: Speedup over MKL, which varies as a function of matrix density and core count.

we compare the performance of real-world datasets from the SNAP library against an MKL single socket, an MKL dual-socket, and a cuSPARSE single GPU A100. We obtain an average speedup of $1.20\times$ over a single socket MKL kernel, $1.29\times$ speedup over a dual-socket MKL kernel, and $1.04\times$ speedup over the cuSPARSE kernel.

5.6 SMASH Summary

SpGEMM workloads are memory-intensive workloads that possess highly irregular memory access patterns. In this work, we presented SMASH, a scalable SpGEMM kernel implementation targeting a custom graph accelerator. Our 3 iterative optimizations exploit the architectural features of the graph accelerator and provide $2.48\times$, $1.5\times$ and $1.3\times$ speedups, respectively. Our atomic hashing optimization, tokenization, and pipelining of SMASH kernels provided us with an average of $1.20\times$, $1.29\times$, and $1.04\times$ speedup over MKL single socket, MKL dual-socket, and GPU A100 hardware, respectively.

Chapter 6

Hardware Acceleration of GNNs

Graph Neural Networks (GNNs) are emerging as a formidable tool for processing non-euclidean data across various domains, including bioinformatics, financial networks, energy networks, telecommunication and social network analysis. Despite their effectiveness, their adoption has not been pervasive because of scalability challenges associated with large-scale graph datasets, particularly when leveraging message passing, posing significant computational bottlenecks. This class of large-scale workloads exhibits irregular sparsity patterns, resulting in unbalanced utilization of computational resources.

6.1 Key Bottlenecks in Accelerating GNNs

Deep Neural Networks have proven to be powerful models for solving problems that rely on data with an underlying Euclidean or grid-like structure [206], such as computer vision, natural language processing, and audio vision. In contrast, Graph Neural Networks (GNNs) have emerged as powerful frameworks for handling non-Euclidean data (e.g., social networks on the scale of billions [163]), achieving impressive performance across various domains such as social science, chemistry, and bioinformatics [196]. However, the computational complexity of GNNs, especially when working with ultra-sparse, large-scale graph datasets, poses challenges due to architectural limitations of traditional hardware (i.e., CPUs / GPUs) [145]. Moreover, GNNs predominantly adopt a recursive neighborhood aggregation methodology, in which each node aggregates the feature vectors of its neighboring nodes to derive its own updated feature vector. The scalability of message passing in GNNs, when applied to large graph structures, poses a significant bottleneck, especially as the size of the graphs surpasses the capacity of on-chip memory hierarchies in today's CPUs and

GPUs [158]. This leads to redundant and time-consuming memory transactions to fetch data from the main memory [14].

We identify the following three key bottlenecks for GNN workloads:

1. Diverse Data Dependence Patterns: GNN workloads feature multiplication and accumulation operations, each demonstrating unique data dependency patterns.

2. Poor Hardware Resource Utilization: Compute units suffer from data starvation and load imbalance due to irregular sparsity patterns exhibited by large input graphs.

3. Memory bloat: The matrix multiplication methods generate a large number of intermediate partial products that require efficient merging to avoid redundant accesses to higher level memory. We further elaborate on each of these three critical bottlenecks identified for GNN workloads.

Diverse Data Dependence Patterns: The process of neighborhood aggregation in graphs can be split into a multiplication stage, followed by a merge/reduction stage. The multiplication stage creates partial products by multiplying the adjacency matrix of the input graph with the feature matrix. The reduction stage accumulates (i.e., merges) the partial products to update the node feature vectors. The multiplication stage's operands depend on data stored in the high-bandwidth memory (HBM) [142]. In contrast, the reduction stage's operands depend on data located within the on-chip memory. Utilizing a singular computational resource for both multiplication and accumulation operations proves suboptimal, as mapping multiplication operations on computing resources tends to compromise the efficiency of mapping the accumulation operations (due to varying data dependency patterns).

Poor Hardware Resource Utilization: The multiplication and accumulation stages are characterized by distinct architectural implications. The multiplication stage typically stalls due to data starvation (stalls accessing the input graph and feature matrix elements), whereas the accumulation stage suffers from uneven partial product distribution due to the sparsity patterns. Prior accelerators [153, 210] have adopted look-ahead buffers for prefetching data, aiming to prevent compute stalls caused by data starvation. While these solutions reduce compute stalls, aggressive prefetching leads to cache pollution as redundant data resides in the cache [154]. These issues can be addressed using two strategies, each catering to their respective problems. (a) *Multiplication mapping:* Implementing a *tiled row-wise product approach* to partition the computation into distinct tasks, which are then dynamically allocated to NeuraCore (computing elements), depending
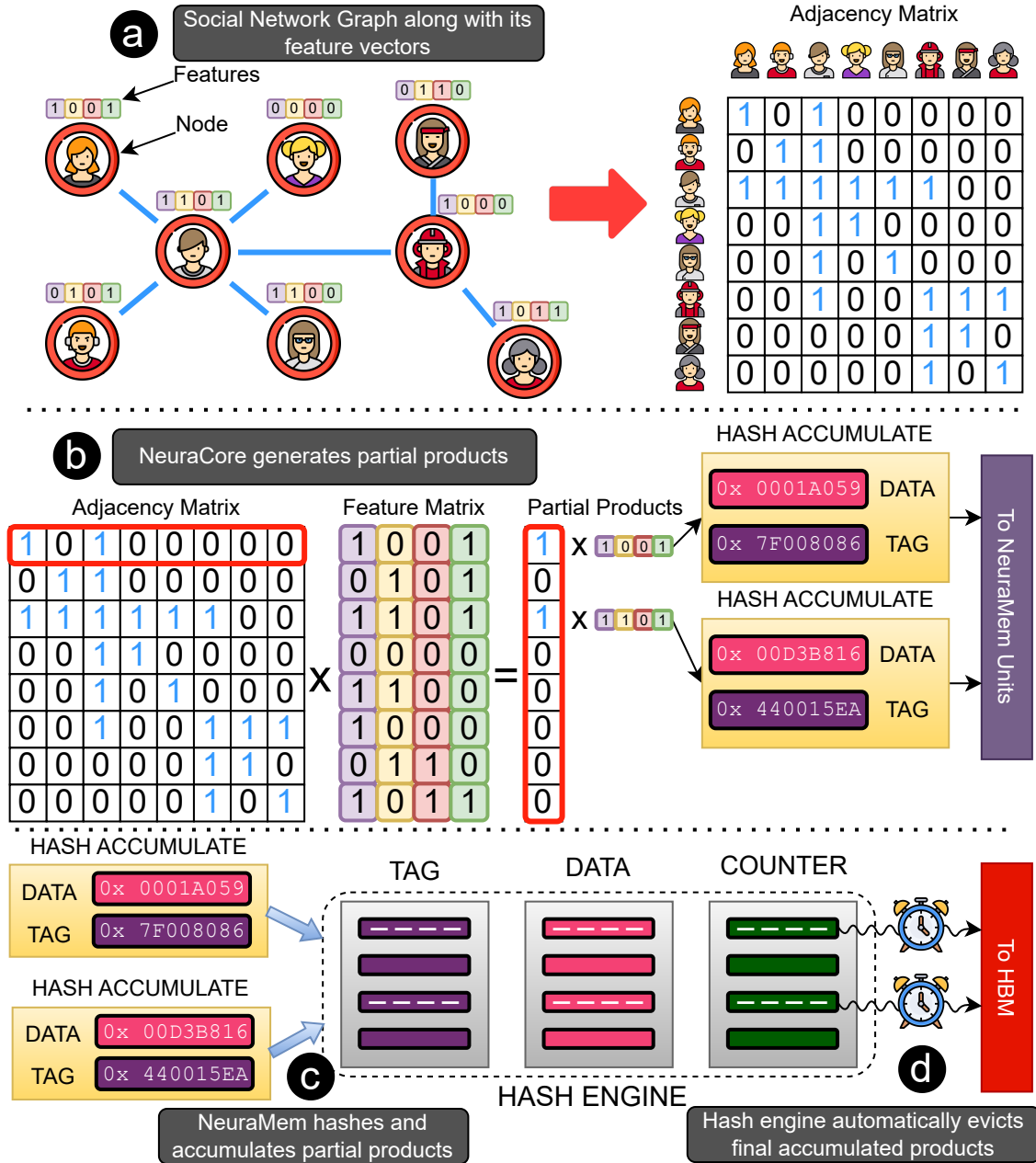
Figure 6.1: NeuraChip overview, illustrating aggregation phase of graph convolution on a social network graph (a). NeuraCore generates partial products based on input graph (b). NeuraMem accumulates partial products (c) and writes back to HBM (d).
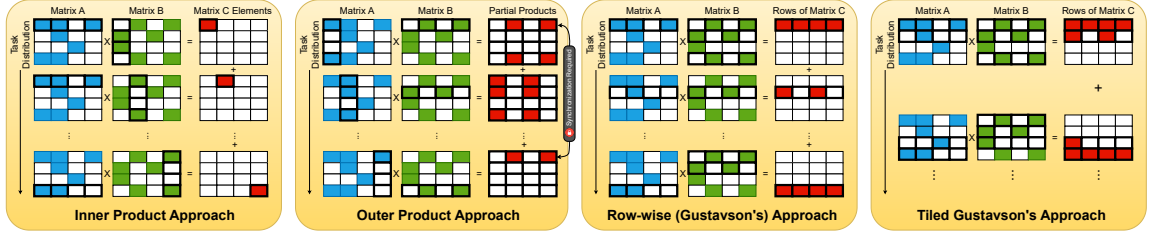
Figure 6.2: Various approaches to matrix multiplication, each showcasing different degrees of data reuse for input and output matrices.

on its utilization. The row-wise product method, also known as Gustavson's algorithm, is a popular choice among recent accelerators such as Gamma [210], MatRaptor [169], and SPADA [117] as this approach has shown high-efficiency when targeting sparse matrix computations in the aggregation stages of GNNs. Developing dedicated components for multiplication enables mapping these operations to NeuraCore, independent of the accumulation stage, thus leveraging the locality of the input data. (b) *Accumulation Mapping:* Using a dynamic reseed hash-based mapping agnostic to sparsity patterns. This allows even distribution of the partial products among the NeuraMem (on-chip memory) accumulation units.

Memory Bloat: Incorporating the row-wise product approach enhances input data locality but creates a large number of partial products. Table 6.1 presents memory bloat for SpGEMM workload across various sparse graph datasets. We define bloat percent as shown in Equation 6.1

$$\text{Bloat Percent} = \frac{pp_{\text{interim}} - nnz_{\text{output}}}{nnz_{\text{output}}} * 100 \tag{6.1}$$

wherein pp_{interim} denotes the count of intermediate partial products and nnz_{output} signifies the count of non-zero elements in the resultant product matrix. Although tiling the computation partially addresses this issue, it does not fully resolve it. Prior solutions such as Gamma [210] have relied on large explicitly managed cache systems similar to FiberCache [210], which consumes up to 72% of the total chip's area. The memory bloat issue can be addressed using a rolling eviction strategy, which automatically evicts a partial product from the on-chip memory once all contributing partial products have been fully accumulated. We enable a strategy using an eviction counter integrated with the on-chip memory hashtables.

The work here develops NeuraChip, an innovative GNN spatial accelerator featuring a decoupled computation pipeline. Decoupling multiplication and accumulation operations into dedicated components, we optimize data reuse through strategic mapping. We explore an adaptive hash-based

Table 6.1: SpGEMM bloat analysis across hyper-sparse graph datasets

Dataset	Node Count	Edge Count	Sparsity (%)	Bloat Percent
2cubes_sphere	101492	1647264	99.9840	205.87
ca-CondMat	23133	186936	99.9651	75.23
cit-Patents	3774768	16518948	99.9999	19.32
email-Enron	36692	367662	99.9727	68.90
filter3D	106437	2707179	99.9761	326.34
mario002	389874	2101242	99.9986	99.43
p2p-Gnutella31	62586	147892	99.9962	10.21
poisson3Da	13514	352762	99.8068	297.92
scircuit	170998	958936	99.9967	66.13
web-Google	916428	5105039	99.9994	104.27
amazon0312	400727	3200440	99.9980	97.21
cage12	130228	2032536	99.9880	127.23
cop20k_A	121192	2624331	99.9821	327.07
facebook	4039	60050	99.1519	2872.80
m133-b3	200200	800800	99.9980	26.93
offshore	259789	4242673	99.9937	205.45
patents_main	240547	560943	99.9990	14.18
roadNet-CA	1971281	5533214	99.9999	35.75
webbase-1M	1000005	3105536	99.9997	36.02
wiki-Vote	8297	103689	99.8494	148.09

compute mapping. Our approach introduces a flexible, dynamic reseeding hash-based compute mapping (DRHM) tailored for GNN workloads. DRHM benefits from the constant lookup times characteristic of hash functions, while also mapping tasks evenly across all computing resources by generating a new seed at predetermined intervals of computation. We also explore how to use rolling evictions in order to address memory bloat. We explore on-chip hash tables to manage partial products, effectively reducing memory congestion caused by their generation.

6.2 Fundamentals of GNN Workloads

Graph neural networks (GNNs) are capable of extracting important features, such as structural motifs (i.e., arrangements of nodes, edges, and metadata) [92], learning not only the individual characteristics of each element (i.e., a node in the graph), but also the interconnections (i.e., the interrelationships between nodes) between elements [219]. GNNs use convolution operations to extract various features from the graph [102]. The methodology employed is called *neighborhood aggregation*, where the final feature vector for each vertex is computed by iteratively aggregating and transforming the input feature vectors of adjacent vertices [219]. This process includes two steps, which are called the *aggregation and combination stages*. This process is carried out iteratively, and after k iterations through these stages, the resultant feature vector of the target vertex signifies the distinct structural data of the vertex's k-hop vicinity [193].

For instance, a Graph Convolutional Network (GCN) is one such GNN model. Equation 6.2 below computes the forward propagation for a single layer in a GCN.

$$X^{(l+1)} = \sigma(AX^{(l)}W^{(l)}) \tag{6.2}$$

where A represents the adjacency matrix of the graph, where each row lists the interconnections of a vertex to all other vertices in the graph. $X^{(l)}$ refers to the input feature vectors of every vertex in the lth layer of matrix X. W contains the GNN's model parameters, which are obtained through model training. $\sigma()$ represents the non-linear activation function, for instance, ReLU (Rectified Linear Unit).

6.3   Architectural Implications of GNN Workloads

Aggregation Stage: The aggregation stage in GNN workloads is critical for capturing the structural information of graphs. It involves gathering and summarizing information from a node's neighbors, which can be a challenging task given the irregular data structures common in graph-based data. This is typically computed with sparse matrix multiplication kernels. Given the high level of sparsity in input graphs, typically above 99%, this stage is characterized by random access patterns in memory, which presents a challenge for traditional architectures that are more suited for linear data access. Additionally, the skewed sparsity patterns often lead to workload imbalance on computing resources, which can impact performance efficiency.

Combination Stage: The combination stage in GNNs involves the integration of node features with neighborhood information. This process is computationally intensive and typically comprises dense matrix multiplications, nonlinear activations, and dimensionality reduction operations. Architecturally, this stage demands high memory bandwidth and efficient data reuse mechanisms to handle large matrices. It also necessitates a balance between compute utilization and memory access, as the combination of features from large graphs can lead to memory bottlenecks. While prior accelerators [117, 210, 212] often focus on sparse matrix multiplication tasks, they do not adequately address dense workload demands. Our NeuraChip accelerator model provides a more generalized solution, addressing the needs of both sparse graph computations and dense workloads. This approach positions NeuraChip as a versatile GNN accelerator, adept at handling both the aggregation and combination stages.

6.4   Sparse Matrix Multiplication: Algorithmic Overview

The Sparse General Matrix-Matrix Multiplication (SpGEMM) kernel execution is characterized by two main stages: the multiplication stage and the accumulation stage as visualized in Figure 6.3. The implementation variations in these stages lead to distinct SpGEMM algorithms. We describe the four approaches to execute the initial multiplication stage, as illustrated in Figure 6.2. These approaches vary in their memory access patterns and the level of parallelism they expose.

The inner product approach, incorporated in InnerSP [10] computes elements of the output matrix directly but is hindered by inefficient input reuse. Conversely, the outer product approach, utilized in OuterSPACE accelerator [142] is hampered by suboptimal output locality due to the creation of numerous batches of intermediate partial product matrices [212]. Our research adopts the

row-wise multiplication approach (i.e., Gustavson's algorithm), selected for the extensive paral-
lelism it provides. Notably, this approach efficiently avoids the memory bloat issue associated with
handling numerous intermediate partial products [212].

The subsequent stage, known as the accumulation stage, merges the generated interme-
diate partial products. Various accumulation methods include heap-based [8], hash-based [132],
sparse accumulator (SPA) based [67], comparator array based [212], and Forwarding Adder Net-
work (FAN) based [129, 147], among others (illustrated in Figure 6.3). This stage can also be
subdivided into on-chip and off-chip accumulation, based on the utilized memory hierarchy. Neu-
raChip merges partial products using on-chip accumulation to reduce redundant main memory data
fetches. For sparse matrices with skewed non-zero distributions, the on-chip accumulation stage can
result in uneven workload distribution, a factor that significantly impacts the overall performance
and efficiency of SpGEMM operations.
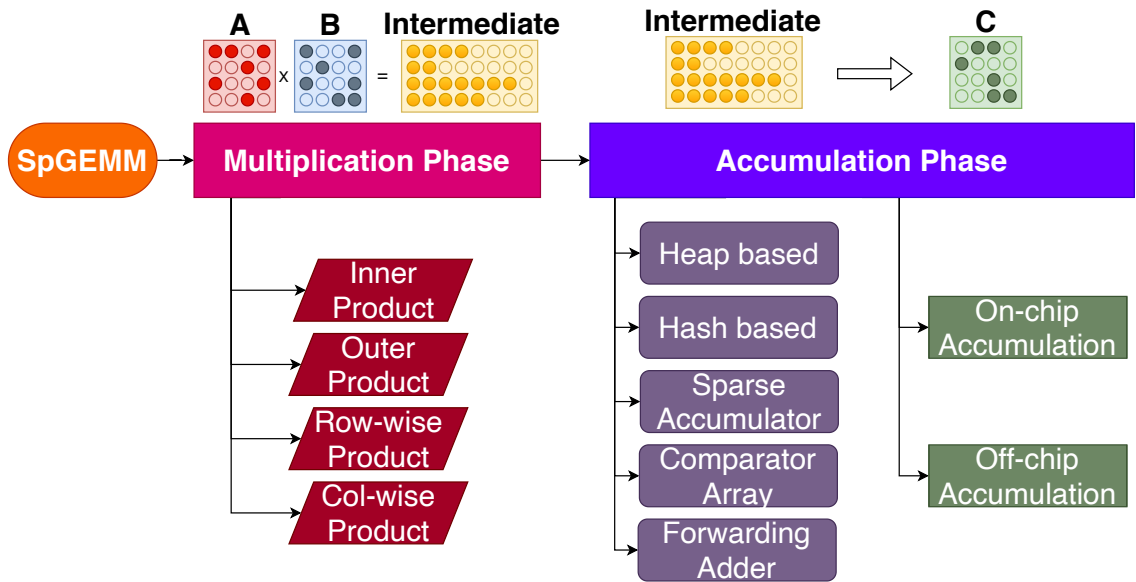
Figure 6.3: Various methods employed in multiplication and accumulation stages.

6.5 Mapping Algorithms: Design and Requirements

Mapping algorithms play a crucial role in efficiently handling computational tasks, partic-
ularly in scenarios involving sparse data structures such as those found in Graph Neural Networks
(GNNs). These algorithms are tasked with assigning tasks or data elements to computational nodes

or memory locations. The key requirements for effective mapping algorithms include:

Consistency: The algorithm must consistently map the same index to the same node. This ensures correctness in data processing.

Low Computational Overhead: The lookup process should be relatively fast, with minimal computational and memory overheads. This efficiency facilitates cost-effective index matching, streamlining partial product reduction.

Sparsity Agnostic: Regardless of skewed sparsity patterns, the mapping algorithm should remain impartial to these variations. This ensures uniform performance across different data sets [30].

Given these requirements, hash-based mapping emerges as a viable solution [39]. However, traditional hash-based methods such as Round Robin Hashing (or Ring Hashing) [174] and Prime Number Based Modular Hashing [19] have limitations [32]. Neither is fully insensitive to sparsity patterns; a specific set of indices might consistently map to the same node, leading to potential workload imbalance.

An alternative approach is random mapping, which ideally achieves sparsity-agnostic mapping by randomly distributing indices. However, to ensure consistency, this method requires maintaining a large lookup table, which is not practical due to memory constraints.

To address these challenges, we propose a novel approach: Dynamic Reseed Hash-Based Mapping (DRHM). This method is similar to prime modular hashing, but with a significant enhancement. After processing a predetermined set of computations, we reseed the hash function. The updated seed values are then stored in a compact lookup table. This dynamic reseeding ensures that the distribution of indices does not become predictable or skewed, effectively mimicking the sparsity-agnostic property of random hashing.

Dynamic Reseed Hash-Based Mapping strikes a balance between the ideal characteristics of random mapping and the practical limitations of traditional hash-based methods. By only storing seed values rather than the entire mapping of indices, it maintains a small memory footprint. Concurrently, it offers the sparsity-agnostic mapping necessary for handling diverse and skewed data sets efficiently. This method significantly enhances the performance of computational tasks, particularly in environments where data sparsity and distribution can vary widely.

6.6 NeuraChip Architecture

NeuraChip [156] is a decoupled spatial accelerator. Its two primary components include: i) the NeuraCore and ii) the NeuraMem. The NeuraCore is specifically tailored for multiplication

tasks, whereas the NeuraMem focuses on accumulating data on-chip [157]. They are arranged in an interleaved pattern and connected through a 2D torus network fabric, as shown in Figure 6.5. To facilitate efficient communication among these components, on-chip routers have been incorporated. NeuraCores and NeuraMems are organized into clusters known as tiles [156]. The accelerator includes a total of eight tiles, each linked to a single Double Data Rate (DDR) channel. Each tile features a memory controller responsible for interfacing with the DRAM banks.

Buffers play a critical role in the functionality of the four major components of our accelerator [158]. Both the NeuraCore and the NeuraMem are equipped with instruction buffers [157]. Additionally, the on-chip routers incorporate packet buffers, and the memory controllers are fitted with buffers for managing both reading and writing operations.

The incorporation of these on-chip buffers enhances the accelerator's flexibility, allowing it to adapt to diverse sparsity patterns. In scenarios where irregular graph structures could lead to network congestion, these on-chip buffers prove beneficial. They ensure that the components consistently have instructions to execute, thus avoiding potential delays or bottlenecks in processing.

6.6.1 Tiled Gustavson's Multiplication Algorithm

GNNs typically employ two primary layers (phases) in their architecture: the neighborhood aggregation phase, which gathers information from a node's neighbors in the graph, and the combination phase, where a node's representation is updated by integrating its own features with those aggregated from its neighbors [62]. This discussion focuses on the aggregation phase, which predominantly involves sparse matrix multiplications.

In this dissertation , we implement a modified version of Gustavson's matrix multiplication algorithm [157]. Gustavson's algorithm operates on a row-stationary approach, processing the output matrix one row at a time. Specifically, it traverses the adjacency matrix row by row, performing a linear combination of these rows as illustrated in Figure 6.4.

Gustavson's approach multiplies each element in a row of the adjacency matrix with all elements in the corresponding row in the feature matrix that has the same row index as the element's column index. Our adaptation enhances Gustavson's method by simultaneously processing multiple rows. We execute the multiplication of four rows at a time, aligning four elements from a column of the adjacency matrix with four elements from a row of the feature matrix. This is achieved using a specialized instruction, denoted as the `MMH4` instruction.

Our technique represents a fusion of Gustavson's algorithm and the outer-product method.
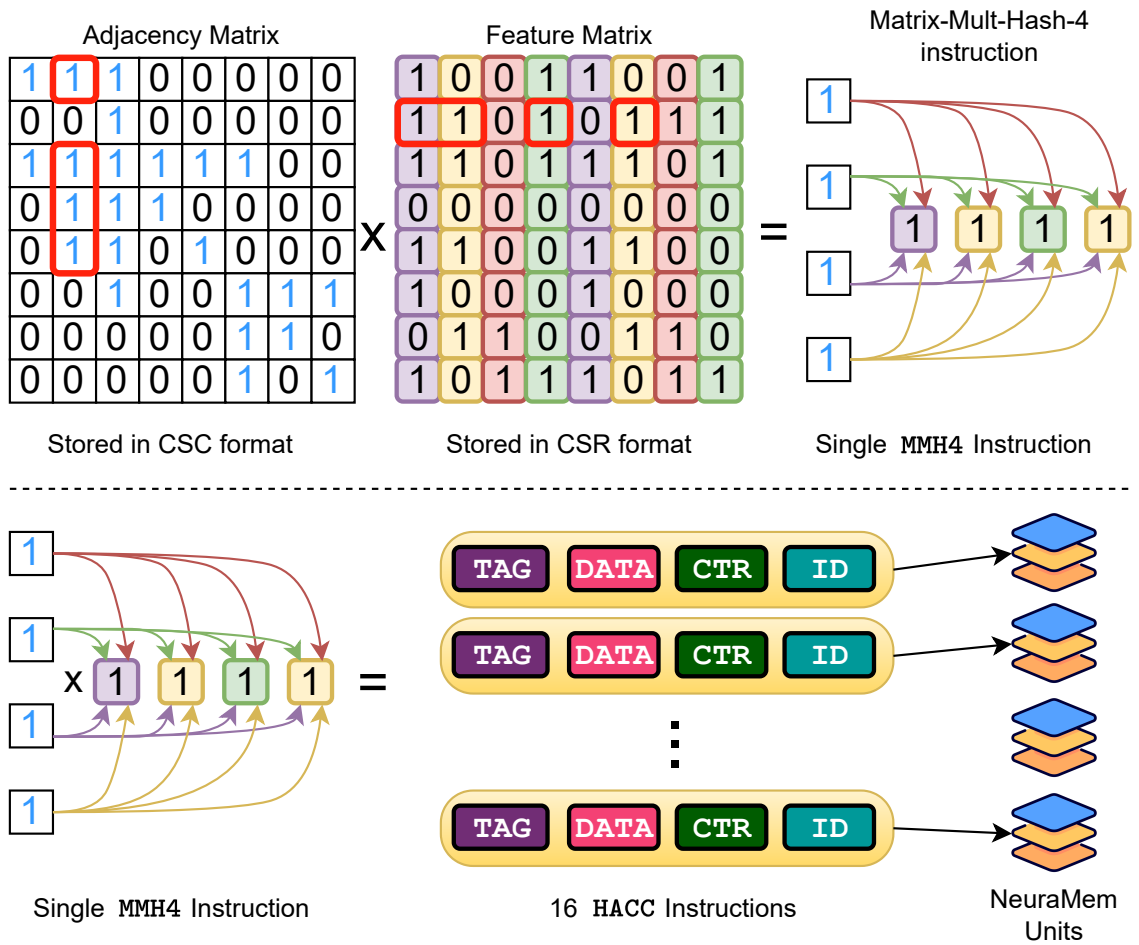
Figure 6.4: Implementation of tiled Gustavson's algorithm using NeuraCore for multiplication and NeuraMem for accumulation.
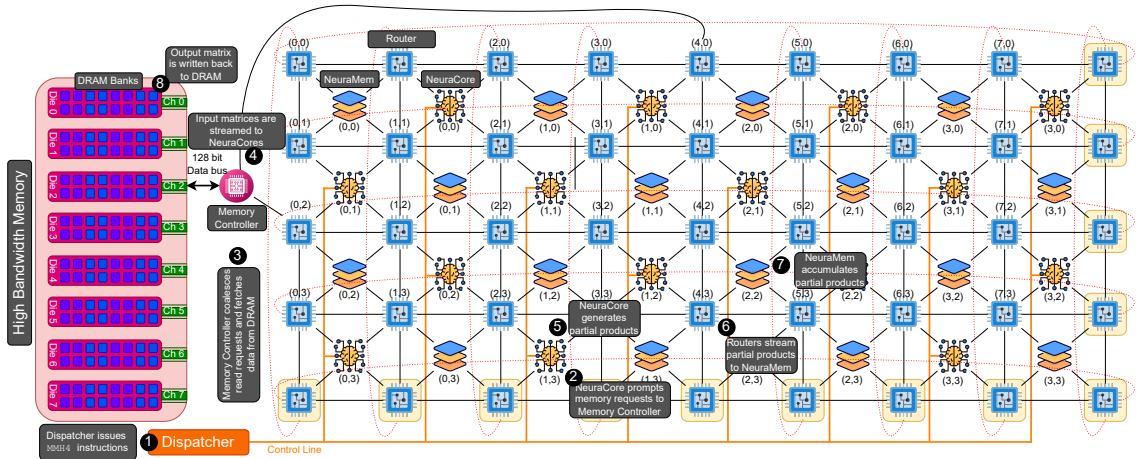
Figure 6.5: Overview of NeuraChip architecture with Tile 16 configuration (16 NeuraCores and NeuraMems per tile with a total of 8 tiles).

Unlike the outer-product approach which finalizes the multiplication of an entire column with a row before moving to the next, our strategy concurrently processes four rows by employing the Gustavson method. The selection of the number 'four' for simultaneous row processing results from design space exploration specific to the NeuraChip accelerator.

To implement this modified Gustavson's approach, the adjacency matrix is stored in a compressed sparse column (CSC) format, and the feature matrix is stored in a compressed sparse row (CSR) format. However, this approach presents two primary challenges:

Unavoidable Index Matching: Employing Gustavson's algorithm and compressed matrix storage formats such as CSR and CSC inherently leads to the necessity of index matching [169]. We address the index-matching overhead with a constant lookup hash function, facilitating the on-chip accumulation of partial products with a constant lookup time. The low overhead provided by our hash function is further optimized by adding a dedicated hash engine, as described in Section 6.6.4.

Memory Bloat Issue: The tiled Gustavson method can result in memory bloat, characterized by the generation of a large number of partial products [10]. To tackle this issue, we have implemented a rolling eviction mechanism. This system accumulates partial products as they are generated and promptly evicts them once the reduction is complete, with further details provided in Section 6.6.4.
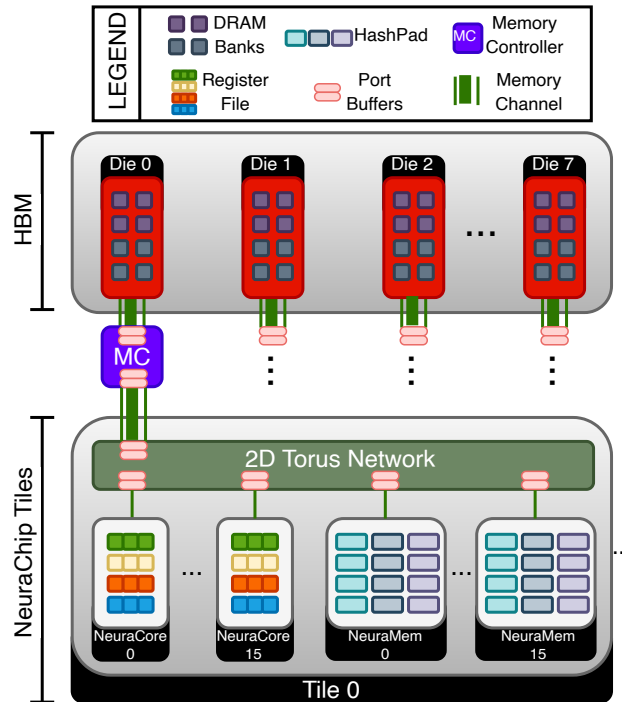
Figure 6.6: NeuraChip memory hierarchy

6.6.2 On-chip Dataflow

To illustrate the data flow within NeuraChip, we walk through an example of an SpGEMM kernel executed on the NeuraChip accelerator (see Figure 6.5 and 6.6). Step ❶ The process begins with the *Dispatcher* issuing **matrix_mult_hash_4** (MMH4) instructions to every *NeuraCore*. Step ❷ The *NeuraCores* trigger memory read requests that are routed to the memory controller. Step ❸ The *Memory Controller* coalesces requests for contiguous memory locations into a singular transaction and reorganizes memory transactions to enhance spatial locality. Step ❹ Input matrix data, fetched from DRAM, is streamed onto respective *NeuraCore* components. Step ❺ The *NeuraCores* compute the partial products, along with their corresponding rolling counters (further details in Section 6.6.3), subsequently generating the **hash_accumulate** (HACC) instructions. Step ❻ HACC instructions are streamed over on-chip routers into NeuraMem components, based on a hash-based mapping. Step ❼ The *NeuraMem* component employs another hash function to hash and accumulate these partial products onto their on-chip memory. Consecutive hashes of partial products with the same TAG are merged within NeuraMem, with each hash insertion decrementing the counter by 1. Step ❽ When the counter reaches zero; this triggers the eviction of the hashline,

and the resultant data is written back to the High Bandwidth Memory (HBM).

6.6.3 NeuraCore

The NeuraCore is the primary compute engine in our accelerator. It computes the multiplication operation and generates the partial products of matrix multiplication operations. It is a simple in-order core with support for matrix instructions. NeuraCore supports a special matrix instruction called `matrix_mult_hash_4` or simply `MMH4`.

where *opcode* represents the operation code, which specifies the MMH4 instruction to be executed by NeuraCore. $Base_{addr}$ denotes the base address used to offset the address of all other addresses involved in this instruction. A_data_{addr} refers to the memory address where the data of matrix A is located (matrix A is stored in CSC storage format). $B\_col\_ind_{addr}$ points to the memory address containing the column indices of matrix B (matrix B is stored in CSR storage format). B_data_{addr} indicates the memory address where data from matrix B is stored. $roll\_counter_{addr}$ denotes the memory address where the rolling eviction counter is located. The pseudocode for executing the `MMH4` instruction is shown in Algorithm 3. Each `MMH4` instruction has the capability to dispatch up to 16 `HACC` instructions (further elaborated in the NeuraMem section).

Algorithm 3: `MMH4` instruction execution

1: **for** $i = 0$ **to** 3 **do**
2:    **for** $j = 0$ **to** 3 **do**
3:       $TAG \leftarrow \text{Mem}[(Base_{addr} + B_{col\_ind\_addr} + j)]$
4:       $DATA \leftarrow \text{Mem}[(Base_{addr} + A_{data\_addr} + i)]$
5:         $\times \text{Mem}[(Base_{addr} + B_{data\_addr} + j)]$
6:       $CTR \leftarrow \text{Mem}[(Base_{addr} + roll\_counter + i * 4 + j)]$
7:       Dispatch `HACC`$(TAG, DATA, COUNTER)$
8:    **end for**
9: **end for**

The operational sequence within NeuraCore is shown in Figure 6.7, and can be broken down into the following steps: Step ❶: The operation starts with the dispatcher transmitting a `MMH4` instruction to NeuraCore, allocating the instruction to one of the available pipelines. Pipelines are allocated using a round-robin scheme. Step ❷: The `MMH4` instruction is decoded by the on-chip decoder. Step ❸: Following decoding, NeuraCore maps instruction variables to the register file, utilizing dynamic register allocation. Step ❹: Post register allocation, the NeuraCore's internal address generator constructs memory requests to fetch elements from the input matrices. Step ❺:
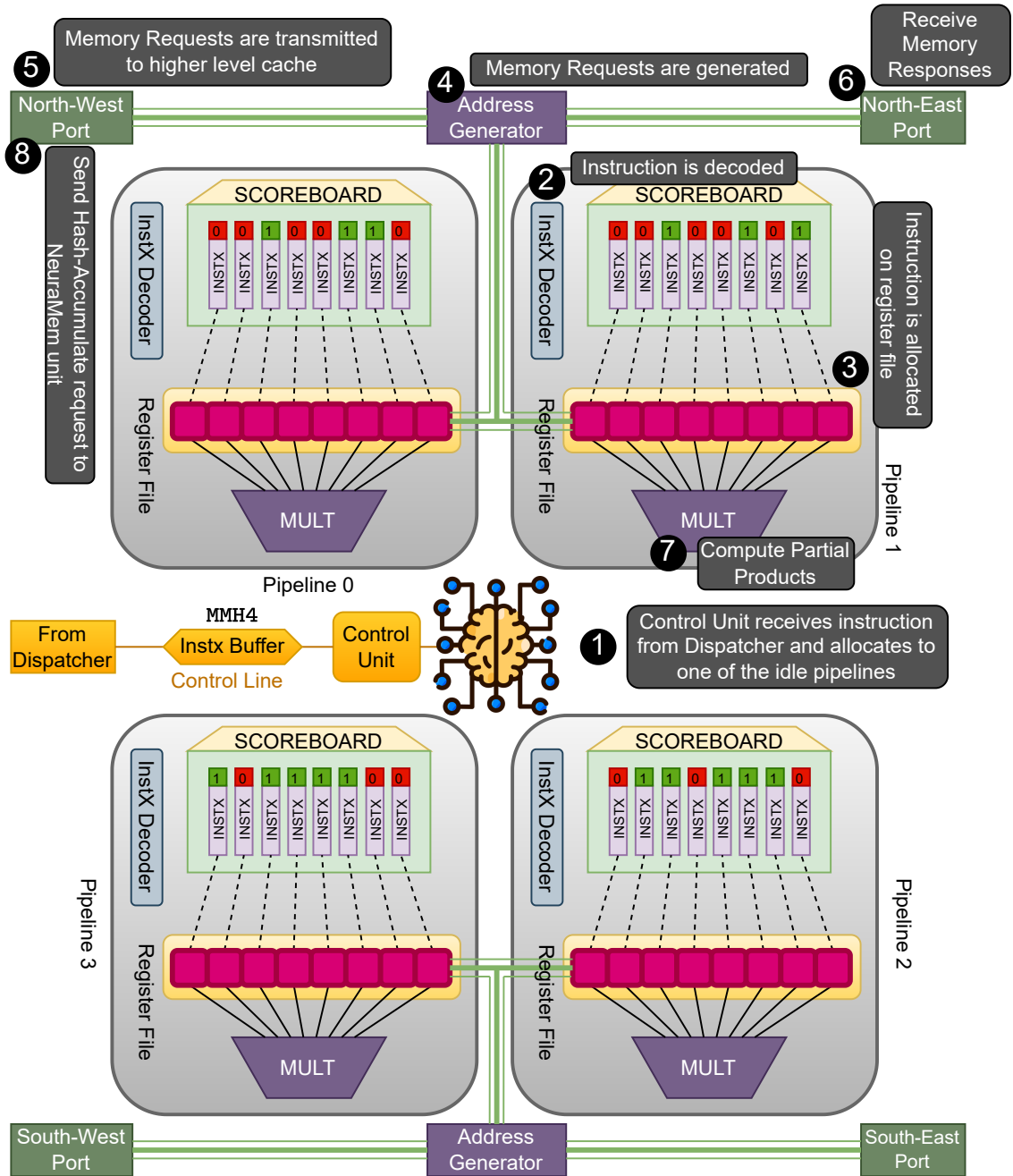
Figure 6.7: Block diagram showing NeuraCore's quad-pipeline layout.

An adaptive routing algorithm [6] selects the best port to dispatch the memory request, which is then forwarded to a higher-level cache. Step ❻: Upon completing the memory request, a response is received at one of the NeuraCore's four ports. This response is then routed toward its respective pipeline. Step ❼: As soon as all memory responses corresponding to a particular instruction are received, the instruction is deemed ready for execution by the scoreboard. Subsequently, the multiplication pipeline calculates the partial product and generates up to 16 HACC instructions. Step ❽: Lastly, the HACC instructions are relayed to NeuraMem units using the most suitable port, as determined by the on-chip hash-based mapping function.

6.6.4 NeuraMem

NeuraMem is a crucial component of the NeuraChip accelerator. While NeuraCore units generate partial products, NeuraMem units handle the on-chip accumulation of these partial products. The central component of NeuraMem units is the Hash-Engine. The layout of various components within NeuraMem is as shown in Figure 6.9.

HashPad: The Hash-Engine operates on what we refer to as "hash-lines" Figure 6.9. A hash-line comprises a single TAG, DATA, and COUNTER entry. The collective TAG array, DATA array, and COUNTER array, essentially the whole set of hash-lines, form what is known as the HashPad, as shown in Figure 6.9.

HACC instruction: NeuraMem supports a special instruction for partial product accumulation called hash_accumulate, or simply HACC instruction. The bit layout of HACC instruction is illustrated in Figure 6.10. Algorithm 4 presents a pseudocode of the HACC instruction, providing clearer insight into its functionality.

Hash-Engine workflow: Figure 6.11 shows a typical sequence of events during the execution of a HACC instruction by the Hash-Engine (illustrated using pseudocode in Algorithm 4).

mat_mult_hash_4 Instruction

opcode	Reg 0	Reg 1	Reg 2	Reg 3	Reg 4
MMH4	$Base_{addr}$	A_data_{addr}	$B\_col\_ind_{addr}$	B_data_{addr}	$roll\_counter_{addr}$
8 bits	32 bits	22 bits	22 bits	22 bits	22 bits
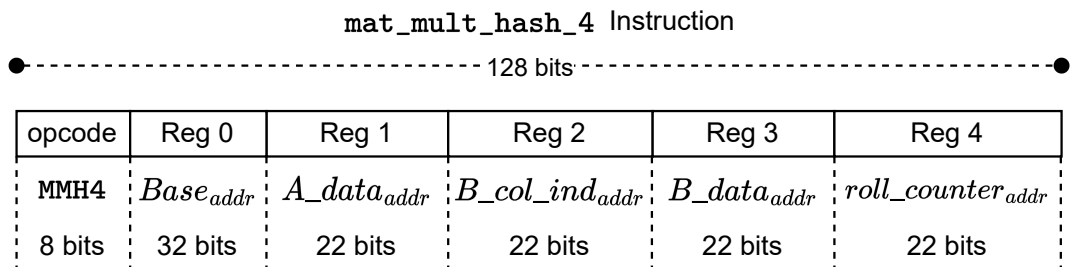
Figure 6.8: MMH4 instruction bit layout.
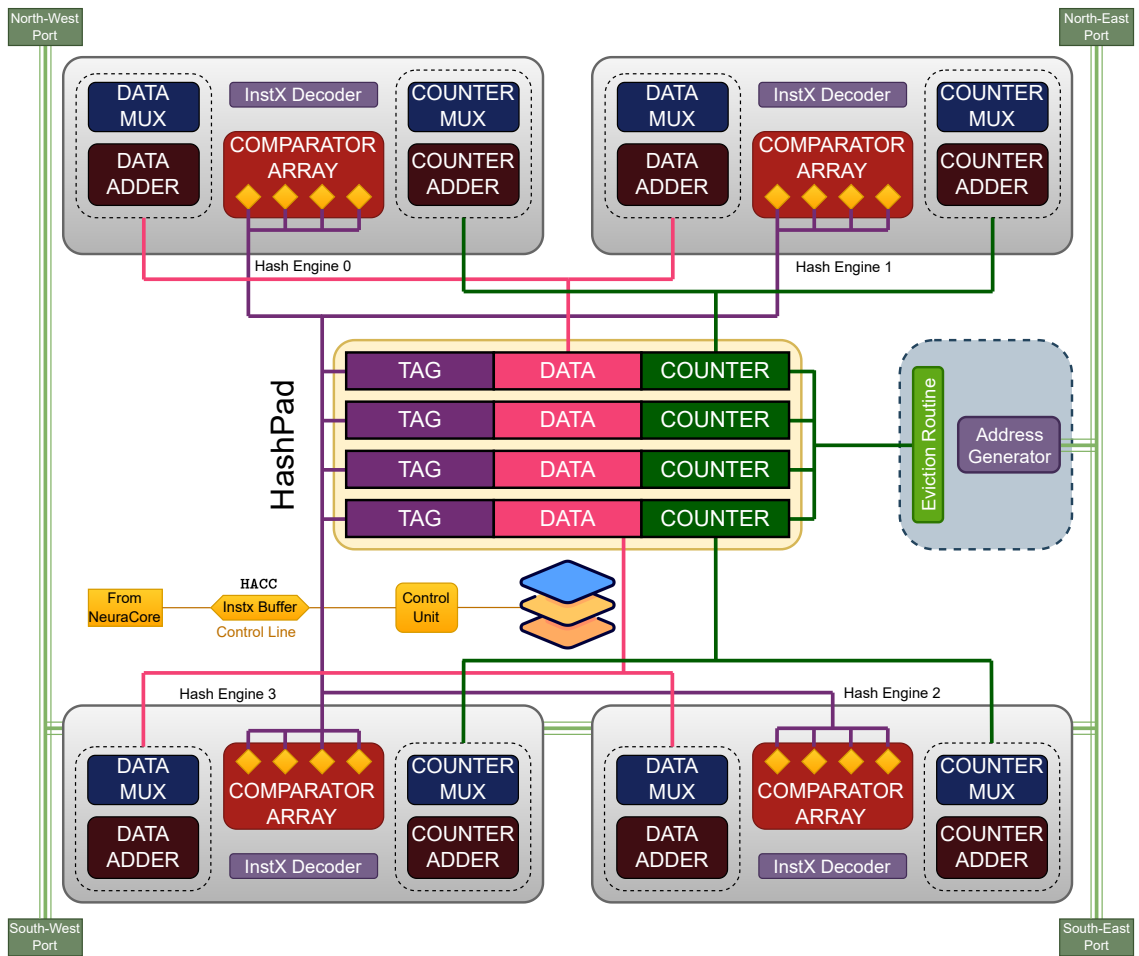
Figure 6.9: Block diagram showing NeuraMem's quad-hash-engine layout.

hash_accumulate Instruction

opcode	Reg 0	Reg 1	Reg 2	Reg 3	
HACC	TAG	$DATA$	$Rolling$ $Counter$	$NeuraMem$ ID	Unused
8 bits	32 bits	32 bits	32 bits	8 bits	16 bits
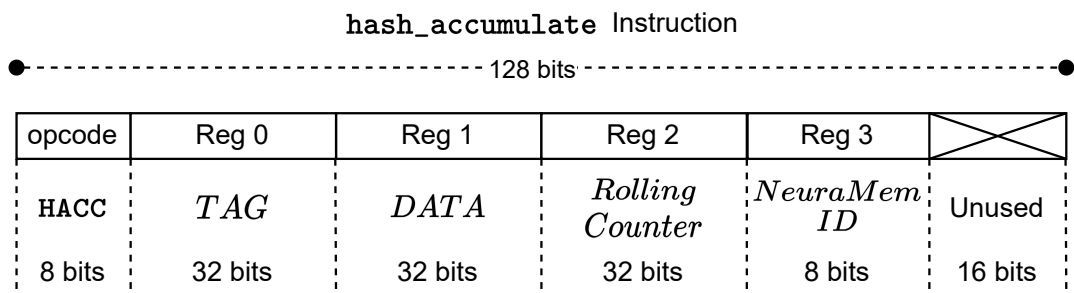
Figure 6.10: HACC instruction bit layout.
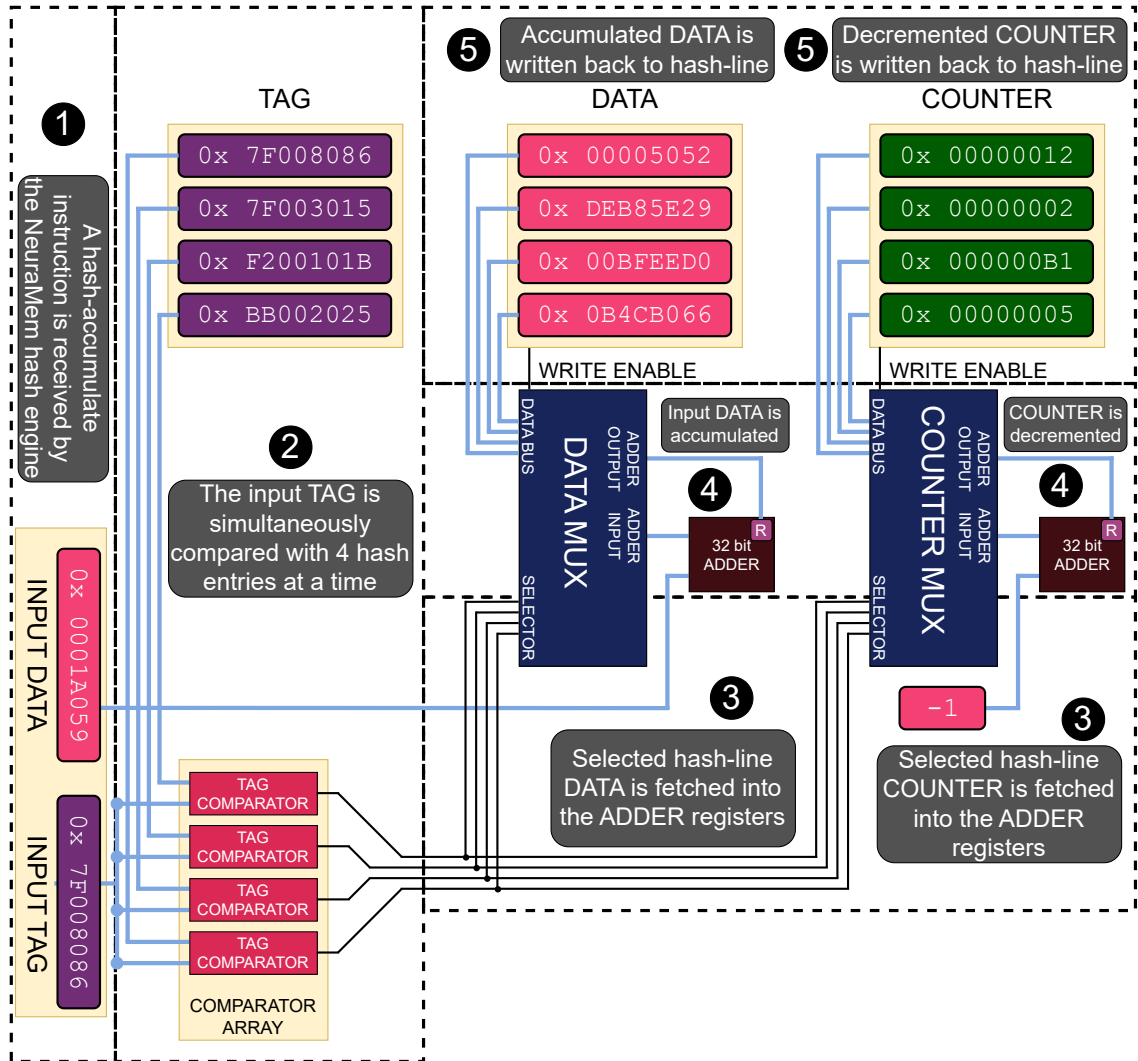
Figure 6.11: The NeuraMem Hash-Engine accumulates a single partial product using the `HACC` instruction.

The process starts in step ❶, where the Hash-Engine receives a `HACC` instruction from the Neura-Core units. This instruction's TAG is simultaneously compared with all the TAGs currently present on the HashPad (step ❷). The multiplexers select the hash-line with the matching TAG in step ❸. The corresponding hash-line's DATA gets accumulated with the `HACC` instruction's data. Simultaneously, the counter for that hash-line is decremented by one (step ❹). The accumulated data and the updated counter are then written back to the HashPad in step ❺. If the TAG from the instruction does not match any of the TAGs in the HashPad in step ❷, the Hash-Engine creates a new entry for the hash instruction and stores its content in a new hash-line.

Algorithm 4: `HACC` Instruction Execution

1: $index \leftarrow \text{Hash}(TAG)$
2: **if** $tag\_array[index] ==$ EMPTY **then**
3:    $data\_array[index] \leftarrow DATA$
4:    $counter\_array[index] \leftarrow COUNTER$
5: **else if** $tag\_array[index] == TAG$ **then**
6:    $data\_array[index] \mathrel{+}= DATA$
7:    $counter\_array[index] \mathrel{-}= 1$
8:    **if** $counter\_array[index] == 0$ **then**
9:      Hash Line Eviction Routine
10:    **end if**
11: **else**
12:    Hash Collision Routine
13: **end if**

Rolling Evictions: The Hash-Engine monitors the completion of partial product accumulation (via the COUNTER, as seen in Figure 6.9. Once the COUNTER reaches zero, indicating that all partial products for a particular TAG have been accumulated, the Hash-Engine automatically evicts the corresponding hash-line, and the accumulated result is written back to the main memory (HBM). This ensures that the hashed partial product spends the minimal possible number of cycles in the HashPad, addressing the memory bloat issue.

6.6.5 Dynamically Reseeding Hash-based Mapping

The performance benefits provided by the NeuraChip accelerator are primarily due to our sparsity-agnostic mapping algorithm, named Dynamically Reseeding Hash-based Mapping (DRHM). DRHM is designed to eliminate computational patterns, promoting an even distribution of workload across all computational resources. Traditional hash-based mappings often lead to concentrated ar-

eas of high activity, known as hot spots, especially when the hash function is optimized for a specific sparsity pattern but encounters a different one. An ideal solution would involve uniformly distributing computational tasks across resources. One such method is random mapping, where tasks are allocated to random resources. However, maintaining consistency in random mapping requires extensive record-keeping (a large lookup table), which is impractical.

We introduce a hybrid approach, Dynamically Reseeding Hash-based Mapping (DRHM), which combines the advantages of consistent lookup times in hashing, a distribution akin to random mapping, and minimal overhead similar to small lookup tables. This method significantly reduces the occurrence of hot spots in the allocation of computational resources.

DRHM utilizes a flexible mapping that adjusts based on a 'seed' parameter, denoted as γ. This parameter is specifically designed to alter the mapping, and consequently, the hash function dynamically. After each row of the input sparse matrix is computed, γ is initialized with a random number. DRHM offers two implementation approaches: one using the k upper bits of the TAG, and the other utilizing the k lower bits of the TAG. The lower-bit and upper-bit hashing equations that accommodate γ seed are presented in Equations 6.3 and 6.4.

$$H_l(\text{TAG}_{32}, \gamma) = ((\text{TAG}_{32} \ll k) \gg k) \cdot \gamma \mod N \tag{6.3}$$

$$H_h(\text{TAG}_{32}, \gamma) = ((\text{TAG}_{32} \gg k) \ll k) \cdot \gamma \mod N \tag{6.4}$$

where TAG represents the unique identifier for each row of the input graph. The term γ acts as a 'seed' to introduce randomness in the mapping. N signifies the total number of available output hash spaces. The operations "$\ll k$" and "$\gg k$" refer to bitwise left and right shifts by k positions, respectively. The modulus operation $\mod$ ensures that the result of the hash function falls within the predefined range of the hash table. These equations assume that the bit-shift operations conform to standard behavior where bits shifted beyond the boundary of the number's bit-width are discarded.

In our experiments, we assessed both upper k-bit address hashing and lower k-bit address hashing. We found that the lower k-bit address hashing method had a lower incidence of hash collisions, due to the higher variability in the lower bits of the address. Consequently, in all the work presented here, we employ the lower k-bit address hashing technique (Equation 6.3). The efficiency of compute mapping using our DRHM approach is evaluated in detail in Section 6.7.

Table 6.2: Individual Component Configuration

Component	Elements	Tile-4*	Tile-16*	Tile-64*
NeuraCore	Registers per pipeline	4	8	16
	Pipelines	2	4	8
	Multipliers	2	4	8
	Address Generators	1	2	2
	Ports	4	4	4
NeuraMem	TAG Comparators	1	4	8
	Hash-Engines	2	4	8
	Hashlines	4096	2048	2048
	Accumulators	128	256	512
	Ports	4	4	4

*Values represent the count of elements per component across three tile configurations.

6.7 Exploring the Design Space of NeuraChip

The flexibility of our NeuraSim simulator, which is used to simulate our NeuraChip accelerator, enables us to evaluate multiple NeuraChip configurations. We have two primary design goals: i) optimizing resource utilization across the accelerator to enhance speedup and ii) striking a balance between performance, chip area, and power consumption to make sure the advantages outweigh the costs.

Tile Size Variation: We introduce three distinct configurations of NeuraChip, named Tile-4, Tile-16, and Tile-64, derived from experimenting with various workloads. The detailed configurations of NeuraCore and NeuraMem components are provided in Table 6.2, while the overall accelerator configurations for these tile sizes are listed in Table 6.3. We focus on six key parameters to assess the architectural impact of these configurations, as shown in Figure 6.12.

Key observations include:

- **Register File Size**: Expanding the register file size allows more `MMH4` instructions to be in-flight and increases the number of read memory instructions that can be issued to HBM. Beyond 8 registers per pipeline (1024 bits per pipeline), we noticed that the DRAM channels are unable to keep up with the high memory demands. This bottleneck is evident in the rise in the cycles per instruction (CPI) and the number of stall cycles, as shown in Figure 6.12.

- **HashPad Size**: Choosing between smaller HashPads with a larger number of NeuraMems versus larger HashPads with fewer NeuraMems, the former proves advantageous for handling extremely sparse matrices. This configuration benefits from high accumulation throughput as

Table 6.3: NeuraChip Configuration

Parameter	Tile-4[*]	Tile-16[*]	Tile-64[*]
Tile Count	8	8	8
NeuraCores per tile	4	16	64
Total NeuraCores	32	128	512
NeuraMems per tile	4	16	64
Total NeuraMems	32	128	512
Memory Controller Count	8	8	8
Routers per tile	8	32	128
Total Routers	64	256	1024
Total Pipelines	64	512	4096
Register File Size per pipeline (bits)	512	1024	2048
Total Hash-Engines	64	512	4096
TAG comparators per Hash-Engine	2	4	8
Total TAG comparators	128	2048	32768
Total HashPad Size (MB)	1.5	3	12
Max operating frequency (GHz)	1	1	1

[*]Values represent the count of components/elements across the entire NeuraChip accelerators for three different tile configurations.

the number of accumulators increases with the number of NeuraMems. This can be seen in the larger number of in-flight HBM memory instructions in Figure 6.12.

- **Component Counts**: With 32, 128, and 512 NeuraCores and NeuraMems in Tile-4, Tile-16, and Tile-64, respectively, while more components enhance peak compute throughput, the configuration is bound by a peak DRAM bandwidth of 128 GB/s. Additionally, workloads do not require a 12 MB on-chip memory HashPad (of tile-64 configuration).

Hash-based Mapping Algorithm Variations: We tested four hash-based mapping schemes. The first, a ring-based mapping (see Figure 6.13), follows round-robin resource allocation, though encounters hot spots in workload distribution. The second, a modular hash-based mapping, uses prime numbers for workload mapping, proposed in previous studies [71, 137, 166, 211]. DRHM, shown in Figure 6.13, addresses hot spots in modular and ring-based mappings by reseeding the hash function after each row of computations. Lastly, we evaluate a random mapping that maintains

a lookup table for each entry. All four techniques are compared in Figure 6.14 for varying sparsity patterns.

Variations in `MMH` and `HACC` Instructions: NeuraChip introduces `MMH` and `HACC` instructions (bit layout of these instructions is illustrated in Figure 6.8 and Figure 6.10), supporting its decoupled architecture. We analyze the cycle count of various `MMH` instruction tile sizes, presented in a CPI histogram in Figure 6.15. `MMH4` emerges as the top choice, balancing temporal locality benefits and cycle count.

We compare the `HACC` instruction's efficiency using two eviction schemes: barrier-based eviction (`HACC-BE`) and our rolling eviction approach (`HACC-RE`). The latter's superiority in reducing average cycle completion is seen in Figure 6.16.

6.8   NeuraChip Evaluation

6.8.1   Experimental Setup

To evaluate the benefits of NeuraChip, we perform benchmarking across two distinct categories of workloads. The first category involves examining NeuraChip's efficiency in handling sparse matrix multiplication tasks. This evaluation uses a standard array of sparse matrices obtained from the Stanford SNAP sparse matrix collection [109]. Our evaluation includes a comparison with some of the latest state-of-the-art sparse matrix accelerators [210, 212] and off-the-shelf mainstream hardware platforms. NeuraChip is benchmarked against the Intel MKL library [185] with an Intel Xeon E5-2630 CPU. We also compare against cuSPARSE [135] and CUSP [43] NVIDIA libraries, as run on a Hopper architecture H100 GPU, and we also consider for comparison an AMD's MI100 GPU using the hipSPARSE library with a rocSPARSE backend [27, 150]. For accelerator comparisons, we compare NeuraChip against OuterSPACE [142] SpArch [212], and Gamma [210]. Additionally, as to the second category of workloads, our evaluation targets a Graph Convolutional Network (GCN) [102] layer using various datasets, allowing us to compare NeuraChip against existing Graph Neural Network (GNN) accelerators EnGN [118], GROW [86], HyGCN [199], and FlowGNN [153].

6.8.2   Simulator Framework

In this study, we present NeuraSim, a cycle-accurate, multi-threaded, modular simulation engine inspired by the Structural Simulation Toolkit (SST) [151]. NeuraSim's modular framework

Figure 6.12: Architectural impact of GCN model varying tile configuration on Cora dataset. Values are normalized to Tile-4 configuration.

(a) Ring Hash-based Mapping

(b) Dynamically Reseeded Hash-based Mapping
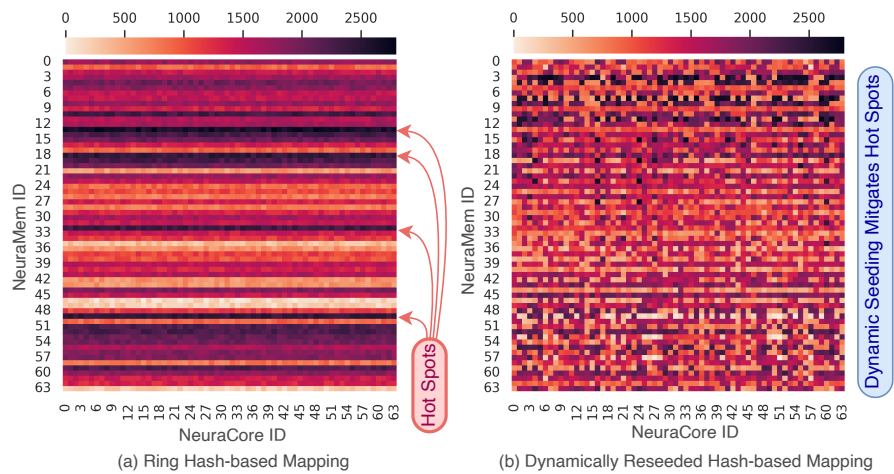
Figure 6.13: Compute mapping heat map, where the X-axis represents multiplications mapped to NeuraCores and Y-axis represents accumulations mapped to NeuraMem.
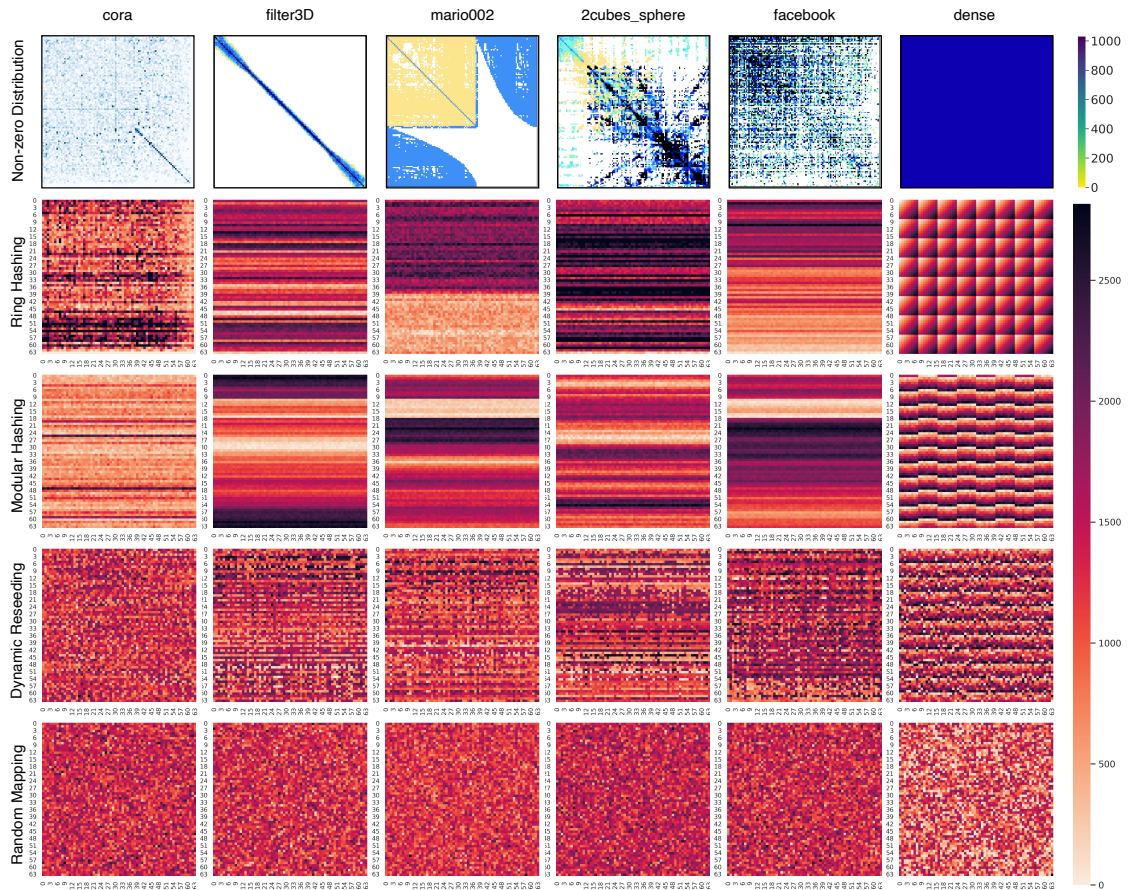
Figure 6.14: Computation mapping heat maps for four distinct hash-based mapping methods, evaluated across five sparse matrices and one dense matrix multiplication. The dynamic reseeding mapping technique is insensitive to sparsity patterns and effectively addresses hot spots in dense matrix computations.

allows for flexible integration of new architectural features, without the need for an entire overhaul of the simulation engine. Developed using POSIX threads (`pthreads`), NeuraSim facilitates parallel simulation. Its dispatcher unit recognizes independent tasks and concurrently executes them on different threads. Additionally, NeuraSim employs MongoDB for backend data storage. NeuraSim also incorporates HBM2 memory simulation, integrating with DRAMsim3 [113], a cycle-accurate and validated DRAM simulator.

Regarding simulation efficiency, NeuraSim achieves 112 Kilocycles per second (KCPS), 48 KCPS, and 11 KCPS on average for the Tile-4, Tile-16, and Tile-64 configurations, respectively. NeuraSim is open-source and faithfully simulates the extended NeuraChip ISA. The NeuraSim source code is accessible on our GitHub repository[1][2].

6.8.3 Comparative Analysis with Sparse Matrix Accelerators

In Figure 6.17, the performance of the NeuraChip in sparse matrix multiplication tasks is compared against various off-the-shelf high-end CPU and GPU platforms, as well as against state-of-the-art SpGEMM accelerators.

NeuraChip provides benefits when compared to Intel's MKL running on an Xeon CPU, surpassing it by a factor of $22.1\times$. Additionally, when compared against NVIDIA's H100 GPU using the CUSP library, NeuraChip achieves a performance boost of $13.3\times$. In comparison to the prior leading sparse matrix multiplication accelerator, Gamma, NeuraChip achieves an average performance improvement of $1.5\times$.

As we can see, NeuraChip outperforms the CPU and GPU computing platforms in all cases. In particular, average performance improvements of $22.2\times$ over the CPU, a $17.1\times$ and $13.3\times$ average speedup over the NVIDIA Hopper GPU using the cuSPARSE and CUSP libraries, respectively, and $16.7\times$ speedup on average over the AMD's MI100 GPU using the hipSPARSE library.

Further, the performance of NeuraChip is evaluated against two outer-product-based sparse matrix accelerators: OuterSPACE [142] and SpArch [212]. While OuterSPACE leverages input data reuse, it encounters excessive generation of partial products (the memory bloat issue), leading to degraded performance. SpArch addresses this with on-chip merger trees; however, these trees require large comparator arrays, occupying about 60% of the chip area. NeuraChip counters the memory bloat through an on-chip cache organization with rolling counters, effectively managing the evic-

---

[1]`https://github.com/NeuraChip/neurachip`
[2]`https://neurachip.us/`

tion of accumulating partial products and alleviating the bloat issue. In comparison, NeuraChip surpasses OuterSPACE and SpArch by factors of $6.6\times$ and $2.4\times$, respectively.

Additionally, the performance of NeuraChip is compared with a row-wise product-based SpGEMM accelerator, Gamma [210], which is based on Gustavson's algorithm. Gamma employs a resource-intensive storage mechanism, FiberCache, to prefetch data, aiming to reduce data fetch latency and prevent compute stalls. However, this approach results in data remaining idle in the caches prior to being accessed by the processing elements. NeuraChip, in contrast, optimizes on-chip storage through a rolling-eviction strategy, enabling automatic eviction of partial products after the reduce operation is complete. Against Gamma, NeuraChip demonstrates a performance superiority of $1.5\times$ average speedup.

6.8.4 Comparative Analysis of GNN Accelerators

In Figure 6.18, we compare the GNN performance of NeuraChip against various state-of-the-art GNN accelerators. The NeuraChip configuration used for GNN assessment differs from that used to compare to SpGEMM accelerators in Table 6.3. Specifically, for the Tile-16 configuration in the GNN accelerator analysis, an architecture comprising 8 tiles is used. Each tile includes a 16×16 grid of NeuraCores, with each core featuring a quad-pipeline design. We have significantly reduced the number of TAG comparators and port buffers, while retaining the hashpad sizes. This particular configuration is capable of delivering a peak performance of 8192 GFLOPs, with an average power consumption of 4.3W.

First, we consider EnGN, a hash-based GNN accelerator [118], and GROW [86]. EnGN employs a unique ring-based edge reducer to efficiently map vertex IDs. However, it encounters challenges in achieving a uniform distribution of computational tasks among its processing elements. In comparison, NeuraChip demonstrates superior performance, outperforming EnGN by 29% on average. This improvement is primarily attributed to the dynamic reseed hashing function within NeuraChip, which ensures balanced task distribution across its computational resources, namely NeuraCore and NeuraMem, thus minimizing processing delays.

GROW utilizes a row-wise multiplication method, incorporating hardware and software co-design elements. A notable aspect of GROW's software strategy is its reliance on graph partitioning, which significantly increases the computational overhead for GNN processing. From a hardware perspective, GROW is equipped with vector processors and employs streaming buffers for handling input and output matrix data. Despite these features, GROW encounters issues sim-

ilar to those seen in Gamma's prefetcher system, where data idling results in suboptimal usage of on-chip memory resources. Comparative performance metrics indicate that NeuraChip surpasses GROW's performance by an average of 58%.

Next, we evaluate our accelerator compared to HyGCN, a hybrid Graph Neural Network (GNN) accelerator, which has specialized engines for aggregation and combination phases [199]. The primary advantage of HyGCN's architecture is its ability to pipeline computations, which is particularly beneficial for GNN layers that typically alternate between aggregation and combination phases. However, a significant limitation arises when the compute duration for one phase substantially exceeds the other, leading to a pipeline stall due to the uneven execution duration of each pipeline stage.

Instead, NeuraChip incorporates distinct components specifically for multiplication and accumulation operations, utilized in both the aggregation and combination phases. This design choice renders NeuraChip impervious to the inefficiencies caused by varying computational times between aggregation and combination phases. On average, NeuraChip outperforms HyGCN's performance by 69%.

Our final comparison is with FlowGNN [153], a reconfigurable dataflow GNN accelerator comprising Node Transformation Units (NTs) and Message Passing Units (MPs). FlowGNN employs queues for real-time task buffering and relies on dynamic pull-based mapping for task distribution to NTs and MPs. In contrast, NeuraChip adopts a push-based mapping strategy for multiplication tasks and a hash-based approach for accumulation tasks. The Dispatcher in NeuraChip assigns `MMH4` instructions to NeuraCores, optimizing input data temporal locality (reuse in NeuraCore register files). The dynamic reseeding hash-based mapping, as detailed in Section 6.6.5, ensures uniform workload distribution regardless of sparsity patterns. Consequently, NeuraChip achieves an average speedup of 30% over GCN workloads tested on the FlowGNN architecture.

6.8.5 Power Consumption and Area Analysis

We assess our accelerator's area and power overheads by implementing its design in Register Transfer Level (RTL). Using Cadence Genus Synthesis Solutions, we synthesize these RTL components targeting an ASAP7 technology library [40], allowing us to determine the area and power consumption for each proposed microarchitectural element. The synthesized chip area requirements for NeuraChip amount to $2.37mm^2$, $10.2mm^2$, and $35.26mm^2$ for the Tile-4, Tile-16, and Tile-64 configurations, respectively.
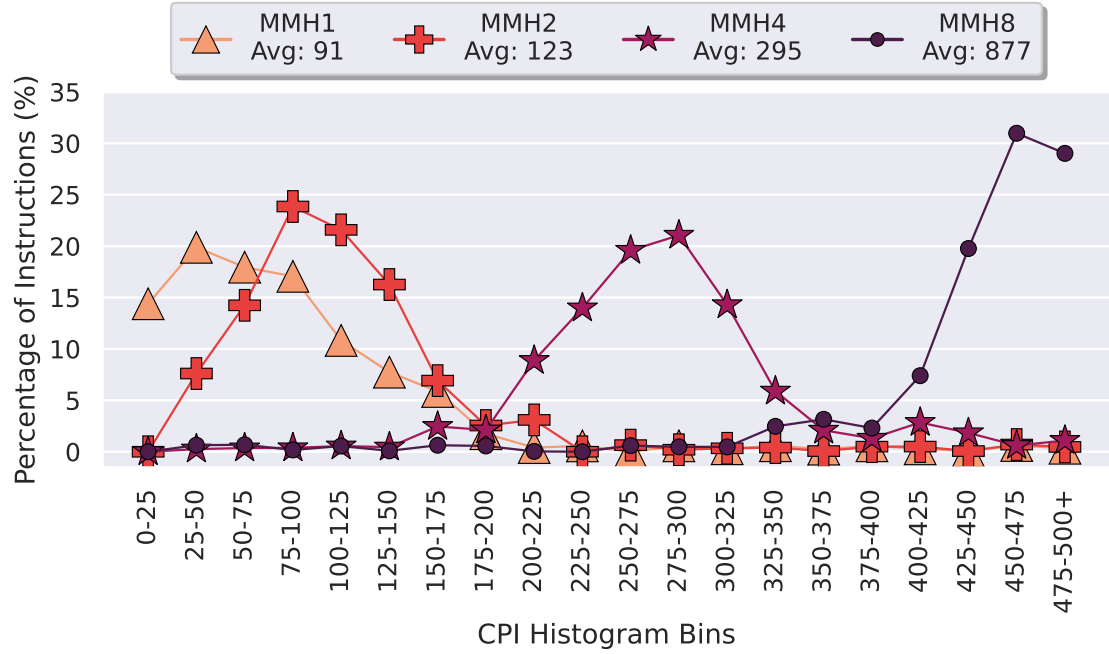
Figure 6.15: Cycles Per Instruction (CPI) histogram plot for four `MMH` instructions with varying tile sizes.

Table 6.4: NeuraChip Power and Area Breakdown

Unit	Area (mm^2)			Average Power (W)		
	Tile-4	**Tile-16**	**Tile-64**	**Tile-4**	**Tile-16**	**Tile-64**
NeuraCore	0.28	2.74	9.36	1.05	1.86	5.76
NeuraMem	1.22	5.10	18.64	6.85	7.36	11.19
Router	0.49	1.98	6.88	2.15	4.88	4.43
Memory Controller	0.38	0.38	0.38	1.41	1.96	2.84
Total	2.37	10.2	35.26	11.46	16.06	24.22
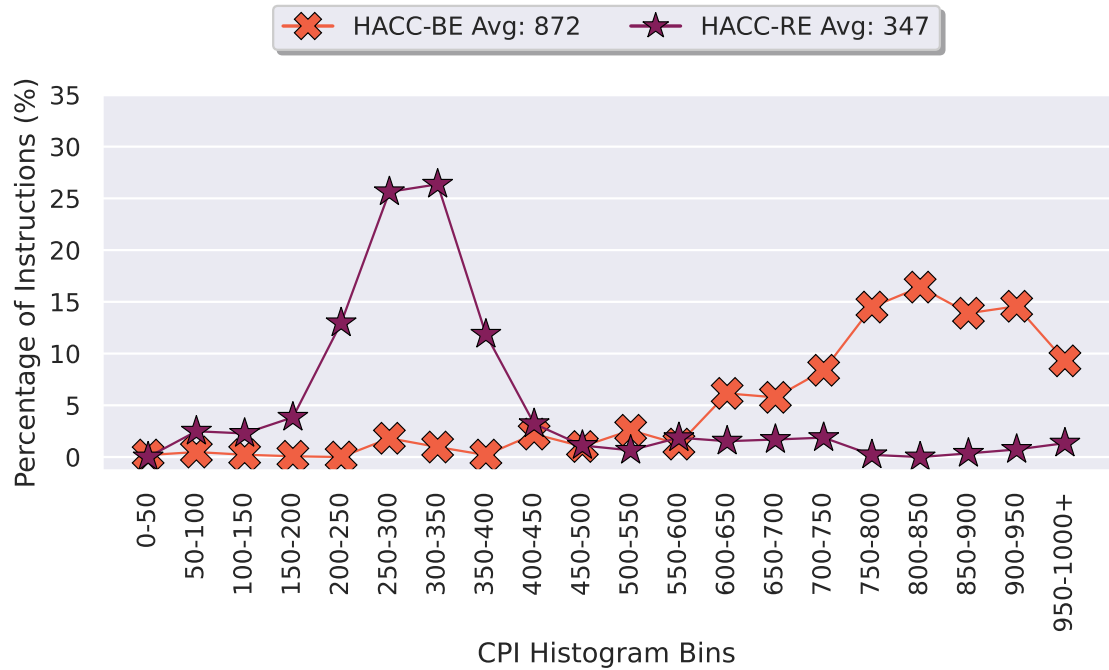
Figure 6.16: Cycles Per Instruction (CPI) histogram plot for `HACC` instructions with barrier based evictions `HACC-BE` and rolling evictions `HACC-RE`.
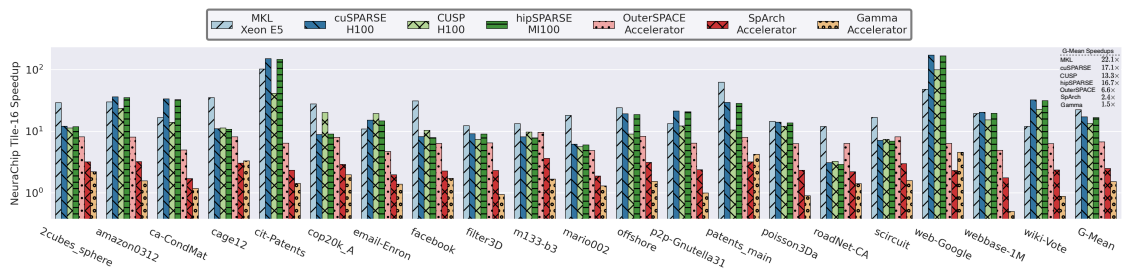
Figure 6.17: Speedup of NeuraChip Tile-16 configuration compared against CPU, GPUs, and SpGEMM accelerators benchmarking sparse matrix multiplication.

Figure 6.18: Percentage speedup of Tile-16 configuration over prior GNN accelerators with GCN workload over different graph datasets.

The breakdown of NeuraChip's area and power is shown in Table 6.4. The majority of the area requirement for NeuraChip is allocated to the NeuraMem unit, as it includes the tag comparator array and the hash-pad (on-chip storage).

6.9 Comparison with Prior Custom Accelerators

Next, we discuss previous studies on sparse matrix multiplication (spGEMM) and Graph Neural Networks (GNN).

SpGEMM Accelerators: InnerSP [10] introduces an accelerator that applies the inner-product method for matrix multiplication. This method offers advantages, such as eliminating the need for on-chip memory for accumulation. However, it suffers from limited input data reuse from both matrices, leading to performance issues when the sparsity patterns do not align with their task mapping algorithm. MatRaptor [169] employs a row-wise multiplication strategy and a round-robin greedy algorithm for allocating input rows to processing elements (PEs). Although this approach enhances input data reuse, it struggles with skewed sparsity patterns. The simplistic round-robin distribution may result in computational hot spots (as elaborated in Section 6.7). SIGMA [147] offers an SpGEMM accelerator equipped with adaptable interconnects. Utilizing a smart global controller, SIGMA dynamically assigns each non-zero pair to PEs via a Benes network. Despite its efficiency in general SpGEMM tasks, SIGMA is less effective with large sparse matrix computations due to the substantial overhead introduced by its bitmap compression format.

Table 6.5: Performance comparison of SpGEMM workload accelerators across various off-the-shelf hardwares.

Architectural Parameters	Xeon E5	NVIDIA H100	AMD MI100
Compute Units	8 Cores AVX2	7296 FP64	7680 FP64
Frequency (GHz)	2.9	1.6	1.5
Peak Performance	186 GFLOPs	26 TFLOPs	11.5
SpGEMM Perf.$^\Phi$ (GOP/s)	1.12	1.86	1.48
On-chip Memory	15 MB$^\tau$	50 MB†	8 MB†
Off-chip Memory	DDR4 136GB/s	HBM 2TB/s	HBM 1.2TB/s
Technology (nm)	32	4	7
Area (mm^2)	356	814	750
Power (W)	85$^\diamond$	300$^\diamond$	300$^\diamond$
Tile-16 Speedup	22.1$\times$	13.3$\times$	16.7$\times$

$\diamond$Max thermal dissipation power from datasheet
† L2 cache size $^\tau$ L3 cache size $^\Phi$Computed on common set of matrices as shown in Table 6.1.

GNN Accelerators: LISA [115] performs GNN computations on Coarse-Grained Reconfigurable Arrays (CGRAs). LISA generates a dataflow graph and utilizes a simulated annealing method for mapping. I-GCN [66] aims to enhance data locality through an *islandization* strategy, clustering densely connected nodes to reduce off-chip memory accesses. However, both the simulated annealing and graph clustering methods introduce considerable computational overheads.

6.10 NeuraChip Summary

In this thesis we presented, NeuraChip, that demonstrates the potential advantages that sparse matrix multiplication workloads can gain from a decoupled architectural design. NeuraChip optimizes multiplication and aggregation phases separately using two distinct components. We have presented an open-source[3], cycle-accurate simulator called NeuraSim, used to demonstrate the effectiveness of our design. The acceleration of GNN workloads is achieved through a blend of high-level optimizations and microarchitectural features. We have synthesized our design using RTL, thereby allowing us to calculate power and area requirements for various NeuraChip Tile sizes. NeuraChip is able to outperform state-of-the-art SpGEMM accelerator by a factor of $1.5\times$ and prior GNN accelerator by $1.46\times$ on average.

---

[3]`https://github.com/NeuraChip/neurachip`

Table 6.6: Performance comparison of state-of-the-art SpGEMM accelerators across various NeuraChip (**NC**) system configurations.

Architectural Parameters	Outer SPACE	SpArch	Gamma	NC Tile-4	NC Tile-16	NC Tile-64
Compute Units	256 PEs	2×8 Mults 16×16 Merger	32 PEs Radix-64	2×4 Neura-Cores	2×16 Neura-Cores	2×64 Neura-Cores
Frequency (GHz)	1.5	1	1	1	1	1
Peak Performance	384 GFLOPs	32 GFLOPs	32 GFLOPs	8 GFLOPs	32 GFLOPs	128 GFLOPs
SpGEMM Perf.$^\Phi$ (GOP/s)	2.9	10.4	16.5	5.15	24.75	30.69 93.17^α
On-chip Memory	4 MB	15 MB*	3 MB*	0.75 MB$^\delta$	3 MB$^\delta$	12 MB$^\delta$
Off-chip Memory	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s	HBM 128GB/s
Technology (nm)	32	40	45	7	7	7
Area (mm^2)	86.74	28.49	30.6‡ 20.44‡	2.37	10.2	35.26
Power (W)	24	9.26	❖	11.46	16.06	24.22
Energy Efficiency (GOPS/W)	0.120	1.123	❖	0.449	1.541	1.266
Area Efficiency (GOPS/mm^2)	0.034	0.365	0.539	2.171	2.426	0.870
Tile-16 Speedup	6.6$\times$	2.4$\times$	1.5$\times$	4.8$\times$	1$\times$	0.807$\times$

❖Gamma does not provide a power performance model $^\delta$HashPad Size *FiberCache Size
$^\alpha$ Simulated using dual stacked HBM providing peak bandwidth of 256 GB/s ‡Gamma synthesizes accelerator using both 45 nm and 40 nm processes, resulting in computing areas of 30.6 mm^2 and 20.44 mm^2, respectively
*Represents column fetchers, row prefetchers, and partial matrix fetchers and writers. $^\Phi$Computed on common set of matrices as shown in Table 6.1.

Chapter 7

Conclusion

This dissertation presents the design and implementation of accelerators for graph computing. Accelerating graph computations poses significant challenges due to the irregular structure of graphs. Through this research, we contribute to the field in three major aspects. Firstly, we analyze the architectural implications of Graph Neural Networks (GNNs) on GPUs and state-of-the-art accelerators. Secondly, we develop algorithmic strategies to enhance the Sparse Generalized Matrix Multiply (SpGEMM) kernel, which is crucial for GNN workloads. Lastly, we introduce a custom Coarse-Grained Reconfigurable Array (CGRA)-based accelerator for GNNs, designed to overcome the bottlenecks identified during GNN profiling.

7.1 GNN Workload Characterization

This dissertation introduces GNNMark, a comprehensive benchmark suite specifically designed to evaluate the performance of Graph Neural Networks (GNNs) on GPUs. This initiative marks the first attempt within the architectural research community to focus on a GNN training-oriented benchmark suite. Utilizing GNNMark, we conduct an in-depth analysis of GNN workloads to identify the architectural challenges associated with training GNN models on GPU platforms. Our findings contribute novel insights into the primary architectural bottlenecks encountered during GNN training, alongside strategies for their potential mitigation.

The performance characteristics of a single GNN model can vary significantly based on the input graph's structure. Contrary to Deep Neural Networks (DNNs), where General Matrix Multiply (GEMM) and convolution operations predominate, our analysis reveals that graph processing within GNNs predominantly requires integer operations. This observation highlights the

need for enhancing integer computation performance to improve overall GNN execution efficiency. Moreover, our study highlights a substantial impact of instruction fetch stalls on GNN performance, indicating that GPU instruction cache limitations could serve as a significant bottleneck. Additionally, our research presents findings on the sparsity observed during GNN training and the efficacy of strong scaling, thereby offering a comprehensive overview of the performance dynamics of GNN training on GPU systems.

7.2 SMASH: GNN Algorithmic Acceleration

Further in this dissertation , we describe the advancements made in the domain of SpGEMM algorithmic acceleration, encapsulating a series of pivotal contributions towards optimizing SpGEMM kernels. The cornerstone of this effort is characterized by a multifaceted analysis aimed at identifying the inherent challenges in developing efficient SpGEMM kernels. This investigation covers a critical evaluation of various sparse matrix multiplication techniques, scrutinizing the merits and drawbacks inherent to each method.

A significant development in this research is the development of a novel sparse matrix storage format, termed as MAPCSR (Memory Aligned Parallel CSR). This innovative approach facilitates parallel computation of each sparse matrix row, significantly enhancing memory access efficiency through memory-aligned storage. The implementation of MAPCSR has demonstrably bolstered SpGEMM performance, yielding a remarkable $1.58\times$ improvement.

Building upon these foundational advancements, we introduce SMASH (Sparse Matrix Atomic Scratchpad Hashing), an optimized SpGEMM kernel tailored for distributed memory architectures. SMASH is implemented in three distinct versions, each iteration exploiting specific architectural features to incrementally enhance performance. This tiered approach not only exemplifies the adaptability and scalability of SMASH but also underpins its efficacy in leveraging the unique capabilities of custom accelerators. The iterative development of SMASH in this dissertation significantly propels forward the state-of-the-art in SpGEMM algorithmic acceleration.

7.3 NeuraChip: GNN Hardware Acceleration

This segment of the dissertation introduces NeuraChip, a state-of-the-art spatial accelerator tailored for Graph Neural Network workloads, marking a significant stride in hardware accel-

eration for graph computing. The design and implementation of NeuraChip embodies a series of innovative contributions that collectively address critical challenges in GNN acceleration.

Heterogeneous Processing Approach: Central to NeuraChip's architecture is a heterogeneous processing strategy that divides computation tasks into distinct multiplication and accumulation phases. This decoupled computation pipeline is meticulously designed to improve data reuse, optimizing the efficiency of operations through strategic component mapping.

Adaptive Hash-Based Compute Mapping: NeuraChip incorporates an adaptive, dynamic reseeding hash-based compute mapping (DRHM) mechanism, specifically engineered for GNN computations. Leveraging the consistent lookup times provided by hash functions, DRHM distributes computing tasks across the accelerator's resources. By dynamically updating the seed at regular computation intervals, it ensures an uniform workload distribution, effectively neutralizing the challenges posed by the varying sparsity patterns inherent in graph data.

Mechanism for Rolling Evictions: To address the bottleneck of memory congestion, a consequence of accumulating partial products, NeuraChip introduces a novel rolling eviction strategy. This approach facilitates the timely eviction of partial products, thereby mitigating memory bloat and ensuring sustained high throughput.

NeuraChip establishes itself as a significant development in the domain of hardware acceleration of GNNs, showcasing an architecture that is not only highly efficient but also adaptable to the diverse and dynamic nature of graph-based computations. The NeuraChip segment of this dissertation underscores the potential of specialized hardware designs in overcoming the unique challenges of accelerating GNN workloads, paving the way for future innovations in the field.

7.4 Contributions of this Dissertation

This dissertation has been dedicated to advancing the field of Graph Neural Network acceleration through a comprehensive exploration of both hardware and software aspects. It has established a new benchmark in the study of GNNs, providing a suite of contributions that lay a solid foundation for future research in this domain. These contributions span from the development of benchmark suites for GNN evaluation to novel algorithmic strategies for sparse matrix multiplication, and from the introduction of an innovative hardware accelerator to the creation of a cycle-accurate simulator for Coarse-Grained Reconfigurable Arrays (CGRA). Specifically, this work includes:

1. The development of a benchmark suite specifically designed for assessing GNN performance, addressing both single and multi-GPU environments. This suite facilitates a nuanced understanding of GNN workloads, enabling targeted improvements in GPU-based graph processing.

2. The presentation of GPU and multi-GPU benchmarking results for GNNs, which shed light on the architectural implications of GNN workloads. These insights contribute to the optimization of GPU resources for enhanced GNN processing efficiency.

3. The introduction of a novel sparse matrix storage format that significantly advances the state of SpGEMM (Sparse Generalized Matrix Multiply) operations. This format underpins the algorithmic optimizations for SpGEMM workloads, demonstrating substantial performance improvements.

4. An in-depth evaluation of SpGEMM optimizations on a custom accelerator, showcasing the potential of hardware-specific adaptations to elevate GNN processing speeds.

5. The proposal of a dynamically reseeding hash-based mapping algorithm tailored for GNN workloads, which optimizes computation distribution and efficiency in hardware accelerators designed for GNNs.

6. The creation of NeuraSim, a cycle-accurate simulator for CGRA architectures, facilitating precise analysis and optimization of GNN accelerator designs.

7. A comprehensive chip power and area analysis of the NeuraChip GNN accelerator, providing valuable metrics for assessing the viability and efficiency of GNN-specific hardware solutions.

Together, these contributions not only provide a path towards optimized GNN processing but also equip the research community with the tools and methodologies necessary for continued innovation in the acceleration of graph neural networks.

7.5 Future Work

The primary objective of this dissertation was to advance the field of Graph Neural Network acceleration through a comprehensive exploration of benchmark suites, algorithmic optimizations, and hardware solutions. Despite the significant strides made in this research, the domain of

GNN acceleration remains vast, with ample opportunities for further exploration and innovation. The following outlines potential directions for future research, building upon the foundational work presented in this thesis:

1. **Enhanced Integration of GNN Workloads with Emerging Hardware Accelerators**: The current landscape of hardware accelerators, including but not limited to NVIDIA's Tensor Cores [122] and Google's TPUs [99, 107], presents a promising avenue for improving GNN performance and energy efficiency. Future work could explore deeper integrations with these accelerators, adapting GNN algorithms to leverage their specific architectural advantages more effectively.

2. **Advanced Memory Technologies for GNN Acceleration**: The evolution of memory technologies, such as in-memory processing [85] and near-memory processing [95], offers new possibilities for enhancing GNN execution. Investigating holistic designs that incorporate these advanced memory solutions could lead to significant improvements in GNN processing speed, energy efficiency, and overall system performance.

3. **Scalability and Efficiency in Multi-GPU Systems**: While this dissertation addressed multi-GPU benchmarking and architectural implications for GNNs, scaling performance linearly with the addition of GPUs remains a challenge. Future research could focus on developing new techniques for GPU-level cooperative thread array (CTA) [93] scheduling and thread migration [42] to minimize performance overhead in multi-GPU systems.

4. **Dynamic Scheduling and Control Flow Optimization for GNN Workloads**: Addressing the limitations in control flow transition between CPUs and GPUs is crucial for optimizing the performance of GNN workloads [98]. Investigating hardware-based GPU schedulers integrated into CPU cores could reduce kernel-launch overhead and memory synchronization challenges, enabling more efficient CPU-GPU collaboration.

5. **Computing Capabilities in Network Devices for Distributed GNN Processing**: As network switches evolve to incorporate computing capabilities, there exists an opportunity to offload certain GNN processing tasks to these devices, potentially reducing data movement costs and improving the efficiency of distributed GNN training and inference [179].

6. **Exploration of Novel Computing Devices for GNN Acceleration**: The advent of novel computing devices extends the horizon for GNN acceleration. Future research should con-

sider incorporating a broader array of devices, including dedicated accelerators, non-traditional memory devices, and smart network components, to achieve comprehensive performance, energy, reliability, and security enhancements.

Building on the contributions of this dissertation, these future research directions aim to further push the boundaries of GNN acceleration, addressing the complex and evolving challenges inherent in graph-based computations and fostering the development of more efficient, scalable, and adaptable GNN processing systems.

Bibliography

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)*, 54(9):1–38, 2021.

[2] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[3] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall Englewood Cliffs, NJ, 1993.

[4] Lilas Alrahis, Satwik Patnaik, Muhammad Shafique, and Ozgur Sinanoglu. Embracing graph neural networks for hardware security. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[5] Francesca Arrigo and Desmond J Higham. Sparse matrix computations for dynamic network centrality. *Applied network science*, 2(1):1–19, 2017.

[6] Giuseppe Ascia, Vincenzo Catania, Maurizio Palesi, and Davide Patti. Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. *IEEE transactions on computers*, 57(6):809–820, 2008.

[7] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[8] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse

matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[9] David A. Bader, Kamesh Madduri, and Satoshi Matsuoka. Data-driven science in the exascale era. *Computer*, 52(8):18–26, 2019.

[10] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. Innersp: A memory efficient sparse matrix multiplication accelerator with locality-aware inner product processing. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 116–128, Piscataway, NJ, 2021. IEEE.

[11] Allison H Baker, Todd Gamblin, Martin Schulz, and Ulrike Meier Yang. Challenges of scaling algebraic multigrid across modern multicore architectures. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 275–286. IEEE, 2011.

[12] Alexandru T. Balaban. Applications of graph theory in chemistry. *Journal of Chemical Information and Computer Sciences*, 16(3):164–176, 1976.

[13] Albert-László Barabási. *Network science*. Cambridge University Press, 2016.

[14] Trinayan Baruah, Kaustubh Shivdikar, Shi Dong, Yifan Sun, Saiful A Mojumder, Kihoon Jung, José L Abellán, Yash Ukidave, Ajay Joshi, John Kim, et al. 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. IEEE, 2021.

[15] Danielle S Bassett and Olaf Sporns. Network neuroscience. *Nature Neuroscience*, 20(3):353–364, 2017.

[16] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[17] Scott Beamer, Krste Asanović, and David A. Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 56–65. IEEE, 2017.

[18] Juliana S Bernardes, Fabio RJ Vieira, Lygia MM Costa, and Gerson Zaverucha. Evaluation and improvements of clustering algorithms for detecting remote homologous protein families. *BMC bioinformatics*, 16(1):1–14, 2015.

[19] Rohit K Bhullar, Lokesh Pawar, Vijay Kumar, et al. A novel prime numbers based hashing technique for minimizing collisions. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, pages 522–527. IEEE, 2016.

[20] Claudio Borio, Mathias Drehmann, and Kostas Tsatsaronis. *Systemic Risk, Crises, and Macroprudential Regulation*. MIT Press, 2014.

[21] David R Bowler, Tsuyoshi Miyazaki, and Michael J Gillan. Parallel sparse matrix multiplication for linear scaling electronic structure calculations. *Computer physics communications*, 137(2):255–273, 2001.

[22] Thomas Bradley. Gpu performance analysis and optimisation. *NVIDIA Corporation*, 2012.

[23] Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[24] Edward Bullmore and Olaf Sporns. The brain as a complex system: using network science as a tool for understanding the brain. *Brain Connectivity*, 1(4):295–308, 2011.

[25] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, pages 233–244, New York, NY, 2009. ACM.

[26] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, New York, NY, 2011. ACM.

[27] C Bunn, Harrison Barclay, A Lazarev, F Yusuf, J Fitch, J Booth, Kaustubh Shivdikar, and D Kaeli. Student cluster competition 2018, team northeastern university: Reproducing performance of a multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake on the amd epyc 7551 architecture. *Parallel Computing*, 90:102568, 2019.

[28] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[29] Guido Caldarelli. *Scale-free networks: Complex webs in nature and technology*. Oxford University Press, 2007.

[30] Philippe Camacho, Alejandro Hevia, Marcos Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In *Information Security: 11th International Conference, ISC 2008, Taipei, Taiwan, September 15-18, 2008. Proceedings 11*, pages 471–486. Springer, 2008.

[31] William M Campbell, Charlie K Dagli, and Clifford J Weinstein. Social network analysis with content and graphs. *Lincoln Laboratory Journal*, 20(1):61–81, 2013.

[32] Zhiruo Cao, Zheng Wang, and Ellen Zegura. Performance of hashing-based schemes for internet load balancing. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, volume 1, pages 332–341. IEEE, 2000.

[33] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[34] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. Semi-supervised learning. *IEEE Transactions on Neural Networks*, 10:529–542, 2006.

[35] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013.

[36] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[37] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. Regnn: A redundancy-eliminated graph neural networks accelerator. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 429–443. IEEE, 2022.

[38] Wen Chen, Yu Zhang, H. Peter Huynh, and Beng Chin Ooi. Graph processing on gpus. *ACM SIGMOD Record*, 47(1):27–38, 2018.

[39] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur)*, 50(1):1–36, 2017.

[40] Lawrence T Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. Asap7: A 7-nm finfet predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016.

[41] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[42] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[43] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[44] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, 2010.

[45] Jack W Davidson and Sanjay Jinturkar. An aggressive approach to loop unrolling. Technical report, Citeseer, 1995.

[46] Timothy A Davis. Graph algorithms via suitesparse: Graphblas: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[47] Mauro Dell'Amico, Stavros N Hadjidimitriou, Manuel Iori, and Stefano Novellani. Public transportation systems as complex networks: A graph theory approach. In *International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, pages 717–722. IEEE, 2015.

[48] Zulong Diao, Xin Wang, Dafang Zhang, Yingru Liu, Kun Xie, and Shaoyao He. Dynamic spatial-temporal graph convolutional neural networks for traffic forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 890–897, 2019.

[49] Shaojin Ding, Tianlong Chen, and Zhangyang Wang. Audio lottery: Speech recognition made ultra-lightweight, noise-robust, and transferable. In *International Conference on Learning Representations (ICLR)*, 2022.

[50] Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. Characterizing the microarchitectural implications of a convolutional neural network (cnn) execution on gpus. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 96–106, 2018.

[51] Shi Dong and David Kaeli. Dnnmark: A deep neural network benchmark suite for gpus. In *Proceedings of the General Purpose GPUs*, pages 63–72, 2017.

[52] Shi Dong and David Kaeli. Dnnmark: A deep neural network benchmark suite for gpus. In *Proceedings of the General Purpose GPUs*, pages 63–72, 2017.

[53] PN Druzhkov and VD Kustikova. A survey of deep learning methods and software tools for image classification and object detection. *Pattern Recognition and Image Analysis*, 26(1):9–15, 2016.

[54] David Ediger, Jason Riedy, David A Bader, and Henning Meyerhenke. Tracking structure of streaming social networks. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1691–1699. IEEE, 2011.

[55] Edwin J. Elton and Martin J. Gruber. Modern portfolio theory and investment analysis. *Journal of Finance*, 1976.

[56] David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. *Algorithms and theory of computation handbook*, 1:9–1, 1999.

[57] Leonhard Euler. The seven bridges of königsberg. *The world of mathematics*, 1:573–580, 1956.

[58] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[59] Alex Fornito, Andrew Zalesky, and Edward Bullmore. Graph-based analysis of brain connectivity. *The Oxford Handbook of Functional Brain Imaging in Neuropsychology and Cognitive Neurosciences*, 2017.

[60] Hao Fu, Matthias Wolf, Tom R.W. Scogland, Guangming Yang, and Weikuan Zhou. Exagraph: Enabling graph applications for exascale systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 444–454. IEEE, 2019.

[61] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[62] Fernando Gama, Antonio G Marques, Geert Leus, and Alejandro Ribeiro. Convolutional neural network architectures for signals supported on graphs. *IEEE Transactions on Signal Processing*, 67(4):1034–1049, 2018.

[63] Wanling Gao, Fei Tang, Lei Wang, Jianfeng Zhan, Chunxin Lan, Chunjie Luo, Yunyou Huang, Chen Zheng, Jiahui Dai, Zheng Cao, et al. Aibench: an industry standard internet service ai benchmark suite. *arXiv preprint arXiv:1908.08998*, 2019.

[64] Liljana Gavrilovska, Vladimir Atanasovski, and Valentin Rakovic. Graph theory based model for cognitive radio network topologies. In *2007 2nd International Conference on Cognitive Radio Oriented Wireless Networks and Communications*, pages 540–544. IEEE, 2007.

[65] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[66] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-gcn: A graph convolutional network accelerator with

runtime locality enhancement through islandization. In *MICRO-54: 54th annual IEEE/ACM international symposium on microarchitecture*, pages 1051–1063, 2021.

[67] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[68] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156–165, 2005.

[69] Bruce L Golden, S Raghavan, and Edward A Wasil. The vehicle routing problem: Latest advances and new challenges. *Operations Research*, 56(5):1–14, 2008.

[70] Xun Gong, Xiang Gong, Leiming Yu, and David Kaeli. Haws: Accelerating gpu wavefront execution through selective out-of-order execution. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(2):1–22, 2019.

[71] Xiangyang Gou, Chenxingyu Zhao, Tong Yang, Lei Zou, Yang Zhou, Yibo Yan, Xiaoming Li, and Bin Cui. Single hash: Use one hash function to build faster hash based data structures. In *2018 IEEE international conference on big data and smart computing (BigComp)*, pages 278–285. IEEE, 2018.

[72] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral. *arXiv preprint arXiv:2006.12138*, 2020.

[73] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[74] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[75] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):1–19, 2015.

[76] Mona Hashemi, Amirabbas Momeni, A Pashrashid, and Siamak Mohammadi. Graph centrality algorithms for hardware trojan detection at gate-level netlists. *International Journal of Engineering*, 35(7):1375–1387, 2022.

[77] Nikolaus Hautsch and Ruihong Huang. Algorithmic trading patterns in xetra orders. In *European Financial Management*. Wiley Online Library, 2012.

[78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[79] Matthias Hein, Jens Eisert, and Hans J. Briegel. Entanglement in graph states and its applications. *Quantum Information and Computation*, 6(4):381–466, 2006.

[80] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[81] Theus Hossmann, Thrasyvoulos Spyropoulos, and Franck Legendre. Know thy neighbor: Towards optimal mapping of contacts to social graphs for dtn routing. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[82] Can Hou, Jiaxin Chen, Yaqing Zhou, Lei Hua, Jinxia Yuan, Shu He, Yi Guo, Sheng Zhang, Qiaowei Jia, Chenhui Zhao, et al. The effectiveness of quarantine of wuhan city against the corona virus disease 2019 (covid-19): A well-mixed seir model analysis. *Journal of medical virology*, 92(7):841–848, 2020.

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

[84] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1029–1042. IEEE, 2022.

[85] Rotem Ben Hur and Shahar Kvatinsky. Memory processing unit for in-memory processing. In *2016 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 171–172. IEEE, 2016.

[86] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 42–55. IEEE, 2023.

[87] Kira Intrator and Kaustubh Shivdikar. Missing 'middle scenarios' uncovering nuanced conditions in latin america's housing crisis. *Cityscape*, 19(2):31–44, 2017.

[88] Vassilis N Ioannidis, Xiang Song, Saurav Manchanda, Mufei Li, Xiaoqin Pan, Da Zheng, Xia Ning, Xiangxiang Zeng, and George Karypis. Drkg-drug repurposing knowledge graph for covid-19, 2020.

[89] Malith Jayaweera, Kaustubh Shivdikar, Yanzhi Wang, and David Kaeli. Jaxed: Reverse engineering dnn architectures leveraging jit gemm libraries. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 189–202. IEEE, 2021.

[90] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition and applications. *arXiv preprint arXiv:2002.00388*, 2020.

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

[92] Yilun Jin, Guojie Song, and Chuan Shi. Gralsp: Graph neural networks with local structural patterns. In *Proceedings of the AAAI Conference on Artificial Intelligence*, number 04, pages 4361–4368, 2020.

[93] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. *ACM SIGPLAN Notices*, 48(4):395–406, 2013.

[94] Gilbert Jonatan, Haeyoon Cho, Hyojun Son, Xiangyu Wu, Neal Livesay, Evelio Mora, Kaustubh Shivdikar, José L Abellán, Ajay Joshi, David Kaeli, et al. Scalability limitations of processing-in-memory using real system evaluations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(1):1–28, 2024.

[95] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th*

Annual International Symposium on Computer Architecture (ISCA), pages 790–803. IEEE, 2020.

[96] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016.

[97] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *HPDC*, pages 239–250, 2014.

[98] Sumin Kim, Seunghwan Oh, and Youngmin Yi. Minimizing gpu kernel launch overhead in deep learning inference on mobile gpus. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 57–63, 2021.

[99] Haklin Kimm, Incheon Paik, and Hanke Kimm. Performance comparision of tpu, gpu, cpu on google colaboratory over distributed deep learning. In *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pages 312–319. IEEE, 2021.

[100] Kevin Kiningham, Philip Levis, and Christopher Ré. Grip: A graph neural network accelerator architecture. *IEEE Transactions on Computers*, 72(4):914–925, 2022.

[101] Kevin Kiningham, Christopher Re, and Philip Levis. Grip: A graph neural network acceleratorarchitecture. *arXiv preprint arXiv:2007.13828*, 2020.

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

[103] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[104] Vladimir Kiriansky and Nir Shavit. Avoiding algorithmic pitfalls for graph analytics with lightnvm. In *2016 International Conference on Supercomputing*, pages 4:1–4:13. ACM, 2016.

[105] Rik Koncel-Kedziorski, Dhanush Bekal, Yi Luan, Mirella Lapata, and Hannaneh Hajishirzi. Text generation from knowledge graphs with graph transformers. *arXiv preprint arXiv:1904.02342*, 2019.

[106] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76. IEEE, 2009.

[107] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale mlperf-0.6 models on google tpu-v3 pods. *arXiv preprint arXiv:1909.09756*, 2019.

[108] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231–247, 1992.

[109] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[110] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Xu Liu, Nathan Tallent, and Kevin Barker. Tartan: evaluating modern gpu interconnect via a multi-gpu benchmark suite. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 191–202. IEEE, 2018.

[111] Guohao Li, Matthias Müller, Ali Thabet, and Bernard Ghanem. Deepgcns: Can gcns go as deep as cnns? In *The IEEE International Conference on Computer Vision (ICCV)*, 2019.

[112] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788. IEEE, 2021.

[113] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[114] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[115] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 444–459. IEEE, 2022.

[116] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, et al. Comparison of the two major classes of assembly algorithms: overlap–layout–consensus and de-bruijn-graph. *Briefings in functional genomics*, 11(1):25–37, 2012.

[117] 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 (ASPLOS)*, ASPLOS '23, page 747–761, New York, NY, USA, 2023. Association for Computing Machinery.

[118] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 2020.

[119] Neal Livesay, Gilbert Jonatan, Evelio Mora, Kaustubh Shivdikar, Rashmi Agrawal, Ajay Joshi, José L. Abellán, John Kim, and David Kaeli. Accelerating finite field arithmetic for homomorphic encryption on gpus. *IEEE Micro*, 43(5):55–63, 2023.

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

[121] Jiageng Mao, Minzhe Niu, Chenhan Jiang, Hanxue Liang, Jingheng Chen, Xiaodan Liang, Yamin Li, Chaoqiang Ye, Wei Zhang, Zhenguo Li, et al. One million scenes for autonomous driving: Once dataset. *arXiv preprint arXiv:2106.11037*, 2021.

[122] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[123] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[124] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[125] Ryan R. McCune, Yue Chen, and Michael K. Reiter. Thinking in parallel for differential privacy. *ACM Transactions on Database Systems (TODS)*, 40(4):1–24, 2015.

[126] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[127] Saiful A Mojumder, Marcia S Louis, Yifan Sun, Amir Kavyan Ziabari, José L Abellán, John Kim, David Kaeli, and Ajay Joshi. Profiling dnn workloads on a volta-based dgx-1 system. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 122–133. IEEE, 2018.

[128] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[129] Francisco Muñoz Martínez, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. Stift: A spatio-temporal integrated folding tree for efficient reductions in flexible dnn accelerators. *J. Emerg. Technol. Comput. Syst.*, 19(4), sep 2023.

[130] Johannes Sebastian Mueller-Roemer, Christian Altenhofen, and André Stork. Ternary sparse matrix representation for volumetric mesh subdivision and processing on gpus. *Computer Graphics Forum*, 36(5):59–69, 2017.

[131] Giridhar Sreenivasa Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran, and Ponnuswamy Sadayappan. Optimal loop unrolling for gpgpu programs. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–11. IEEE, 2010.

[132] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th Inter-*

national Conference on Parallel Processing (ICPP), pages 101–110, Piscataway, NJ, 2017. IEEE.

[133] Attoor Sanju Nair, Jyh-Charn Liu, Laurence Rilett, and Saurabh Gupta. Non-linear analysis of traffic flow. In *ITSC 2001. 2001 IEEE Intelligent Transportation Systems. Proceedings (Cat. No. 01TH8585)*, pages 681–685. IEEE, 2001.

[134] Sharan Narang and Gregory Diamos. Deepbench, 2016.

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

[136] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. arXiv preprint arXiv:1906.00091, 2019.

[137] Wing WY Ng, Yueming Lv, Daniel S Yeung, and Patrick PK Chan. Two-phase mapping hashing. *Neurocomputing*, 151:1423–1429, 2015.

[138] Hanlin Niu, Yu Lu, Al Savvaris, and Antonios Tsourdos. Efficient path planning algorithms for unmanned surface vehicle. *IFAC-PapersOnLine*, 49(23):121–126, 2016.

[139] Tesla NVIDIA. V100 gpu architecture whitepaper, 2017.

[140] Jose Ordonez-Lucena, Pablo Ameigeiras, Diego R Lopez, Juan J Ramos-Munoz, Javier Lorca, and Jesus Folgueira. Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges. *IEEE Communications Magazine*, 55(5):80–87, 2017.

[141] Jayashree Padmanabhan and Melvin Jose Johnson Premkumar. Machine learning in automatic speech recognition: A survey. *IETE Technical Review*, 32(4):240–251, 2015.

[142] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736, Piscataway, NJ, 2018. IEEE.

[143] Shirui Pan, Ruiqi Hu, Guodong Long, Jing Jiang, Lina Yao, and Chengqi Zhang. Adversarially regularized graph autoencoder for graph embedding, 2018.

[144] Zhaoqing Pan, Weijie Yu, Xiaokai Yi, Asifullah Khan, Feng Yuan, and Yuhui Zheng. Recent progress on generative adversarial networks (gans): A survey. *IEEE Access*, 7:36322–36333, 2019.

[145] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, MD Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. Maxk-gnn: Towards theoretical speed limits for accelerating graph neural networks training. *arXiv preprint arXiv:2312.08656*, 2023.

[146] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[147] 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. IEEE, 2020.

[148] Sivasankaran Rajamanickam, Seher Acer, Luc Berger-Vergiat, Vinh Dang, Nathan Ellingwood, Evan Harvey, Brian Kelley, Christian R Trott, Jeremiah Wilke, and Ichitaro Yamazaki. Kokkos kernels: Performance portable sparse/dense linear algebra and graph kernels. *arXiv preprint arXiv:2103.11991*, 2021.

[149] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing dma engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 78–91. IEEE, 2018.

[150] ROCmSoftwarePlatform. Rocmsoftwareplatform/hipsparse: Rocm sparse marshalling library.

[151] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, Elliot Cooper-Balis, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[152] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[153] 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. IEEE, 2023.

[154] Vivek Seshadri, Onur Mutlu, Michael A Kozuch, and Todd C Mowry. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 355–366, 2012.

[155] Kaustubh Shivdikar. *SMASH: Sparse Matrix Atomic Scratchpad Hashing*. PhD thesis, Northeastern University, 2021.

[156] Kaustubh Shivdikar, Nicolas Bohm Agostini, Malith Jayaweera, Gilbert Jonatan, José L. Abellán, Ajay Joshi, John Kim, and David Kaeli. Neurachip: Accelerating gnn computations with a hash-based decoupled spatial accelerator. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024.

[157] Kaustubh Shivdikar, Nicolas Bohm Agostini, Malith Jayaweera, Gilbert Jonatan, Jose L Abellan, Ajay Joshi, John Kim, and David Kaeli. Neurachip: Accelerating gnn computations with a hash-based decoupled spatial accelerator. *arXiv preprint arXiv:2404.15510*, 2024.

[158] Kaustubh Shivdikar, Yuhui Bao, Rashmi Agrawal, Michael Shen, Gilbert Jonatan, Evelio Mora, Alexander Ingare, Neal Livesay, José L Abellán, John Kim, et al. Gme: Gpu-based microarchitectural extensions to accelerate homomorphic encryption. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 670–684, 2023.

[159] Kaustubh Shivdikar, Gilbert Jonatan, Evelio Mora, Neal Livesay, Rashmi Agrawal, Ajay Joshi, José L Abellán, John Kim, and David Kaeli. Accelerating polynomial multiplication for homomorphic encryption on gpus. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 61–72. IEEE, 2022.

BIBLIOGRAPHY

[160] Kaustubh Shivdikar, Ahan Kak, and Kshitij Marwah. Automatic image annotation using a hybrid engine. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–6. IEEE, 2015.

[161] Kaustubh Shivdikar, Kaushal Paneri, and David Kaeli. Speeding up dnns using hpl based fine-grained tiling for distributed multi-gpu training.

[162] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing*, pages 445–448, 2005.

[163] Aaron Smith. What people like and dislike about facebook, Jul 2020.

[164] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[165] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling betweenness centrality using communication-efficient sparse matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, New York, NY, 2017. ACM.

[166] Fenglong Song, Zhiyong Liu, Dongrui Fan, Junchao Zhang, Lei Yu, Nan Yuan, and Wei Lin. Design of new hash mapping functions. In *2009 Ninth IEEE International Conference on Computer and Information Technology*, volume 1, pages 45–50. IEEE, 2009.

[167] Olaf Sporns. Networks of the brain. *MIT Press*, 2011.

[168] Mohit Srinivasan, Ahan Kak, Kaustubh Shivdikar, and Chirag Warty. Dynamic power allocation using stackelberg game in a wireless sensor network. In *2016 IEEE Aerospace Conference*, pages 1–10, Piscataway, NJ, 2016. IEEE.

[169] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780, Piscataway, NJ, 2020. IEEE.

[170] Jacob R Stevens, Dipankar Das, Sasikanth Avancha, Bharat Kaul, and Anand Raghunathan. Gnnerator: A hardware/software framework for accelerating graph neural networks. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 955–960. IEEE, 2021.

[171] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127, 2012.

[172] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[173] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[174] Toyofumi Takenaka, Satosi Kato, and Hidetosi Okamoto. Adaptive load balancing content address hashing routing for reverse proxy servers. In *2004 IEEE International Conference on Communications (IEEE Cat. No. 04CH37577)*, volume 3, pages 1522–1526. IEEE, 2004.

[175] Yu-Hang Tang, Oguz Selvitopi, Doru Thom Popovici, and Aydın Buluç. A high-throughput solver for marginalized graph kernels on gpu. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 728–738. IEEE, 2020.

[176] Mohammad Khavari Tavana, Yifan Sun, Nicolas Bohm Agostini, and David Kaeli. Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-gpu systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 664–674. IEEE, 2019.

[177] Swadhin Thakkar, Kaustubh Shivdikar, and Chirag Warty. Video steganography using encrypted payload for satellite communication. In *2017 IEEE Aerospace Conference*, pages 1–11, Piscataway, NJ, 2017. IEEE.

[178] Roberto Todeschini and Viviana Consonni. *Handbook of Molecular Descriptors*. Wiley, 2016.

[179] Yuta Tokusashi, Huynh Tu Dang, Fernando Pedone, Robert Soulé, and Noa Zilberman. The case for in-network computing on demand. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[180] Nenad Trinajstic. *Chemical graph theory*. CRC Press, 1983.

[181] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[182] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. Nupar: A benchmark suite for modern gpu architectures. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, pages 253–264, 2015.

[183] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[184] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 372–383, 2019.

[185] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, Yajuan Wang, Endong Wang, Qing Zhang, Bo Shen, et al. Intel math kernel library. *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pages 167–188, 2014.

[186] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[187] Pin-Han Wang and Wei-Sheng Chou. Network topology design and bandwidth allocation for mpls/gmpls-based recovery. *Journal of Lightwave Technology*, 21(1):79–91, 2003.

[188] Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, volume 4, pages 670–676, 2004.

[189] Boris Weisfeiler and Andrei A Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

[190] Peter Willett. The calculation of molecular structural similarity: Principles and practice. *Molecular Informatics*, 25(2):127–136, 2006.

[191] Tak-Lam Wong, Yusheng Li, Zili Xu, and Jianer Chen. Frequent itemsets mining on big transactional data. *IEEE Transactions on Knowledge and Data Engineering*, 27(8):2261–2273, 2015.

[192] F. Y. Wu. Two-dimensional ising square lattices with a free boundary. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10020, 2004.

[193] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.

[194] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. An efficient compiler framework for cache bypassing on gpus. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 516–523. IEEE, 2013.

[195] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. Coordinated static and dynamic cache bypassing for gpus. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88. IEEE, 2015.

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

[197] Yuto Yamaguchi, Tsubasa Takahashi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. Turank: Twitter user ranking based on user-tweet graph analysis. In *Web Information Systems Engineering–WISE 2010: 11th International Conference, Hong Kong, China, December 12-14, 2010. Proceedings 11*, pages 240–253. Springer, 2010.

[198] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Characterizing and understanding gcns on gpu. *IEEE Computer Architecture Letters*, 19(1):22–25, 2020.

[199] 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. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[200] Jaewon Yang, Julian McAuley, and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Knowledge Discovery and Data Mining*, pages 745–754, 2012.

[201] Kai Yang and Mengqi Qi. Dynamic transportation routing using real-time data for logistics applications. *Transportation Research Part E: Logistics and Transportation Review*, 110:46–59, 2018.

[202] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, pages 40–48. PMLR, 2016.

[203] Pengcheng Yao, Long Zheng, Yu Huang, Qinggang Wang, Chuangyi Gui, Zhen Zeng, Xiaofei Liao, Hai Jin, and Jingling Xue. Scalagraph: A scalable accelerator for massively parallel graph processing. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 199–212. IEEE, 2022.

[204] Rozhin Yasaei, Shih-Yuan Yu, and Mohammad Abdullah Al Faruque. Gnn4tj: Graph neural networks for hardware trojan detection at register transfer level. In *2021 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pages 1504–1509. IEEE, 2021.

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

[206] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? *Advances in neural information processing systems*, 27, 2014.

[207] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE*

International Symposium on High-Performance Computer Architecture (HPCA), pages 460–474. IEEE, 2022.

[208] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[209] Eva Zangerle, Martin Pichl, Wolfgang Gassler, and Günther Specht. # nowplaying music dataset: Extracting listening behavior from twitter. In *Proceedings of the first international workshop on internet-scale multimedia management*, pages 21–26, 2014.

[210] 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)*, ASPLOS '21, page 687–701, New York, NY, USA, 2021. Association for Computing Machinery.

[211] Haowen Zhang, Yuandong Chan, Kaichao Fan, Bertil Schmidt, and Weiguo Liu. Fast and efficient short read mapping based on a succinct hash index. *BMC bioinformatics*, 19:1–14, 2018.

[212] 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)*, pages 261–274, Piscataway, NJ, 2020. IEEE.

[213] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. Architectural implications of graph neural networks. *IEEE Computer Architecture Letters*, 19(1):59–62, 2020.

[214] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[215] Da Zheng, Minjie Wang, Quan Gan, Zheng Zhang, and George Karypis. Learning graph neural networks with deep graph library. In *Companion Proceedings of the Web Conference 2020*, pages 305–306, 2020.

[216] Ruohuang Zheng and Sreepathi Pai. Efficient execution of graph algorithms on cpu with simd extensions. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 262–276. IEEE, 2021.

[217] Caiming Zhong, Duoqian Miao, and Ruizhi Wang. A graph-theoretical clustering method based on two rounds of minimum spanning trees. *Pattern Recognition*, 43(3):752–766, 2010.

[218] Dengyong Zhou, Bernhard Schoelkopf, and Thomas Hofmann. Graph embedding and extensions: A general framework for dimensionality reduction. *IEEE transactions on pattern analysis and machine intelligence*, 29(1):40–51, 2009.

[219] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[220] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100. IEEE, 2018.

[221] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.

Biography

Kaustubh Shivdikar was born in Mumbai, India, on December 5, 1994. He obtained his Bachelor of Science degree in Electrical Engineering from the Veermata Jijabai Technological Institute, University of Mumbai, in 2016. He went on to receive his Master of Science and Doctor of Philosophy degrees in Electrical and Computer Engineering from Northeastern University, Boston, USA, in May 2020 and May 2024, respectively. His Ph.D. research was supervised by Dr. David Kaeli at the Northeastern University Computer Architecture Research (NUCAR) Laboratory. Kaustubh is a member of IEEE and ACM. His research fields encompass Computer Architecture Simulator Design, Graph Neural Network Accelerators, Sparse Matrix Accelerators, and Homomorphic Encryption Accelerators.[1]

---

[1] https://wiki.kaustubh.us