Optimizing Emerging Applications Through Software Hardware Co-Design

by

Yuhan Chen

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2024

Yuhan Chen

chenyh@umich.edu

ORCID iD: 0000-0002-4835-8568

DEDICATION

Dedicated to my parents Jianjun Chen and Xiaojuan Chen.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Trevor Mudge, for his guidance and support throughout my PhD journey. He not only provides excellent research insights but also advice in work and life. It has been enjoyable to work with Prof. Trevor Mudge as I have the freedom to work on topics I want to, and in the meantime, he is also available to help when I hit obstacles.

I would also like to thank Dr. Nishil Talati and Prof. Ronald Dreslinksi for their generous help in all aspects of research, from choosing research directions and how to conduct research to paper writing. They selflessly helped me grow as a young researcher and, together with Prof. Trevor Mudge, gave me invaluable support throughout the years.

I would like to thank Prof. Zhengya Zhang for being my committee member. Thank you for taking your time and giving constructive advice on my proposal and thesis.

In addition, I would like to thank Prof. Hun-Seok Kim, Prof. David Blaauw, Prof. Scott Mahlke, Prof. Alex Bronstein, Prof. Krisztian Flautner, Prof. Baris Kasikci, Dr. Tanvir Ahmed Khan; I feel honored to have the opportunity to collaborate with you and learn from your experience.

I am also extremely lucky to have the opportunity to work with a group of the most talented and nice colleagues, fellow students, and friends. I enjoy the time brainstorming ideas, developing solutions, and writing papers with you. Specifically, I would like to thank Haojie Ye, Sanjay Sri Vallabh Singapuram, Yichao Yuan, Alireza Khadem, Hee Woo Kim, Armand Behroozi, Sanketh Vedula, Kuan-Yu Chen, Mason Nelson, Tarunesh Verma, Jingyuan Zhu, Yufeng Gu, Yunjie Pan, Shibo Chen, Yuchen Xia, Wynn Kaza, Xin He, Siying Feng, Yichen Yang, Xueyang Liu, Subhankar Pal, Aporva Amarnath, and many more. I had a great time working with you as well as having fun outside research with you; thank you so much.

Finally, I would like to express my deepest gratitude to my family. I would like to thank my parents Jianjun Chen and Xiaojuan Chen. They gave me endless support throughout my entire life and raised me to be an honest and responsible person. Without them, I would not have had the opportunity to pursue my PhD degree and become the person I am today. I would also thank my grandparents, Qizhi Chen, Hongmei He, Changchun Chen, and Haiju Qian, for their encouragement during my PhD years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

ABSTRACT

Emerging applications such as video transcoding and graph algorithms have seen fast development and broad adoption recently. It is crucial to improve the performance of these emerging applications for cost-efficiency and scalability. This thesis focuses on video transcoding and graph algorithms and uses software-hardware co-design to optimize their execution.

Video transcoding is rapidly growing as the demand for online streaming services continues to strive, and understanding the hardware bottleneck in performing video transcoding is the stepping stone to develop dedicated hardware for it.

Graph data structure is widely used in modeling complicated relationships between entities. Algorithms and applications that utilize the expressiveness of graphs are rapidly evolving and employed in various domains like social networks, chemistry, biology, and physics. With the expanding family of graph algorithms and the exploding size of real-world graphs, it is hard for hardware to keep up with the ever-growing demand for processing power for graph algorithms. To make the issue worse, the irregular memory access pattern in graph algorithms makes it hard to fully utilize traditional hardware like CPUs and GPUs.

In this thesis, I propose software and hardware co-design to improve the performance of emerging applications. At a high level, I first present hardware characterization that reveals the hardware bottlenecks with the change in software parameters. Then I benchmark the performance of the most popular graph sparsification algorithms on their performance in preserving graph properties. Finally, I propose a power-efficient accelerator supporting multiple dataflows for Graph Convolutional Networks.

Specifically, first, I perform CPU characterization on video transcoding, revealing the hardware bottlenecks (*e.g.* frontend, backend, branch misprediction, stalls) and how they shift with software parameters. Second, I use graph sparsification to tackle the exploding size of real-world graphs. I conduct a comprehensive benchmark on 12 graph sparsification algorithms, exploring their performance in preserving 16 essential graph properties on 14 real-world graphs, and give insights into how to choose the appropriate sparsification method for different down-stream tasks. Last, I present PEDAL, a power-efficient Graph Convolutional Network (GCN) accelerator designed to support multiple dataflows, achieving both high execution efficiency and flexibility.

CHAPTER 1

Introduction

1.1 Emerging Applications

An emerging application is an application that is new or has gained increasing attention in recent years. The hardware characteristics of emerging applications are often not well understood, and the execution is under-optimized. As they get more heavily deployed in life, it is increasingly important to optimize them for execution efficiency and enable them to scale further in the future. In this thesis, I will focus on video transcoding and graph algorithms as representative emerging applications.

1.2 Video Transcoding

Video transcoding decodes videos from the source format and encodes them to the desired format for distribution. The demand for online video streaming has rapidly increased in recent years, and it costs streaming providers billions of dollars to transcode these videos [7]. Understanding the hardware bottleneck with different transcoding setups can guide the development of dedicated hardware for transcoding, or better schedule different transcoding tasks to the appropriate hardware, potentially saving both transcoding time and hardware cost.

1.3 Graph Data Structure and Application

A graph is a data structure that models entity relationships using vertices and edges. The flexibility and expressiveness make graphs an ideal tool for representing complex relationships. For example, a graph can be used to describe a social network [64], where each vertex represents a user, and an edge between vertices represents interactions between users. More information can be embedded into a graph by assigning types to vertices and edges (Heterogeneous graphs [157, 142]) and associating each vertex and edge an embedding that includes extra information beyond the graph structure [149, 42, 69]. In the social network example, the graph can be made heterogeneous. Different vertex types can represent a company account or a personal account. Different edge types can represent various interactions like messaging, friending, following, etc. A vertex embedding can include extra information like the age and gender of a user.

Graphs are used in many algorithms and applications due to the expressiveness of graph data structure. Citation networks[111] and road networks [119] are similar to the social networks mentioned above, on which centrality-based algorithms like betweenness centrality can extract important vertices from the graph, and distance-based algorithms like All-Pair-Shorted-Path (APSP) and Dijkstra's algorithm [55] can find the shortest path between any two vertices. Page Rank [113] is another essential algorithm applied on web graphs to find highly relevant pages to keywords, which is the backbone of today's search engine. Graphs are also used in scientific computing; for example, chemical and biological networks are used to model protein structures [80, 79], and graph neural networks are used in practical physics to model their interactions [129]. Graph Neural Networks (GNNs) are an emerging application that builds neural networks that take graphs as input [85] and learn from the graph structure and embeddings to make predictions for vertices, edges, and graphs.

1.4 Motivation

To improve the performance of emerging applications like video transcoding and graph algorithms, one must understand the bottleneck first and then combine software and hardware solutions to achieve the best results. There are multiple obstacles to running graph algorithms and applications efficiently. One obstacle is the irregular memory access in the graph data structure, which makes retrieving data from memory a bottleneck. Another obstacle is the size of real-world graphs, which can be very large and require a lot of hardware resources and time to process.

The way graphs are accessed usually requires retrieving data from non-continuous regions in the memory, which makes it hard for caches to capture locality. This inherent irregular memory access leads to the under-utilization of computing resources because it's constantly blocked by data retrieval [34].

The real-world graph datasets often consist of billions of vertices and trillions of edges, making it hard for hardware to host the large amount of data. The overflowed data needs to travel down the memory hierarchy and be spilled to the main memory or even disks. This, together with the irregular memory access pattern, makes retrieving data even slower, leaving the computing resources even more underutilized.

These problems motivate the need for dedicated software and hardware solutions to speed up graph algorithms and applications. The efficient execution of graph algorithms is crucial to expand them to large-scale graphs and achieve lower latency.

1.5 Dissertation Contribution and Organization

The goal of this dissertation is to address the aforementioned problems both from the software angle and the hardware angle. At a high level, the contributions in this dissertation include CPU characterization of video transcoding, benchmarking of graph sparsification algorithms, and hardware Graph Convolutional Network accelerator. More specifically, the individual contributions of each of the works are detailed below.

- **CPU characterization on video transcoding (Chapter 3).** Characterizing hardware with changing software parameters helps understand where the hardware bottleneck is and facilitates future modification to the hardware. This work characterizes CPU when performing video transcoding with FFmpeg [1]. Although not directly related to graphs, this work reveals the bottleneck shifts between the CPU frontend and backend with the change of FFmpeg parameters and presets. The work focuses on two most important parameters: *crf* and *refs*, which control the transcoding rates and number of reference frames. This work found that increasing *crf* and *refs* will shift the bottleneck from the frontend to the backend, and the roofline model can explain the observation using operational intensity. The work also applied Graphite [117] and AutoFDO [46] and achieved an average speedup of 4.66%, and without change to the hardware, achieved 3.72% speedup by applying a custom scheduler with the understanding of hardware characterization. Understanding how software choices can change the bottleneck in hardware is helpful in the following and future works.

- **Demystifying graph sparsification (Chapter 4).** Graph sparsification is a general approach to reduce the amount of work in graph algorithms. I make the observation that most real-world graphs contain redundant information that contributes little to the final results. Pruning out a large portion of the graph will not significantly impact the quality of downstream tasks. However, different graph algorithms and applications depend on different graph properties, and the choice of sparsification algorithms is essential in preserving these graph properties. This work is the first to perform comprehensive benchmarking to reveal the relationship between sparsification algorithms and their perseverance on graph properties. This work evaluated 12 graph sparsification algorithms and analyzed 16 essential graph metrics on 14 real-world graphs with diverse characteristics, collecting more than 30,000 data points and providing insights into choosing the appropriate sparsification algorithm for the corresponding graph properties. The work also open-sourced the easily extendable framework to add more graph properties and sparsification algorithms for future research.

- **GCN accelerator (Chapter 5).** Graph Convolutional Network (GCN) is a type of Neural Network that takes graphs as input and learns from the graph structure and embeddings to predict vertices, edges, and graph properties. Due to graphs' inherent irregular memory access, GCNs often suffer from low execution efficiency. There are two phases called aggregation and combination in GCNs. The way of performing aggregation and the order of executing the two phases form multiple dataflows to perform GCN execution. Existing accelerators for GCN only support one dataflow when executing GCNs, which only achieves optimal efficiency with certain aggregation functions and input graphs. This work proposes a hardware accelerator that supports three different dataflows, accommodating linear and non-linear aggregation functions, small and large graphs, and sparse and dense vertices embeddings. It achieves 144.5×, 9.36×, and 2.55× speedup compared to CPU, GPU, and HyGCN, respectively, and 8856×, 1606×, 8.4× and 1.78× better power efficiency compared to CPU, GPU, HyGCN, and EnGN respectively. I also trained a decision tree with 400 synthetic datasets to automatically and accurately choose the best dataflow for a GCN algorithm and input graph. The decision tree chooses the best dataflow with 90% accuracy.

1.6 Impact Statement

The current and future impact of this dissertation work is summarized below.

- **CPU characterization on video transcoding.** This work reveals how software parameter choices will affect the hardware bottleneck. The characteristics learned can be used to guide the design of hardware accordingly to address the software needs. The work demonstrated some simple utilization of the hardware characteristics by applying

Graphite, AutoFDO, and a custom scheduler to achieve better performance. This work is included in the 2020 IEEE International Symposium on Workload Characterization.

- **Demystifying graph sparsification.** This work covers the most popular sparsification algorithms and evaluates their performance on the most widely-used graph metrics, then makes a comprehensive comparison and provides insights into how to select the best sparsification algorithm for the downstream tasks. This work also created a framework for evaluating sparsification algorithms. The framework is open-sourced and easily extendable, which facilitates future research expanding to more sparsification algorithms and graph metrics. The work is included in the 2024 International Conference on Very Large Data Bases.

- **GCN accelerator.** Prior accelerator only supports certain dataflow when executing GCNs, which either sacrifice efficiency or flexibility. This work is the first to propose a hardware accelerator that supports three different dataflows and automatically selects the best one corresponding to the aggregation function and the input graph characteristics. Besides, due to the design using much fewer processing elements, it achieves better power efficiency while still having comparable performance. This word is included in the 2023 Design, Automation and Test in Europe Conference and is nominated for the best paper award.

CHAPTER 2

Background

2.1 Video Transcoding

Video streaming is responsible for 82% of the Internet traffic in 2022 [4]. Video streaming service providers like YouTube and Netflix need to perform video transcoding before streaming the videos. Video transcoding is the process of decoding an encoded video into raw frames and re-encoding the frames into a video using the specified encoding format, frame size, frame rate, bit rate, etc. This accommodates end-users on various terminal devices with different network conditions and demands different video qualities. Video transcoding often has a wide range of requirements for the target video format, with various trade-offs between the transcoding time and transcoded video quality. These discrepancies stress the hardware in different ways and thus can have different bottlenecks in hardware.

2.2 Graph

A graph is a data structure consisting of vertices (also called nodes) and edges; the vertices usually represent entities, and the edges usually represent the relationship between entities. The edges can be both directed or undirected and weighted or unweighted. To define a graph formally: $\boldsymbol{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{w})$, where $\mathcal{V}$ and $\mathcal{E}$ denotes the set of vertices and edges in $\boldsymbol{G}$ respectively, and $\boldsymbol{w}$ denotes the weights of the edges. In a directed graph, each edge has a source and a destination vertex, while an undirected graph implies a bidirectional relationship.

In an unweighted graph, all edges have a default weight of 1. An adjacency matrix is used to represent a graph, denoted by $\boldsymbol{A}$, with the entries in $\boldsymbol{A}$ defined as:

$$\boldsymbol{A}_{ij} = \begin{cases} \boldsymbol{w}_{i \to j} & \text{if } e_{ij} \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

2.2.1 Graph Properties

Graph properties (also called metrics) are used to describe the graph in specific ways. For example, the degree distribution of a graph describes how skewed the distribution of the number of neighbors (a.k.a degree) in a graph is; graph diameter describes the furthermost distance between two vertices in a graph. These graph properties are utilized in graph algorithms and applications to extract information from the graph. For example, degree distribution may be used to distinguish a social network from a road network, as a social network is likely more skewed than a road network; graph diameter may be more related to distance-related algorithms like All Pair Shortest Path (APSP) and Single Source Shortest Path (SSSP). In this thesis, the essential graph properties are included and grouped into five groups. These graph properties are summarized as follows, and a more detailed description can be found in Chapter 4.1.2.

Basic Metrics describes the high-level information of a graph. *Degree Distribution* describes the skewness of edge distribution in a graph. *Laplacian Quadratic Form* is a fundamental quantity in graph theory [39], and it facilitates the analysis of various graph properties, including connectivity and spectral characteristics [32].

Distance Metrics describes the distance-related information of a graph. *All Pairs Shortest Path (APSP)* measures the minimum distance between any pair of vertices. Distance captures the proximity between two vertices. The *Diameter* of a graph is the maximum distance between any pair of vertices. *Vertex Eccentricity* is the length of the longest shortest path from a source vertex s to all other vertices. The minimum eccentricity is the graph **radius**, and the maximum eccentricity is the graph **diameter**.

Centrality Metrics is a set of metrics that measure the significance or ranking of vertices in a graph. *Betweenness centrality* suggests that vertices appearing on numerous shortest paths rank higher. *Closeness centrality* uses the average distance to all other reachable vertices to rank a vertex; the shorter the average distance is, the higher the ranking is. *Eigenvector centrality* measures the influence of a vertex [22]. A high eigenvector score means a vertex is connected to many vertices whose eigenvector scores are also high [110]. *Katz centrality* is a variant of *Eigenvector centrality*, it Katz centrality quantifies the influence of a vertex by considering the number of immediate neighbors and vertices connected to those immediate neighbors [82].

Clustering Metrics is closely related to grouping vertices into communities. *Number of communities* measure the degree of how scattered a graph is. k-means [98], agglomerative clustering [109], and DBSCAN [58] are often used to perform graph clustering. *Local Clustering Coefficient (LCC)* of a vertex v represents the proportion of pairs of neighbors of v that are connected. It evaluates the density of connections among the neighbors of a vertex [19]. *Global Clustering Coefficient (GCC)* [100] measures the fraction of closed triplets in all triplets. A triplet of nodes can consist of two (open) or three (closed) undirected edges [19]. *Clustering F1 score* assess the similarity between a given clustering and a reference clustering [103].

Application-level Metrics are ones directly used in applications. *PageRank* is designed to rank web pages [113]. The underlying concept suggests that pages linked by numerous important pages bear greater significance. *Min-cut and Max-flow* measure the smallest total weight of edges that disconnect the source vertex s from the sink vertex t, or the maximum amount of flow that can traverse from the source vertex s to the sink vertex t. *Graph Neural Networks (GNNs)* [127] are neural networks that operate on graphs. GNNs learn from the graph structure and embeddings and make classification and prediction on vertex, edge, or graph-level tasks [160, 92, 54].

2.3 Graph Algorithms

This section covers popular graph algorithms, grouped into traditional and emerging ones (specifically GNNs). The traditional graph algorithms are mainly used in Chapter 4, and GNNs are used in both Chapter 4 and Chapter 5.

2.3.1 Traditional Graph Algorithms

Traditional graph algorithms have existed for a long time and are widely used. Examples of such algorithms are Page Rank (PR), Connected Components (CC), Single Source Shortest Path (SSSP), Breadth First Search (BFS), etc. They are usually deterministic across runs, for example, SSSP will generate the same vertex distance given the graph is not changed, and PR may take different number of iterations to converge if initialized differently, but will have identical or very close results eventually. These traditional graph algorithms focus on different graph properties and reveal various aspects of the graph. SSSP focuses on distance-related graph properties, and PR focuses on the ranking of vertices in a graph. Chapter 4 utilized these graph algorithms to evaluate the effect of graph sparsification algorithms in preserving them.

2.3.2 Emerging Graph Algorithm: Graph Neural Networks (GNNs)

Emerging graph algorithms have drawn more attention in recent years. They are proposed to solve real-world problems. For example, graph mining [106, 139] finds certain motifs in a graph, which can be used for detecting certain behaviors or patterns like fraud detection. Graph Neural Networks (GNNs) is another emerging graph algorithm that takes graphs as input and uses neural networks to learn from the graph structure and embeddings. Then, the network is used to make classification or prediction on vertex level, edge level, or graph level tasks. In this thesis, GNN is used as a representative example of emerging graph algorithms. In Chapter 4, I evaluate the performance of different graph sparsification algorithms on GNNs. In Chapter 5, I propose PEDAL, a GCN accelerator to speed up GCN execution with high power efficiency.

2.4 Graph Sparsifition

Graph sparsification is a technique that approximates a given graph by a sparse graph with a subset of vertices and edges. An effective sparsification algorithm aims to maintain specific graph properties relevant to the downstream task while minimizing the graph's size. Graph algorithms often suffer from long execution time due to the irregularity and the large real-world graph size. Graph sparsification can significantly reduce the run time of graph algorithms by substituting the complete graph with a much smaller sparsified graph without significantly degrading the output quality. However, the interaction between numerous sparsifiers and graph properties is not widely explored, and the potential of graph sparsification is not fully understood. In this thesis, the 12 most representative graph sparsification algorithms are covered, and I evaluate their performance in maintaining 16 widely-used graph metrics on 14 real-world input graphs spanning various categories, characteristics, sizes, and densities. The graph sparsification algorithms are summarized as follows, and a more detailed description can be found in Chapter 4.1.3.

Random **sparsification** sparsify the graph by randomly sampling a subset of edges to keep in the sparsified graph. The edges are selected with equal probability. ***K-Neighbor*** **sparsification** selects k edges for each vertex, and if a vertex has less than k vertices, all of its edges are included. The edges are selected with probability proportional to their weights (uniform for unweighted graphs). ***Rank Degree*** **sparsification** starts from seed vertices, then ranks neighbors according to their degree in descending order. The edges connecting each seed vertex to its top-ranked neighbors are selected and incorporated into the sparsified graph. This process is repeated on newly added vertices to expand the graph. ***Local Degree* sparsification** is similar to the *Rank Degree* sparsification as it preserves edges incident to high-degree vertices. For each vertex, *Local Degree* incorporates edges to the top $deg(v)^{\alpha}$ neighbors ranked by their degree in descending order, where $\alpha \in [0, 1]$ controls the degree of sparsification. ***Spanning Forest*** is a subgraph that consists of multiple spanning trees with a minimal number of edges. Kruskal's algorithm [87] and Prim's algorithm [118] can be used to construct a *Spanning Forest*. ***t-Spanner*** is a family of subgraphs that approximates the pairwise distances between vertices in the original graph. A *t-Spanner* is a subgraph such that any pairwise distance is at most t times the distance in the original graph. ***Forest Fire*** sparsification model constructs the graph by adding one vertex at a time and forming edges to specific subsets of the existing vertices. Subsequently, it "spreads" from v to other vertices in the graph with a certain predefined probability, creating edges between v and the newly discovered vertices. This process assembles "burning" through edges probabilistically, hence the name *Forest Fire* [90]. **Similarity-based sparsifiers** Similarity-based sparsifiers constitute a group of sparsification algorithms based on similarities between vertices measured by specific metrics. **Global Sparsification** selects edges based on similarity scores globally. ***G-Spar*** sorts the Jaccard scores globally and selects the edges with the highest similarity score. ***SCAN*** [150] uses structural similarity measures to detect clusters, hubs, and outliers. **Local Sparsification** selected edges based on similarity scores locally. The ***L-Spar*** [125] includes edges with the highest Jaccard scores incident to each vertex locally. ***Local Similarity*** sparsification works similarly to *L-Spar*, but it further ranks edges using the Jaccard score and computes $log(rank(edge))/log(deg(v))$ as the similarity score. **Effective Resistance (*ER*) Sparsification** is derived from the analogy of an electrical circuit and applied to a graph. In this context, edges represent resistors, and the effective resistance of an edge corresponds to the potential difference generated when a unit current is introduced at one end of the edge and withdrawn from the other. Once the effective resistance is calculated, a sparsified subgraph can be constructed by selecting edges with a probability proportional to their effective resistances.

2.5 Hardware Characterization

Hardware characterization is crucial in understanding the hardware bottleneck for specific algorithms. This understanding is essential in making hardware accelerator design decisions. Many tools and methodologies are available to perform hardware characterization. Linux Perf [10] is a profiling tool using CPU performance counters and various enhancements. Intel Vtune [8] is a multi-platform profiling tool that is based on Performance Monitoring Unit (PMU) counters and incorporates techniques like Event-Based Sampling (EBS). For hardware characteristics that are hard to profile using only performance counters and to evaluate the performance of custom hardware, simulation is widely used. Event-based hardware like the Sniper The Sniper multi-core simulator [43] trades off a lower accuracy for a higher simulation speed. Cycle-accurate simulators like gem5 [36] simulate accurately what happens at each cycle; they are more accurate but take very long to run. Akram *et al.* put together a survey of different simulators [25]. In this thesis, I use Linux perf, Intel Vtune, and sniper in Chapter 3, and in Chapter 5, I implement a cycle-accurate simulator to evaluate the performance of the proposed GCN accelerator.

CHAPTER 3

CPU Microarchitectural Performance Characterization of Cloud Video Transcoding

Video streaming services are becoming increasingly popular, taking up a considerable portion of Internet traffic today. According to the Cisco Visual Networking Index report [4], video streaming took up 75% of the Internet traffic in 2017 and will take up 82% of the Internet traffic in 2022. Besides video streaming, online gaming that also uses video traffic is rising rapidly and is expected to grow 15 times by 2022. Figure 3.1 shows the Internet traffic from 2017 to 2022.

Video streaming service providers (*e.g.*, YouTube, Netflix, and Facebook) transfer (upload and download) only encoded videos to reduce video size and corresponding Internet traffic. In most use cases, the uploaded video format differs from the distributed video format as the video distribution must support a wide variation in network bandwidth, screen resolution, and user preferences [99]. Consequently, streaming service providers apply a large number of video transcoding—the process of decoding an encoded video into raw frames and re-encoding those frames in a different encoding format [148]—operations. Therefore, performance optimization of video transcoding workloads can save millions of dollars in computational and energy costs.

The performance implications of video transcoding have inspired a rich set of prior works. Existing works have compared the performance of different transcoding algorithms [52, 104] and variation in transcoding performance for different videos [99]. While these works fill

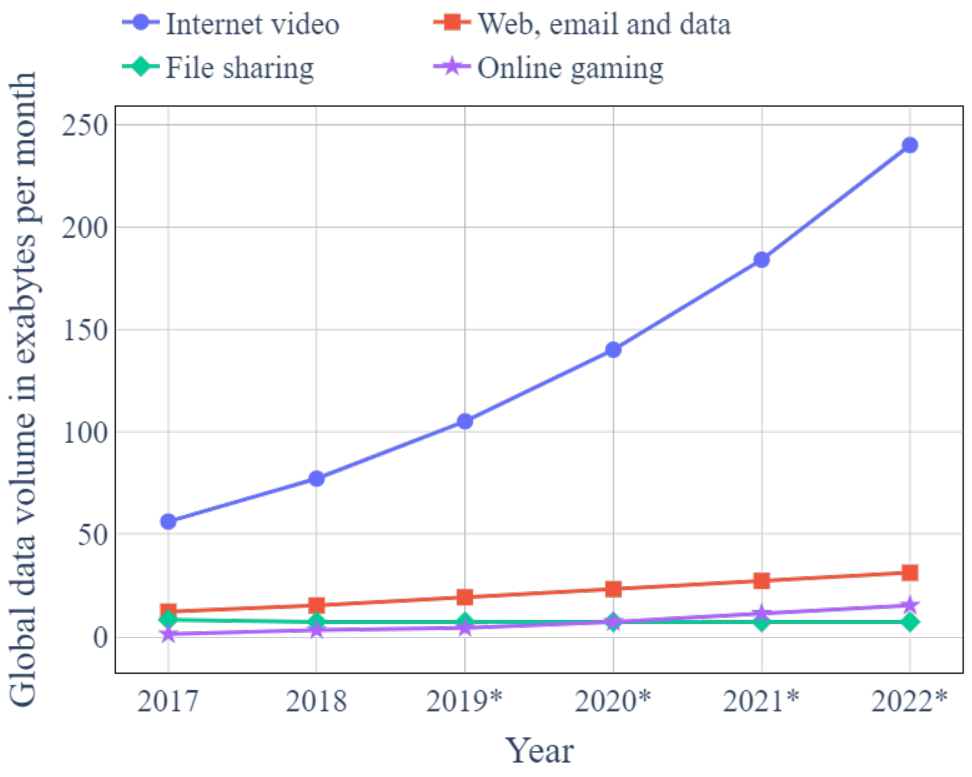


Figure 3.1: Data volume of global consumer internet traffic from 2017 to 2022, by subsegment (in exabytes per month) [4]. The trend shows a rapid growth of video traffic both in absolute and relative terms.

some gaps in understanding video transcoding workloads, several open questions exist in CPU microarchitectural bottleneck identification for these workloads. In this work, I aim to answer these questions by studying the microarchitectural characteristics of video transcoding operations in response to variations in different transcoding parameters and inputs.

For performance characterization of video transcoding workloads, I utilize a wide range of CPU hardware performance counters using Intel VTune [8] and Linux perf [10]. Specifically, I leverage the Top-down Microarchitecture Analysis Method [153] to identify bottlenecks in the CPU microarchitecture for different video transcoding operations. Based on this methodology, I investigate the performance of the leading video transcoding software, FFmpeg. FFmpeg offers many options to balance between transcoding speed, transcoded video quality, and transcoded file size. I examine how different values of these parameters affect microarchitectural performance issues for video transcoding workloads.

The intrinsic complexity of videos also affects transcoding performance. Videos with high motion and frequent scene transitions are of higher complexity, and they require longer transcoding time and a larger file size under the same quality constraint. *vbench* [99] is a benchmark developed for cloud video services. It uses clustering techniques to select 15 videos to cover a significant cross-section of a corpus of millions of videos. I use *vbench* to show how different video complexity affects video transcoding performance.

In this work, I make the following contributions:

- I identify key performance bottlenecks in CPU microarchitecture for various video transcoding workloads. Specifically, I observe that instruction cache, data cache, and branch prediction units suffer from frequent inefficiency for video transcoding operations. Moreover, microarchitectural performance issues change rapidly due to variations in transcoding options and video complexity.

- I leverage the state-of-the-art profile-guided optimization technique (AutoFDO [46]) to improve instruction cache and branch prediction performance of video transcoding workloads. I also apply a polyhedral optimizer (Graphite [117]) to improve the data cache performance of video transcoding operations. AutoFDO provides a 4.66% average speedup, while Graphite provides a 4.42% average speedup across workloads.

- I design a scheduler that assigns different video transcoding tasks to processors with varying configurations of microarchitecture based on transcoding parameters and inputs. In a simple case study, the designed scheduler performs 3.72% better than the random scheduler and matches the performance of the best scheduler 75% of the time.

The rest of the paper is organized as follows: I provide the background of video transcoding in §3.1. I describe the experimental methodology in §3.2. §3.3 reports experimental evaluation results. I briefly summarize the related works in §3.4. Finally, I conclude in §3.5.

3.1 Background

Streaming videos have several important properties. A series of raw image frames constitute a video, and the number of pixels in each frame is defined as video resolution. For example, a full high-definition (Full-HD) video contains 1920×1080 pixels per frame and is also known as a 1080p video. The number of frames for each second of video is defined as frame rate and expressed in frames per second or *FPS*. Streaming service providers support videos of different frame rates (24-60 *FPS*). Without compression, a single-second standard video (with 1080p resolution and 30 *FPS* frame rate) requires 178 MB of space [14]. Videos are encoded in several standard formats to reduce this high storage and network transmission cost.

3.1.1 Video Transcoding

Video transcoding is the process of converting one encoding format to another, and it is necessary because streaming service end-users have different requirements in terms of video resolution, frame rate, and encoding format based on their device capability and network condition. Today, more than 500 hours of videos are uploaded to YouTube every minute [7]. Since each uploaded video must be transcoded at least once [99], streaming service providers perform many video transcoding operations. Moreover, the cost of performing video transcoding is expensive. For instance, Amazon Elastic Transcoder charges 0.03$ to transcode a single-minute video clip [2]. At this rate, transcoding 500 hours of videos will require around 1800$.

Video transcoding is performed in two stages: (1) an encoded video is decoded into raw frames, and (2) these frames are encoded again in a different format. The decoding stage is deterministic and, hence, relatively straightforward. On the other hand, the encoding stage is much more complex as it models the video compression problem as a heuristic-driven search space exploration problem. Moreover, the encoding stage has two primary components: (1) *Intra-frame encoding* compresses pixels within a single frame by eliminating spatial redundancy, and (2) *Inter-frame encoding* compresses pixels across different frames by eliminating temporal redundancy. The intra-frame encoding divides a frame into several macroblocks [78]. On the other hand, the inter-frame encoding categorizes each frame as I (Intra-coded), B (Bidirectional predicted), or P(Predicted) picture frame [13].

FFmpeg is the leading video transcoding framework that can perform a wide range of operations (*e.g.*, decoding, encoding, transcoding, filtering, multiplexing, etc.) for different video encoding formats [1]. Since FFmpeg is the most widely used video transcoding software, I specifically focus on FFmpeg workloads. FFmpeg is typically compiled with x264, an open-source library developed by VideoLAN that implements state-of-the-art video encoding algorithms [16].

3.1.2 x264 Encoder

x264 is the state-of-the-art video encoder [16]. x264 achieves high performance with its rate control, motion estimation, macroblock mode decision, and quantization algorithms. The details of the algorithms are out of the scope of this work, but I describe what each of them does as I focus on how different algorithm parameters affect the transcoding performance.

3.1.2.1 Rate Control

Rate control is the mechanism to impose a constraint on bitrate or quality. It can be performed at three different granularities: at a coarse-grained level for a group of pictures (GOP), for a single picture, and at a fine-grained level for macroblocks. There are mainly six rate control modes: constant QP (CQP) controls the amount of quantization; average bitrate (ABR) tries to achieve the target average bitrate; 2-pass average bitrate (2-Pass ABR) is similar to ABR except it runs twice as the first pass provides a better estimation for the second pass encoding; constant bitrate (CBR) imposes a constant bitrate; constant rate factor (CRF) controls the quality rather than the bitrate, and constrained encoding (VBV) constrains the bitrate to a certain maximum. Among the six modes, only CBR is applied at the granularity of a macroblock. Other modes are applied at the granularity of the picture.

3.1.2.2 Motion Estimation

Motion estimation is the most complex and time-consuming component of the x264 encoding process. It detects the motion of objects (*e.g.*, translation, rotation, and tilting), encodes only the motion information, and thus saves space by not storing the entire frame. x264 provides four integer-pixel motion estimation methods: diamond (dia), hexagon (hex), uneven multi-hexagon (umh), and exhaustive (esa). Each mode represents a different search pattern, each more complex and time-consuming than the previous, but generates better motion estimation.

3.1.2.3 Macroblock Mode Decision

When encoding, each frame is partitioned into 16×16 macroblocks, which can be further partitioned into smaller blocks. An I-frame can only have I-macroblocks because it must not depend on other frames to decode, a P-frame can have both I-macroblocks and P-macroblocks, a B-frame can have I-macroblocks, P-macroblocks, and B-macroblocks.

3.1.2.4 Quantization

After motion estimation and macroblock mode decision, the residue between the original frame and prediction frame is computed. The x264 encoder uses trellis quantization [145, 105] to improve the storage efficiency of the residue. Users can select one of three levels of trellis quantization provided by the x264 encoder.

3.1.3 CPU vs. GPU

Videos can be transcoded in both CPUs and GPUs [5]. Typically, GPUs are faster than CPUs in terms of video transcoding time. However, GPUs perform worse than CPUs in terms of video compression ratio and quality. Hence, GPUs are leveraged to transcode only live-streamed videos where transcoding speed matters more than the transcoded video size or quality. Moreover, video transcoding in GPUs is relatively new and supports only a subset of video formats [99]. In practice, GPUs are used only as a hardware accelerator instead of the primary transcoder [26]. Therefore, in this work, I focus on video transcoding in CPUs.

3.1.4 Video Selection

Randomly selected videos could lead to biased and unrepresentative profiling results. In this work, I use videos from *vbench* benchmark suite [99]. The videos from *vbench* benchmark suite are representative of cloud transcoding workloads. *vbench* uses clustering techniques to select 15 videos of 5 seconds each from a corpus of millions of videos [99], and therefore is diverse and representative of real videos. I study the microarchitectural characteristics of the video transcoding operation for all *vbench* videos. I also use a video called Big_Buck_Bunny [3], widely studied in prior works [89, 97]. I list the detailed information of videos in Table 3.1. *vbench* also introduces a new video property, entropy, to represent the complexity of a video. This property specifies the number of bits required to encode a video with the visually lossless quality [99]. A higher entropy suggests the video is more complex, for example, involves more motion or frequent scene transition and thus requires more computing resources and a higher bitrate.

3.2 Methodology

Hardware platforms. I use a 4-core 3.5GHz Intel Xeon E3 CPU (NUMA with 1 socket). The memory hierarchy of the machine consists of 64KB of private L1-cache (32KB private instruction and 32KB private data), 256KB of private L2 cache, 8MB of shared L3 cache, and 16GB of RAM.

Software platforms. All experiments are conducted in Ubuntu 16.04 (Linux kernel version 4.15.0) using GCC version 5.5.0, ffmpeg version N-82144-g940b890, and x264 version 148-r2762-90a61ec.

Table 3.1: *vbench* videos info

Full Name	Short Name	Resolution	FPS	Entropy
desktop_1280x720_30.mkv	desktop	720p	30	0.2
presentation_1920x1080_25.mkv	presentation	1080p	25	0.2
bike_1280x720_29.mkv	bike	720p	29	0.9
funny_1920x1080_30.mkv	funny	1080p	30	2.5
cricket_1280x720_30.mkv	cricket	720p	30	3.4
house_1920x1080_30.mkv	house	1080p	30	3.6
game1_1920x1080_60.mkv	game1	1080p	60	4.6
game2_1280x720_30.mkv	game2	720p	30	4.9
girl_1280x720_30.mkv	girl	720p	30	5.9
chicken_3840x2160_30.mkv	chicken	2160p	30	5.9
game3_1280x720_59.mkv	game3	720p	59	6.1
cat_854x480_29.mkv	cat	480p	29	6.8
holi_854x480_30.mkv	holi	480p	30	7
landscape_1920x1080_29.mkv	landscape	1080p	29	7.2
hall_1920x1080_29.mkv	hall	1080p	29	7.7

3.2.1 Transcoding Metrics and Parameters.

Video transcoding workloads maintain a unique trade-off among three key performance metrics: (1) transcoding speed (measured by transcoding time in seconds), (2) transcoded video quality (measured by Peak Signal to Noise Ratio [PSNR] in decibels [dB]), and (3) transcoded video file size (measured by bitrate in Kbps or Mbps). FFmpeg, in combination with x264, provides many encoding options to balance among these three performance metrics. Among all such options, the most critical parameters are *crf* and *refs* [15, 12], and therefore, I investigate the microarchitectural performance implications of video transcoding in response to variation in these two parameters. Figure 3.2 shows how these parameters (*crf* and *refs*) affect key transcoding metrics (speed, quality, and size).

crf actively controls the transcoded video quality. An increase in *crf* value results in video quality degradation after encoding. In x264 encoding, *crf* can be varied from 0 to 51. Videos encoded with *crf* 0 are lossless, while videos encoded with *crf* 51 are the worst quality. x264 uses 23 as the default value for *crf*. *crf* also passively impacts transcoding speed and transcoded file size. An increase in *crf* value results in faster transcoding time and smaller transcoded file size.

On the other hand, *refs* directly controls the transcoded video file size. *refs* (Reference frame number) specifies how many reference frames will be used during inter-frame encoding in addition to the frame immediately prior to the current frame [9]. In x264 encoding, *refs* can be varied from 1 to 16. An increase in *refs* value expands the encoding search space, improves the compression possibility, and hence reduces transcoded video file size. However, increasing the *refs* value also slows down the transcoding process due to larger search space exploration. *refs* has no impact on transcoded video quality.

In addition to *crf* and *refs*, I also study the performance impact of different x264 presets (a combination of standard values for all transcoding parameters) that vary other transcoding

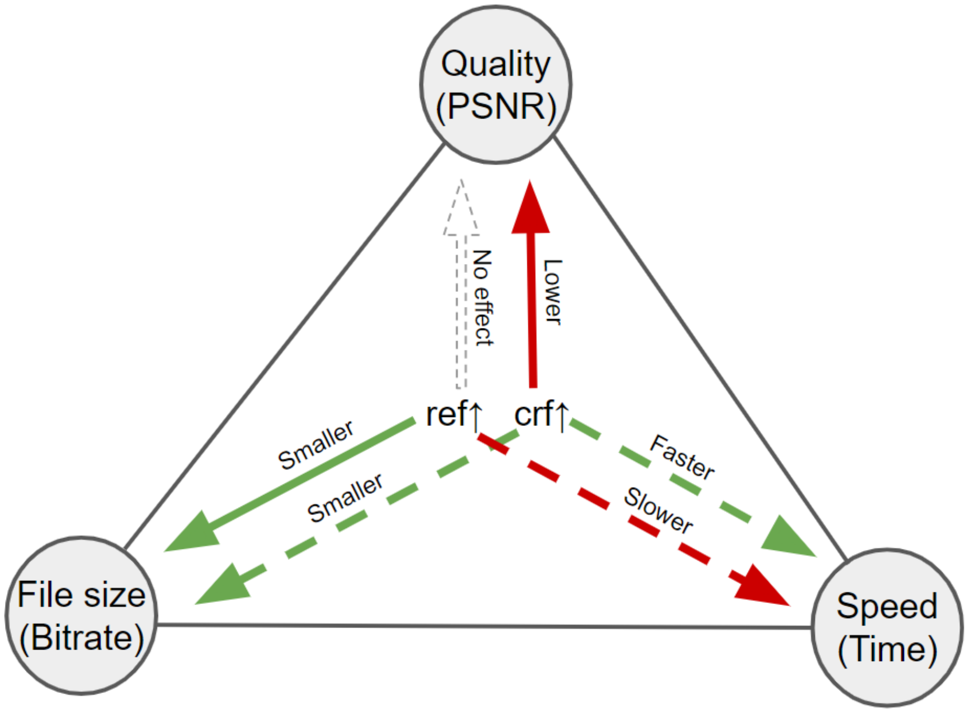


Figure 3.2: Transcoding speed, video quality, and file size triangle. It shows the effects of increasing *crf* and *refs* on the three metrics. A green line denotes a positive impact, a red line represents a negative impact, a solid line denotes an active impact (purpose of changing the option), and a dotted line indicates a passive impact (side effect).

options including motion estimation, macroblock mode decision, quantization, and frame type decision.

3.2.2 Tools

3.2.2.1 VTune

The Intel VTune profiler [8] is a performance analysis tool that leverages a large number of hardware performance counters provided by Intel Performance Monitoring Unit (PMU) [11]. Specifically, VTune uses the Top-down microarchitecture analysis method [153] to identify performance bottlenecks for CPU workloads. In Top-down methodology, performance issues are categorized into four major categories—retiring, bad speculation, front-end bound, and back-end bound—measured in the percentage of pipeline slots. A pipeline slot represents hardware resources needed to process one micro-operation (μOp). Ideally, the pipeline slots should be filled with instructions and successfully retire, but limited resources or bad speculations can lead to wasted pipeline slots.

Front-end bound pipeline slots are unused due to issues like instruction cache misses and instruction decoder unavailability. On the other hand, back-end bound slots are unused because of problems including data cache misses (memory bound) and computational unit shortage (core bound). Bad speculation issues are mainly due to branch mispredictions. Finally, retired slots denote properly utilized pipeline slots.

I leverage VTune to understand how different parameters and video workloads affect microarchitectural performance problems during transcoding. Particularly, I investigate how front-end bound, bad speculation and back-end bound issues are affected by different transcoding settings. Moreover, I use VTune to determine the root cause of performance problems.

3.2.2.2 Linux perf

Linux perf [10] provides a simple command-line interface to profile CPU executions. I leverage perf mainly to reveal more fine-grained details such as L1, L2, L3, and branch misses per kilo instructions (MPKI).

3.2.2.3 AutoFDO

AutoFDO [46] is the state-of-the-art feedback-directed optimization (FDO) tool. AutoFDO captures the frequently-taken branches and optimizes their layout to reduce instruction cache misses and branch mispredictions.

3.2.2.4 Graphite

Graphite [117] is a polyhedral analysis and optimization tool for GCC. It uses the polyhedral model to optimize nested loops, where optimizations like loop tiling and loop fusion can be applied to enable better cache locality. I use graphite to reduce back-end stalls during the transcoding operation by improving L1, L2, and L3 cache hit rates.

Table 3.2: Selection of the important options for different presets, adapted from [6]

Option	ultrafast	superfast	veryfast	faster	fast	medium	slow	slower	veryslow	placebo
aq-mode	0*	1	1	1	1	1	1	1	1	1
b-adapt	0*	1	1	1	1	1	1	2*	2*	2*
bframes	0*	3	3	3	3	3	3	3	8*	16*
deblock	[0:0]*	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]	[1:0]
me	dia*	dia*	hex	hex	hex	hex	hex	umh*	umh*	tesa*
merange	16	16	16	16	16	16	16	16	24*	24*
partitions	none*	+i8x8,+i4x4*	-p4x4	-p4x4	-p4x4	-p4x4	-p4x4	all*	all*	all*
refs	1*	1*	1*	2*	2*	3	5*	8*	16*	16*
scenecut	0*	40	40	40	40	40	40	40	40	40
subme	0*	1*	2*	4*	6*	7	8*	9*	10*	11*
trellis	0*	0*	0*	1	1	1	2*	2*	2*	2*

* value differ from medium (default) preset

3.2.2.5 Sniper

Sniper [43] is an open-source x86 simulator that can accurately simulate CPU executions with high speed. Moreover, Sniper allows modifying different microarchitecture parameters to study the performance impact of varying different processor configurations. I utilize Sniper to simulate the proposed scheduling algorithm with multiple μarch configurations.

3.2.3 Profiling Setup

3.2.3.1 Across *crf* & *refs*

I vary different transcoding parameters (*crf* from 1-51 and *refs* from 1-16) and investigate 816 different combinations for a single video, and study the profiling results. I use VTune to capture the high-level profiling results grouped into four categories and then use perf to get fine-grained results.

3.2.3.2 Across Presets

The x264 encoder provides ten predefined setups (presets) to vary different transcoding parameters for different usage scenarios [6]. I list parameter values for these presets in Table 3.2. All presets also specify a *crf* and *refs* number. I investigate the performance impact of *crf* and *refs* separately; I use the default *crf* (23) and *refs* (3) values for different presets. I investigate the performance impact of different presets for a single video.

3.2.3.3 Across Videos

Finally, I study the performance of transcoding for a wide range of videos from the *vbench* benchmark suite with the parameters *crf* =23, *refs* =3, and x264 preset being medium.

3.2.4 Optimization Setup

Implementing new optimizations for video transcoding is not the primary focus of this work. Instead, I study the impact of several optimizations to show the potential for improvement.

3.2.4.1 AutoFDO & Graphite

I optimize the video transcoding operation using AutoFDO to avoid instruction cache misses (grouped under front-end issues in Top-down methodology) and branch mispredictions (grouped under bad speculation issues in Top-down methodology). To apply AutoFDO, I use the FFmpeg program to transcode multiple videos and collect execution profiles using perf during transcoding. Then, I optimize FFmpeg by recompiling the program with the collected profile.

I optimized the video transcoding operation using Graphite to reduce data cache misses (grouped under back-end issues in Top-down methodology). Graphite is integrated into GCC and can be directly used by enabling specific optimization flags (`-floop-interchange -ftree-loop-distribution -floop-block`) during compilation. I enable those optimization flags during the compilation of the FFmpeg program.

3.2.4.2 Smart Scheduler

Streaming service providers may have transcoding servers with different μarch configurations. Even without optimizing the algorithm or implementation, knowing how to intelligently assign tasks to the server that best fits the task can fully utilize the resources and save transcoding time. The profiling results can be used as a reference to schedule transcoding tasks to the fitting server.

I consider four transcoding tasks, each with different video, *crf*, *refs*, and preset combinations, as shown in Table 3.3. I also modify the baseline μarch configuration of the Sniper simulator (gainestown) to create 4 μarch configurations. Different μarch configurations are optimized to reduce different types of pipeline issues by varying different microarchitectural resources. Table 3.4 describes the baseline and four modified configurations. I use different strategies to assign tasks to different μarch configurations (servers) and use the Sniper simulator to measure the transcoding time.

Table 3.3: Transcoding parameters used for Sniper simulation

Task#	Video	*crf*	*refs*	Preset
1	desktop	30	8	veryfast
2	holi	10	1	slow
3	presentation	35	6	veryfast
4	game2	15	2	medium

Table 3.4: Different microarchitectural configurations for Sniper simulation. The baseline is the default configuration provided by Sniper, Gainestown. fe_op is optimized to reduce front-end stalls with larger L1i-cache and iTLB. be_op1 and be_op2 are optimized to reduce back-end stalls by increasing the capacity of data caches and other pipeline resources. bs_op is optimized to avoid bad speculation stalls by replacing the default pentium_m branch predictor with the Tage branch predictor.

Config Name	L1d	L1i	L2	L3	L4	itlb	ROB	RS	issue at dispatch	branch predictor
baseline	32K	32K	256K	8M	None	128	128	36	No	Pentium_m
fe_op	-	64K	-	-	-	256	-	-	-	-
be_op1	64K	-	512K	4M	16M	-	-	-	-	-
be_op2	-	-	-	-	-	-	256	72	Yes	-
bs_op	-	-	-	-	-	-	-	-	-	Tage

- means same as baseline

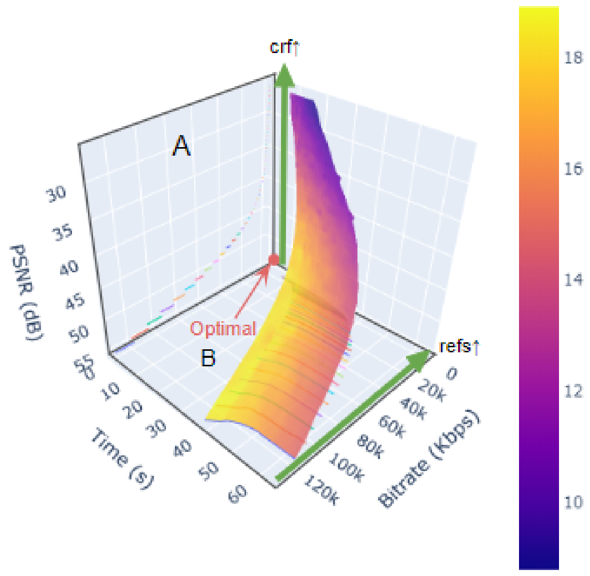
I evaluate three different schedulers. The random scheduler randomly assigns tasks among servers, so I use the average value of all four servers as its performance. The smart scheduler assigns tasks to the best-fit server under the constraint that the four tasks must be assigned to different servers (one-to-one constraint), preventing any server from over-utilizing or under-utilizing. Finally, the best scheduler assigns tasks to the best-fit server without the one-to-one constraint.

3.3 Evaluation Results

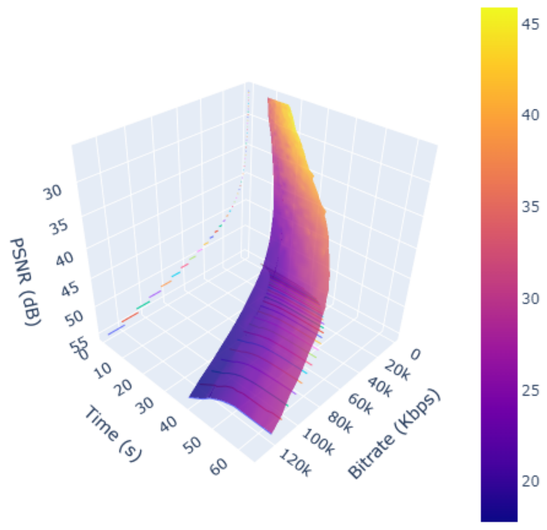
3.3.1 Profiling

3.3.1.1 Across *crf* & *refs*

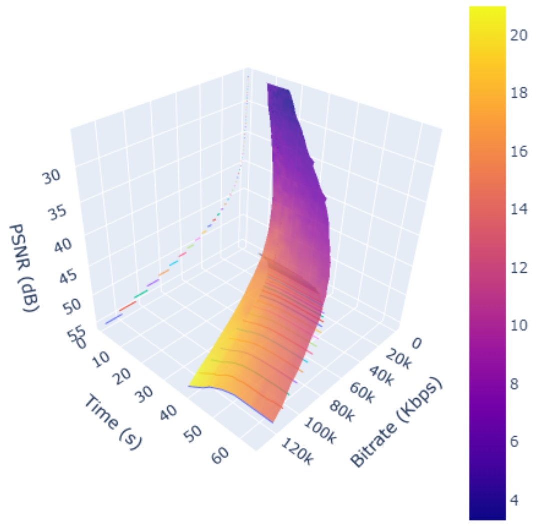
Figure 3.3 shows the heatmaps of 816 combinations where *crf* is varied from 1 to 51 and *refs* is varied from 1 to 16. The projections of each data point to the three axes represent video quality (PSNR), file size (bitrate), and transcoding speed (time), respectively, and the color represents either front-end, back-end, or bad speculation bound percentage in terms of pipeline slots. All three heatmaps are of the same shape but represent different bounds. As shown in Figure 3.3, both increasing *crf* and *refs* reduce front-end and bad speculation bound slots but increase back-end bound slots.



(a) Front-end Bound



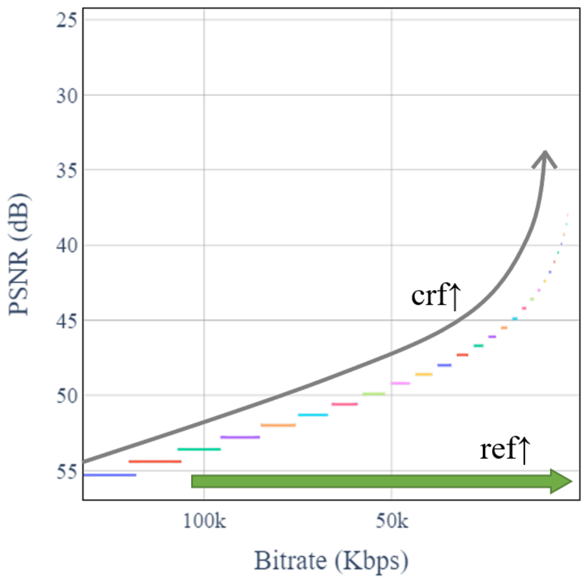
(b) Back-end Bound



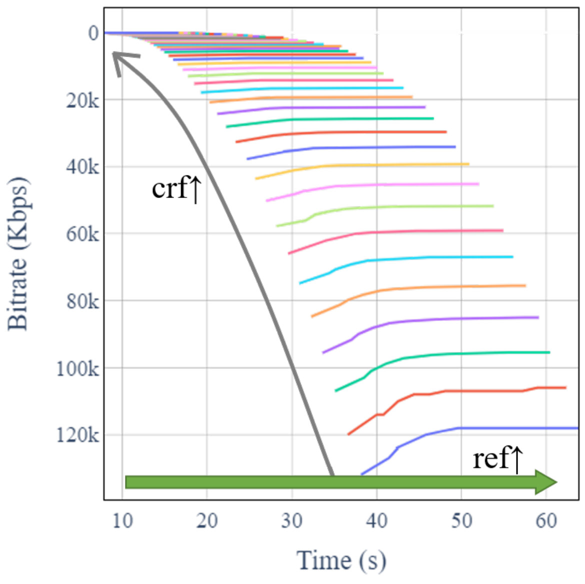
(c) Bad speculation bound

Figure 3.3: Heatmaps of front-end, back-end, and bad speculation bound pipeline slots (%)

Figure 3.4 shows two projections into planes A and B from Figure 3.3. Projection A has 51 horizontal lines for 51 discrete PSNR values. Each line represents one *crf* value as *crf* controls the video quality. With *crf* fixed, increasing *refs* can help save file size. Each horizontal line's



(a) Projection A



(b) Projection B

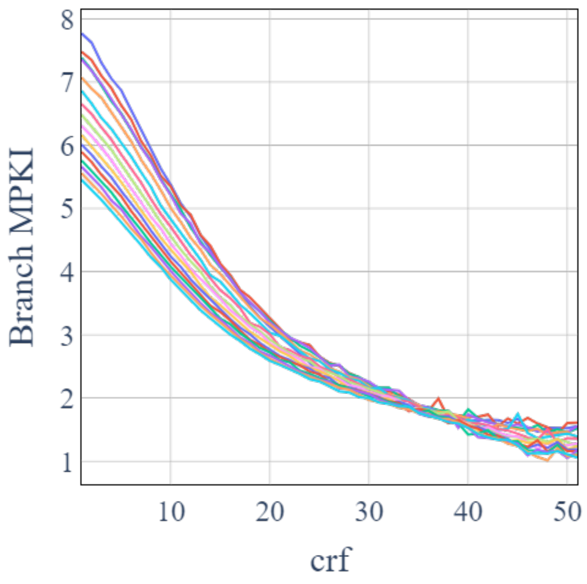
Figure 3.4: Projection A & projection B

length shows the bitrate range when increasing *refs* from 1 to 16. The longer the horizontal line is, the more it can benefit from increasing the *refs* value. With *crf* increasing, PSNR decreases, meaning the video quality deteriorates. Also, with *crf* increasing, the line length decreases, denoting a diminishing return for increasing *refs*.

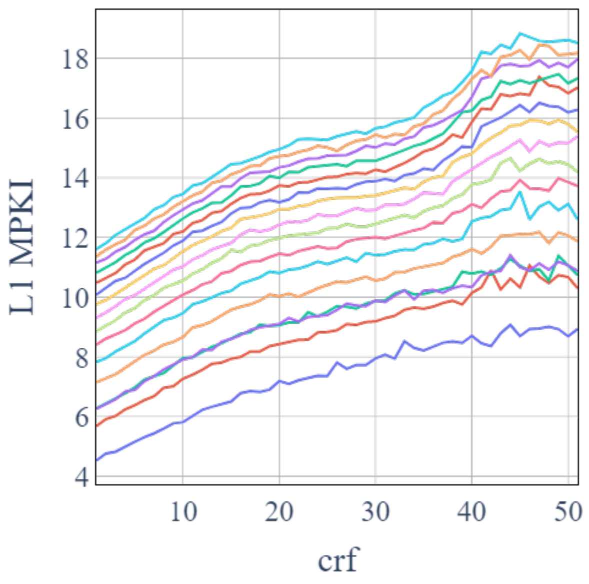
Projection B is the relation between time and *refs*, where increasing *refs* does not linearly decrease the file size. For each *crf*, there is an elbow point beyond which increasing *refs* has little or no return. Moreover, increasing *crf* makes the line flatter, meaning high *crf* benefits less from increasing *refs*, which aligns with the conclusion from projection A.

The main takeaways from the three heatmaps and two projections are: low *crf* benefits more from increasing *refs*, and increasing *refs* has diminishing returns. The result is video-dependent, and the elbow points can differ for different videos, but the trend shown is universally applicable.

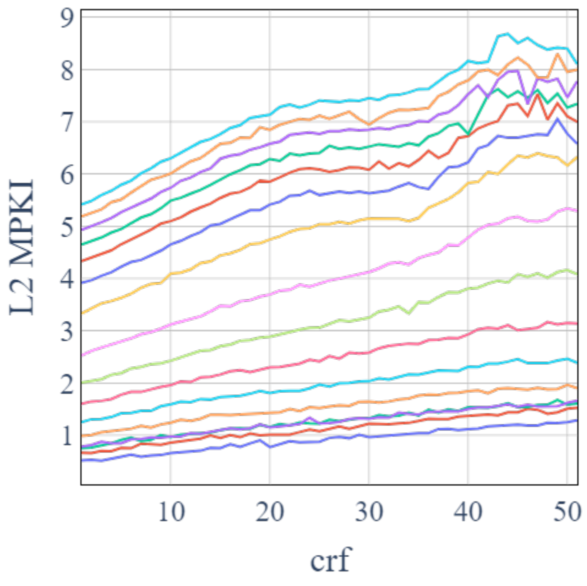
Further analysis shows the front-end bound slots are primarily due to the inefficiency in micro-instruction translation engine (MITE), and decoded stream buffer (DSB), both



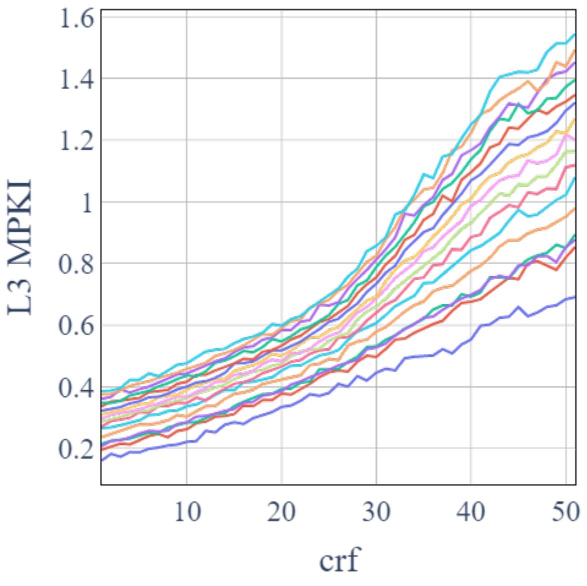
(a) Branch



(b) L1 cache



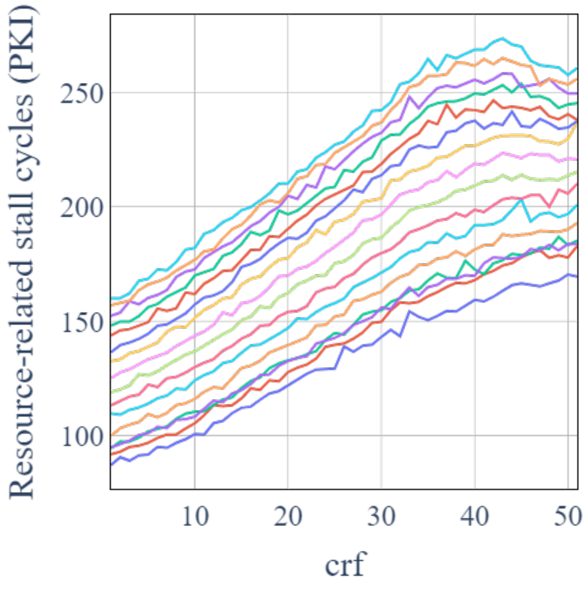
(c) L2 cache



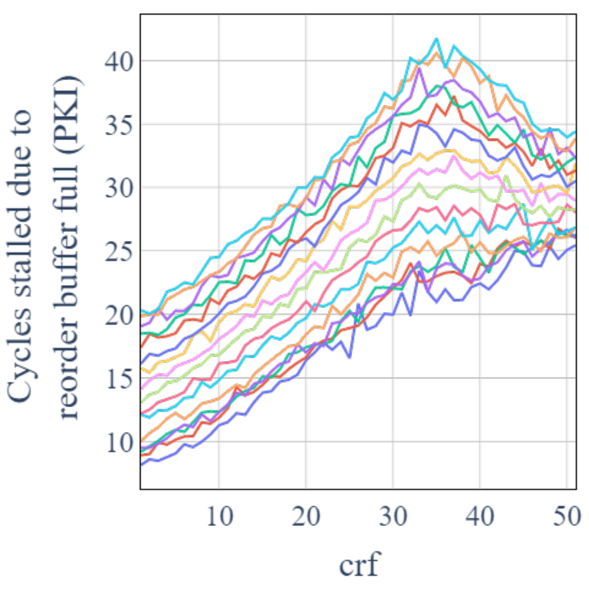
(d) L3 cache

Figure 3.5: Branch prediction and cache miss performance for different values of transcoding parameters, *crf* and *refs*.

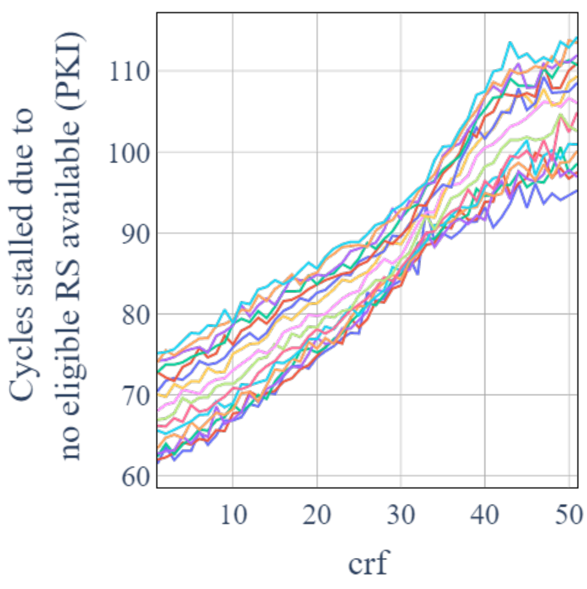
related to decoding instructions (instruction to micro-op conversion). Front-end bound slots represent only a small fraction of overall pipeline slots and do not change significantly for different *crf* and *refs* combinations. Back-end issues are responsible for most of the wasted



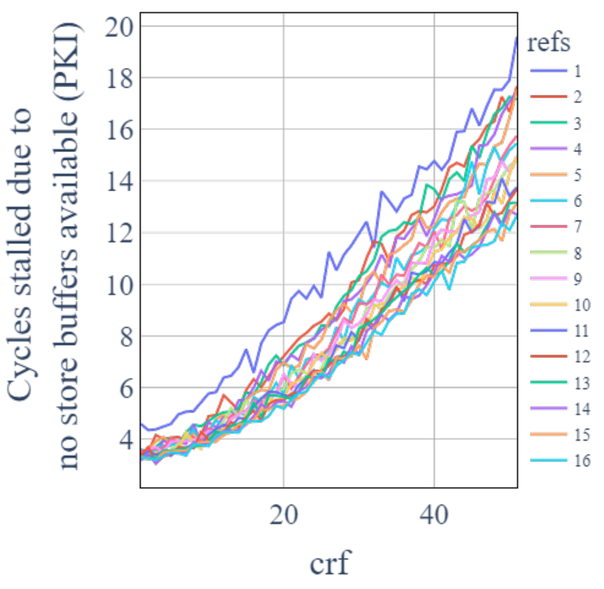
(a) Resource - Any



(b) Resource - ROB



(c) Resource - RS



(d) Resource - SB

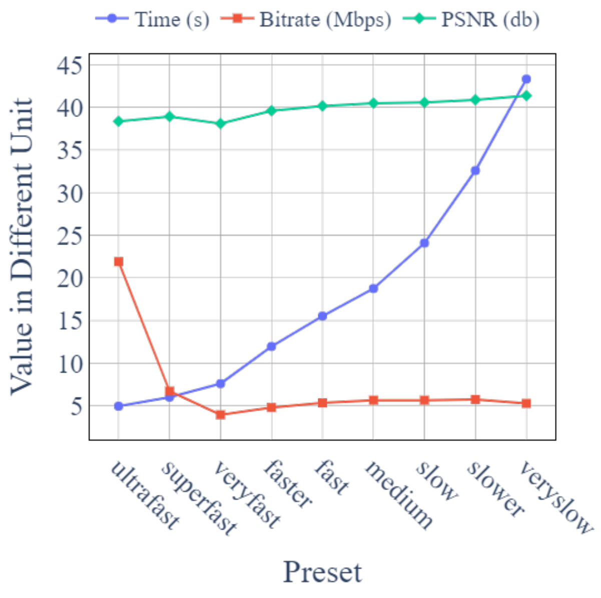
Figure 3.6: Stalls due to microarchitecture resources for different values of transcoding parameters, *crf* and *refs*.

pipeline slots. The back-end bound issues can be further divided into memory-bound and core-bound problems. Memory-bound slots mean the pipeline is stalled because the required data is unavailable. Core bound means the pipeline is stalled because the hardware resources (functional units) needed to perform operations are unavailable. In the evaluation, Bad speculation bound slots are almost always due to branch mispredictions.

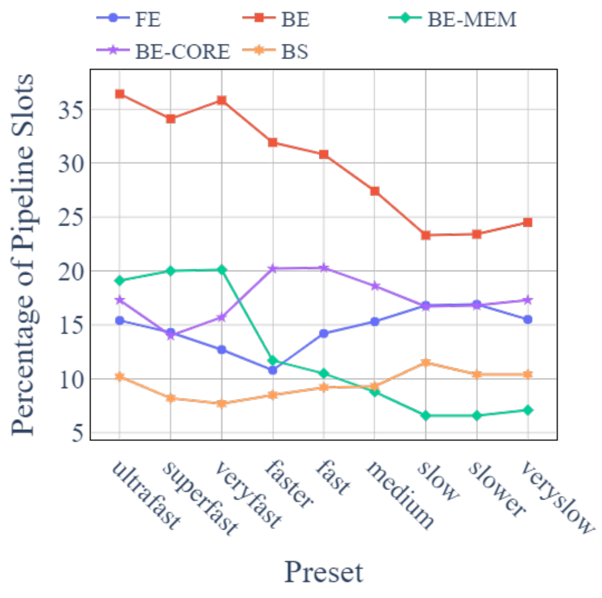
To further investigate these wasted pipeline slots, I evaluate the inefficiency of several microarchitectural resources. Specifically, I study the variation in eight hardware performance events in response to changes in *crf* and *refs*. Figure 3.5 and figure 3.6 show the results. Figure 3.5a shows that branch mispredictions per kilo instructions decreases when both *crf* and *refs* increase. Figure 3.5b, 3.5c, and 3.5d depicts misses per kilo instructions (MPKI) for L1, L2, and L3 data caches respectively. These cache misses are mainly responsible for the memory-bound component within the back-end bound slots. Figure 3.6a, 3.6b, 3.6c, and 3.6d denotes inefficiency in pipeline execution units and constitute the core bound component within the back-end bound slot. These inefficiencies show a similar trend of deteriorating when either *crf* or *refs* increase. Here, store buffer (SB) efficiency is a notable exception as the number of stalls due to unavailable store buffer decreases when *refs* increases.

The trend shown in both memory bound and core bound issues can be explained using the roofline model [147], a performance model that correlates performance with operational intensity. The roofline model defines operational intensity as how much computation is performed for each byte of DRAM traffic. For low operational intensity, CPU performs little arithmetic operation on each piece of data, and the workload is bound by memory. As operational intensity increases, the utilization of the CPU and the overall performance increase. The workload becomes compute-bound when the operational intensity is high enough to occupy all CPU resources.

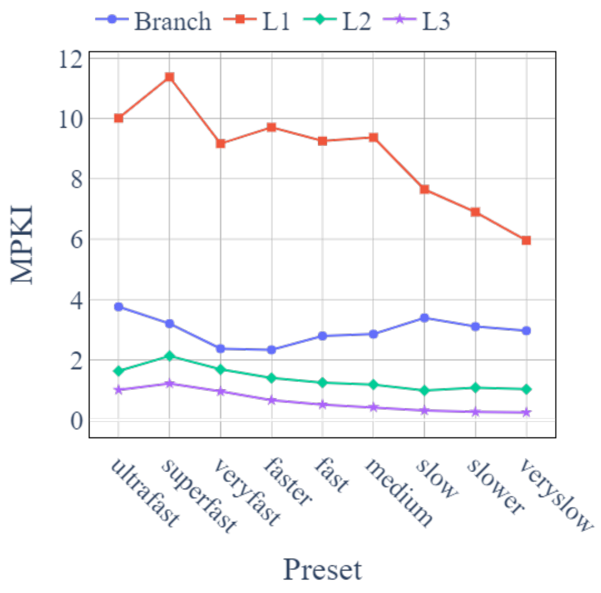
Increasing *crf* relaxes the quality constraint and requires less computation for the same amount of data transfer, thus causing a lower operational intensity. On the other hand, increasing *refs* increases the total number of executed instructions and memory traffic, but more on memory traffic, thus lowering the operational intensity. For lower operational intensity, processors have limited computations to hide the memory latency, resulting in



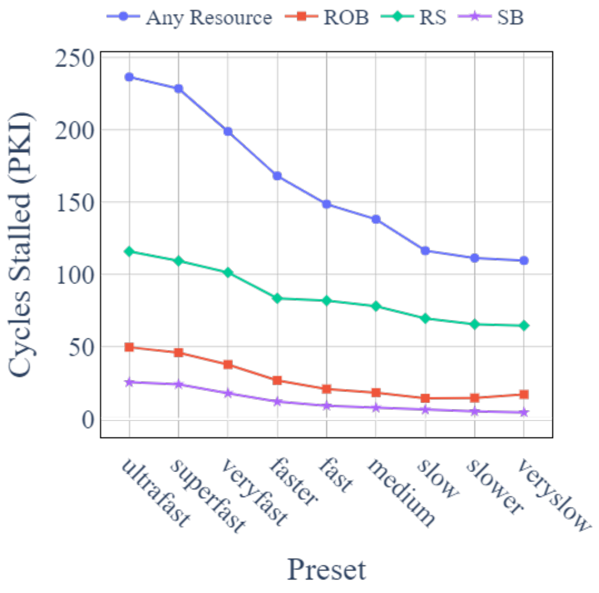
(a) Time, Bitrate, PSNR



(b) FE, BE, BS



(c) Branch, Cache MPKI

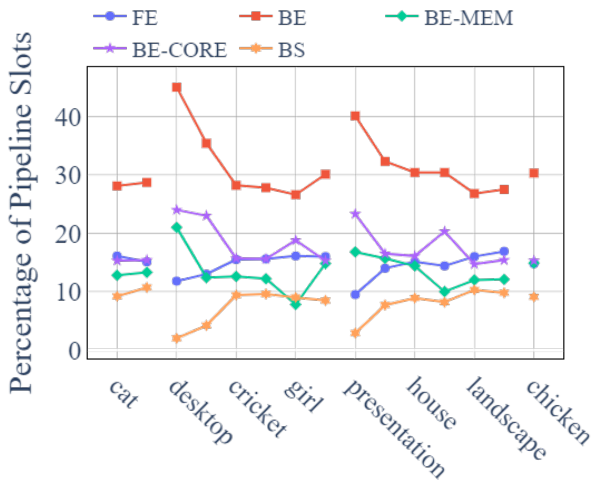


(d) Resources

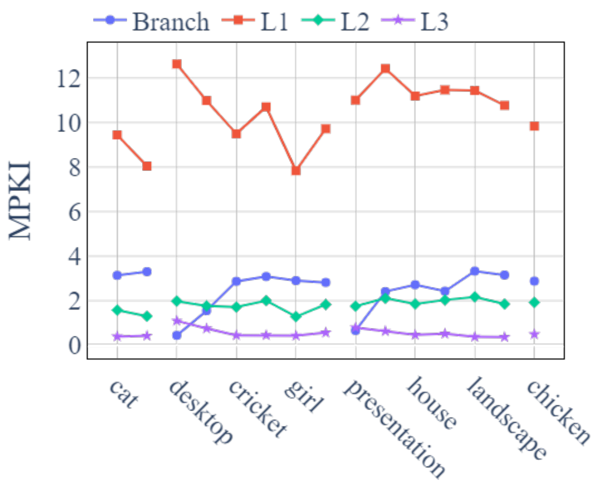
Figure 3.7: Profiling results for different transcoding presets

higher memory-bound stalls.

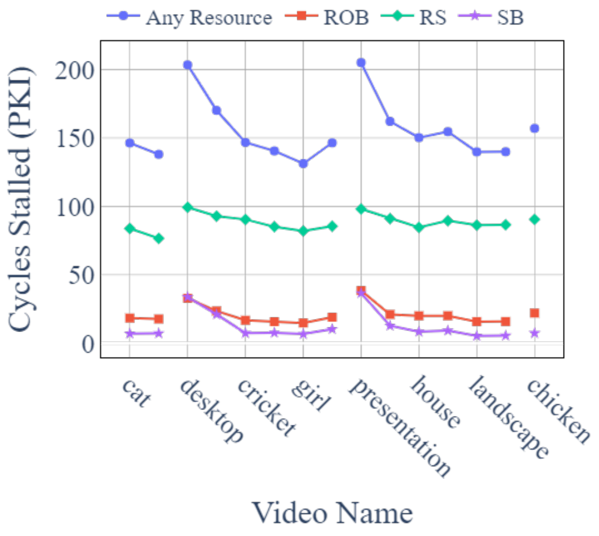
The roofline model can also explain the lower amount of front-end bound slots. As the CPU is waiting for memory traffic, it exhausts the non-arithmetic resources (*e.g.*, Reorder



(a) FE, BE, BS



(b) Branch, Cache MPKI



(c) Resources

Figure 3.8: Profiling results for different videos

buffer [ROB], reservation stations [RS], and store buffer [SB]) quickly. Consequently, the CPU stops fetching new instructions and has fewer front-end bound stalls. Note that SB stalls show different trends compared to ROB and RS stalls with a change in *refs*. That is because, higher *refs* results in better video compression which requires less number of total store operations.

3.3.1.2 Across Presets

Figure 3.7a shows how transcoding time, bitrate, and PSNR change for different transcoding presets. Similarly, Figure 3.7b shows the percentage of front-end, back-end, and bad speculation bound slots (%) for different transcoding presets. From the fastest to the slowest preset, transcoding time increases as expected. As *crf* is fixed, PSNR has a minor increase while bitrate shows excellent improvement from ultrafast to superfast, and superfast to veryfast, and then shows diminishing or even no returns with any slower presets. The trend of transcoding time and bitrate suggests that without any strict time constraint, tuning up the preset to veryfast can trade a small increment in transcoding time for file size reduction. Figure 3.7c shows that the branch MPKI fluctuates with no clear direction. Data cache MPKI goes down while using a slower preset. The trend agrees with Figure 3.7b, where only back-end issues have a clear trend of going down. This is mainly because the memory-bound component decreases. Figure 3.7d shows stalls due to resource unavailability, which can also be explained with the roofline model [147]. A slower preset has a higher operational intensity, thus it is less likely to run into memory-bound issues. Consequently, fewer instructions block ROB, RS and SB waiting for memory.

3.3.1.3 Across Videos

I now investigate the variation in microarchitectural characteristics while transcoding different videos. I first group videos based on different resolutions and then sort them based on different entropy [99]. The gaps in Figure 3.8 separate different resolution groups. As Figure 3.8a shows, with increased video entropy, front-end and bad speculation bound slots increase, and back-end bound slots decrease. Figure 3.8b and 3.8c show the variation of branch misprediction, memory bound, and core bound slots for different videos. As branch mispredictions dominate the bad speculation issues for video transcoding workloads, branch MPKI and slots lost due to bad speculation follow a similar trend. L1, L2, and L3 data cache MPKI follow the same trend as the memory-bound slots. Similarly, stalls due to other pipeline resources follow the same trend as the core-bound slots. The roofline model can also be applied here. Videos with higher entropy are more complex and need higher operational intensity to encode under the same quality constraint, leading to lower back-end bound issues.

3.3.2 Optimization

3.3.2.1 AutoFDO & Graphite

I optimize FFmpeg with AutoFDO and Graphite to reduce front-end, bad speculation, and back-end bound stalls while transcoding different videos. Figure 3.9 shows the results. AutoFDO provides an average speedup of 4.66%, with a maximum of 5.2%. On the other hand, Graphite provides an average improvement of 4.42%, with a maximum of 4.87%.

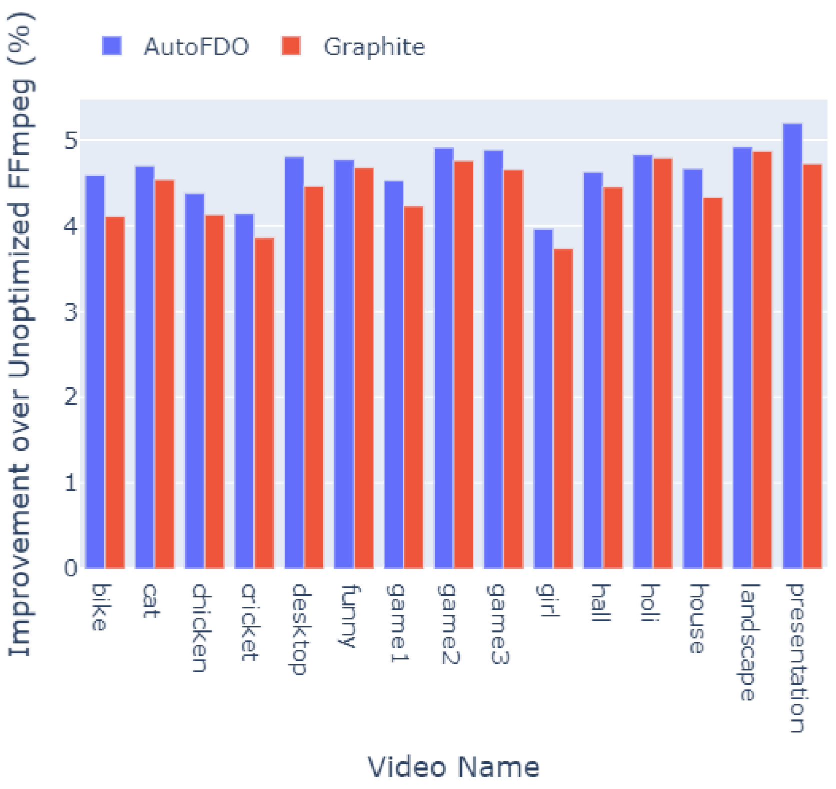


Figure 3.9: Speedup provided by AutoFDO-optimized FFmpeg binary and Graphite-optimized FFmpeg binary. The number is the average of 32 combinations of transcoding parameters (*crf*, *refs*, and presets).

3.3.2.2 Smart Scheduler

Figure 3.10 shows the speedup provided by three schedulers over the default configuration. All four μarch configurations have better microarchitectural resources than the default baseline, so all schedulers show performance gain. However, on average, the characterization-driven smart scheduler outperforms the random scheduler by 3.72%. Moreover, the smart scheduler provides the same schedule as the best-fit server in three out of four cases.

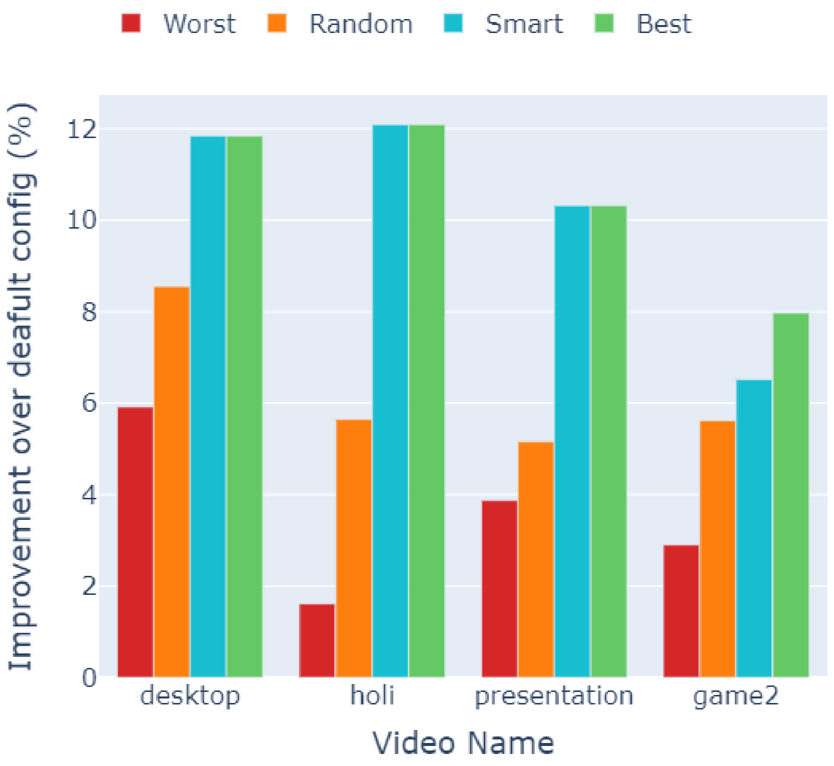


Figure 3.10: Transcoding speedup over the baseline μarch configuration. The random scheduler uses the average improvement of four modified μarch configurations. The one-to-one constraint is imposed on the smart scheduler but not on the best scheduler.

3.4 Related Work

The performance of video transcoding significantly impacts computational and energy savings. Realizing this significance, a rich set of prior works has investigated video transcoding performance. I describe related works in four categories.

Performance profiling of video transcoding. Different prior works have investigated video transcoding performance from different perspectives. For example, COVT [144] measures the transcoding time and compression ratio for many transcoding presets and video types and uses the results for efficient resource allocation. Other works [77, 49] aims to predict video transcoding workloads' power consumption. In comparison, I focus on microarchitectural bottlenecks while transcoding various videos with different parameter values.

Algorithmic optimization. Many prior works have examined the algorithmic optimization of video transcoding operations. For example, parallel transcoding on the Cloud [30, 96] optimizes for parallelism, and DCT transcoder [95] optimizes for fast DCT-domain transcoding. Sung et al. [135] propose a method to utilize the quadtree information from the decoding process to accelerate the encoding process. Zhang et al. [158] observe that the video background barely changes for certain types of videos and utilize this observation to achieve fast transcoding. In comparison, I notice the microarchitectural resource inefficiency during video transcoding and leveraged state-of-the-art compiler optimizations to improve the hardware resource utilization.

System/architectural optimization. Several prior works have designed efficient systems for video transcoding. Specifically, [56, 163] optimizes the storage efficiency while transcoding videos in content delivery networks (CDNs). GPU-accelerated VTU for MEC [26] leverages GPUs to accelerate the transcoding operation. Cloud Transcoder [93] bridges the gap between internet videos and mobile devices by offloading the bulk of the transcoding operation from mobile devices to the cloud. The characterization of video transcoding provides several insights into performance bottlenecks. These insights can be leveraged to design an efficient video transcoding system in the future, as I have shown with the smart scheduler experiment.

Adaptive video streaming. Adaptive video streaming services tune video transcoding parameters to generate videos of different quality [134, 29, 81]. The values of these parameters are predicted based on the network condition [156, 151, 154, 65]. As I investigate the impact of changes in transcoding parameters, the results can guide better resource utilization for these adaptive video streaming services.

3.5 Conclusion

In this paper, I characterize the CPU microarchitectural performance of video transcoding workloads. I vary all the major configurable options of video transcoding operation and explore their impact on microarchitectural performance. I find that most transcoding workloads suffer from back-end issues in the form of high data cache misses. At the same time, video transcoding operations suffer from instruction cache misses and branch mispredictions in some specific scenarios. To overcome data cache misses, I apply polyhedral optimizer, Graphite on transcoding workloads, and achieve 4.42% average speedup. I show that the state-of-the-art profile-guided optimization technique, AutoFDO, can reduce instruction cache misses and branch mispredictions of video transcoding workloads to provide a 4.66% average speedup. Finally, keeping the bottleneck diversity in mind, I propose a smart scheduler that assigns the best microarchitectural configuration for different transcoding tasks. On average, the proposed scheduler outperforms the random scheduler by 3.72% and matches with the best scheduler in 75% of cases.

CHAPTER 4

Demystifying Graph Sparsification Algorithms in Graph Properties Preservation

Graphs are ubiquitous because of their great expressiveness and flexibility. Graphs can be used to represent complex relationships between individuals (vertices in the graph) by making connections (edges in the graph). Graphs are widely used to represent data in various application domains, e.g., social networks [64], citation and communication networks [111], chemical and biological networks [80], etc. Many algorithms are also developed to exploit the abundant features that graphs provide, e.g., Dijkstra's algorithm [55], Ford-Fulkerson algorithm [63], Graph Neural Networks [85], etc.

Despite their usefulness, graphs are often inefficient to work with due to memory irregularity. Many works are proposed to tackle the problem [130, 139, 48]. However, most works develop dedicated software or hardware solutions for a small set of graph algorithms, which leads to a high design cost and limited applicability. In this work, I investigate graph sparsification, a generally applicable technique to reduce the amount of work in graph algorithms.

Graph sparsification is a technique to approximate a given graph by a sparse graph that preserves certain properties. This way, the downstream tasks can be executed on the sparsified graph to improve run time. An ideal sparsification algorithm needs to achieve a high prune rate while keeping the downstream task behavior close to that of the original full graph.

There are many sparsification algorithms with different focuses on the graph properties to be preserved, and of different complexity. There are also many graph metrics that different graph-centric algorithms rely on. However, with a large number of sparsification algorithms and graph metrics, the connections between sparsifiers and their performance in preserving the graph metrics are missing.

In this work, I extensively investigate 12 graph sparsification algorithms and evaluate their performance in preserving 16 widely-used graph metrics in multiple groups. I also cover 14 real-world graphs spanning various categories with diverse characteristics, sizes, and densities. The findings reveal that no single sparsifier does the best in preserving all graph properties, and it is essential to select appropriate sparsifiers based on the downstream task.

In summary, I make the following contributions in this work:

- I summarize the most widely-used graph metrics and the most representative graph sparsification algorithms and dig into the algorithmic details for a better understanding.
- I build a framework to perform graph sparsification and evaluate their performance on various graph metrics at different prune rates. The framework is open-source and extendable to future sparsification algorithms, graph metrics, and graphs.
- I perform N-to-N evaluation on the sparsification algorithms and graph metric, give a comprehensive performance breakdown and provide insights with the results.

4.1 Overview

4.1.1 Preliminaries

In this section, I introduce the basic notions used in this paper.

Consider a graph $\boldsymbol{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{w})$, where $\mathcal{V}$ and $\mathcal{E}$ denotes the set of vertices and edges in $\boldsymbol{G}$ respectively, and $\boldsymbol{w}$ denotes the weights of the edges. A graph can be either directed or undirected. In a directed graph, each edge has a source and a destination vertex, while an undirected graph implies a bidirectional relationship. Furthermore, a graph can be weighted or unweighted; in an unweighted graph, all edges have a default weight of 1. $|\mathcal{V}|$, $|\mathcal{E}|$ represent the number of vertices and edges, respectively. A graph is considered *connected* if a path exists between any pair of vertices [20]. The adjacency matrix is denoted by $\boldsymbol{A}$, with the entries in $\boldsymbol{A}$ defined as:

$$\boldsymbol{A}_{ij} = \begin{cases} \boldsymbol{w}_{i \to j} & \text{if } e_{ij} \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

The graph Laplacian matrix is denoted by $\boldsymbol{L}$ defined as follows:

$$\boldsymbol{L}_{ij} = \boldsymbol{D} - \boldsymbol{A} = \begin{cases} deg(v_i) & \text{if } i = j, \\ -\boldsymbol{w}_{i \to j} & \text{if } e_{ij} \in \mathcal{E}, \\ 0 & \text{otherwise.} \end{cases}$$

Note that the Laplacian matrix is only considered for undirected graphs, thus the graph Laplacian is a positive semi-definite matrix. I now present a formal definition of the graph sparsification problem.

Definition 1 (Graph Sparsification) *Let $\boldsymbol{G} = (\mathcal{V}, \mathcal{E}, \boldsymbol{w})$ be a given graph. A sparsified subgraph $\boldsymbol{H} = (\mathcal{V}, \tilde{\mathcal{E}}, \tilde{\boldsymbol{w}})$ is constructed such that $|\tilde{\mathcal{E}}| = (1-\rho)|\mathcal{E}|$. The function f that creates $\boldsymbol{H}$ from $\boldsymbol{G}$, $\boldsymbol{H} = f(\boldsymbol{G})$, is called a graph sparsification algorithm (also referred to as a sparsifier), while ρ is defined as the prune rate.*

This study focuses solely on edge sparsification, implying that the original vertex set is kept while selecting a subset of edges. This approach is adopted for several reasons: 1) the edge set typically possesses a significantly larger size than the vertex set and contains more redundant information, 2) the majority of sparsification algorithms focus on pruning edges rather than vertices, and 3) most graph metrics require the complete set of vertices for evaluating the performance of sparsification algorithms.

4.1.2 Graph Metrics

4.1.2.1 Basic Metrics

This section introduces some fundamental graph metrics.

Degree Distribution. The degree of a vertex is defined as the number of edges incident to it. The degree distribution provides a comprehensive perspective on the graph's structure, enabling the classification of different types of graphs. For instance, a randomly generated graph might exhibit a uniform degree distribution, whereas a real-world social network has a power-law distribution.

Laplacian Quadratic Form. This is defined as $\boldsymbol{x}^T \boldsymbol{L}\boldsymbol{x}$, where $\boldsymbol{L}$ represents the graph Laplacian, and $\boldsymbol{x} \in \mathbb{R}^{|\mathcal{V}|}$ is an arbitrary vector. The Laplacian quadratic form is a fundamental quantity in graph theory [39], and it facilitates the analysis of various graph properties, including connectivity and spectral characteristics [32].

4.1.2.2 Distance Metrics

This section includes a collection of metrics associated with the pairwise distances between vertices in graphs.

All Pairs Shortest Path (APSP). APSP measures the minimum distance between any pair of source vertex u and destination vertex v. Breadth-First Search (BFS) and Dijkstra's algorithm [55] are often used to determine APSP. Distance captures the proximity between two vertices. APSP are used in various domains such as data center network design [50] and urban service system planning [122].

Diameter. The diameter of a graph $\boldsymbol{G}$ is defined as the maximum distance between any pair of vertices u and v. If $\boldsymbol{G}$ is disconnected, its diameter is considered infinite. The diameter is useful in various applications, including transportation network planning [38] and the analysis of routing and communication network quality [57].

Vertex Eccentricity. Vertex eccentricity is defined as the length of the longest shortest path from a source vertex s to all other vertices in $\boldsymbol{G}$. Note that the minimum eccentricity is the graph radius, and the maximum eccentricity is the graph diameter. Vertex eccentricity is infinite for disconnected graphs. It identifies vertices located near the geometrical center of the graph. Vertex eccentricity has practical applications in identifying network periphery in routing network [102, 137]. Or identifying proteins readily functionally reachable by other components in protein networks. [137, 114].

4.1.2.3 Centrality Metrics

Centrality measures are a set of metrics employed to assess the significance or ranking of vertices in various manners.

Betweenness. Betweenness centrality for vertex v is defined as

$$C_{\text{betweenness}}(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}.$$

Here, σ_{st} denotes the total number of shortest paths from vertex s to t, while $\sigma_{st}(v)$ refers to the number of shortest paths passing through v. The underlying intuition suggests that vertices appearing on numerous shortest paths exhibit high betweenness centrality. It can be employed to identify hubs in a transportation network [128] or to identify important vertices (people) in social networks [41].

Closeness. Closeness centrality [33] of a vertex v is defined as

$$C_{\text{closeness}}(v) = \frac{1}{\sum_u d(u,v)}.$$

Here, $d(u,v)$ represents the shortest distance between vertices u and v. The underlying intuition is that vertices with a shorter average distance to all other reachable vertices exhibit high closeness centrality. It can identify essential genes in protein-interaction networks [71] or crucial metabolites in metabolic networks [101].

Eigenvector. The eigenvector centrality of a vertex v is defined as

$$C_{\text{eigenvector}}(v) = \frac{1}{\lambda} \sum_{u \in N(v)} C_{eigenvector}(u).$$

where $N(v)$ is the neighbour of v, and λ is the greatest eigenvalue of the adjacency matrix $\boldsymbol{A}$. Eigenvector centrality measures the influence of a vertex [22]. A high eigenvector score means a vertex is connected to many vertices whose eigenvector scores are also high [110]. Google's PageRank [113] and Katz centrality are two variants of eigenvector centrality. Katz centrality is discussed in the next paragraph and PageRank in Section 4.1.2.5. Eigenvector centrality is useful for assessing opinion influence in sociology and economics [120], or the firing rate of neurons in neuroscience [62].

Katz. Katz centrality quantifies the influence of a vertex by considering the number of immediate neighbors and vertices connected to those immediate neighbors [82]. Distant neighbors are penalized by an attenuation factor α^k, where k represents the hop distance from the central vertex. In this paper, I use $\alpha = 1/(max(degree) + 1)$. The eigenvector centrality is defined as

$$C_{Katz}(v) = \sum_{k} \sum_{u} \alpha^k (\boldsymbol{A}^k)_{uv}.$$

4.1.2.4 Clustering Metrics

Graph clustering groups vertices into communities, ensuring dense connections within communities and sparse connections between communities. This section covers graph clustering-related metrics.

Number of communities. The most basic metric in graph clustering is the number of communities. For graphs with a known number of communities k, certain clustering algorithms, such as k-means [98], can construct exactly k communities. Alternatively, some algorithms like agglomerative clustering [109] and DBSCAN [58] can automatically determine the optimal number of clusters.

Local Clustering Coefficient (LCC). LCC of a vertex v represents the proportion of pairs of neighbors of v that are connected. It evaluates the density of connections among the neighbors of a vertex [19]. The LCC is defined as follows:

$$LCC(v) = \frac{|e_{jk} : j, k \in N_v, e_{jk} \in E|}{\alpha k_v(k_v - 1)}.$$

where N_v denotes the set of neighbors of the vertex v, and k_v is the number of neighbors of vertex v. Here, $\alpha = 1$ for directed graphs, and $\alpha = 0.5$ for undirected graphs. LCC, originally proposed by Watts and Strogatz, is used to determine whether a graph is a small-world network [143]. **Mean clustering coefficient (MCC)** is the mean of the local clustering coefficient of all vertices.

Global Clustering Coefficient (GCC). GCC [100] measures the fraction of closed triplets in all triplets. A triplet of nodes can consist of two (open) or three (closed) undirected edges [19].

$$GCC(v) = \frac{\#Closed\ triplets}{\#All\ triplets}.$$

Clustering F1 score. The F1 score can be employed to assess the similarity between a given clustering and a reference clustering [103]. Suppose there are k clusters C_i $(i \in [1, k])$ obtained from a specific algorithm for graph G and s reference clusters R_j $(j \in [1, s])$ to compare with. Note that s may not be equal to k. The following matrix illustrates the relationship between C_i and R_j:

$$\begin{array}{ccccc} & \boldsymbol{R_1} & \boldsymbol{R_2} & \boldsymbol{\ldots} & \boldsymbol{R_s} \\ \boldsymbol{C_1} & a_{11} & a_{12} & \ldots & a_{1s} \\ \boldsymbol{C_2} & a_{21} & a_{22} & \ldots & a_{2s} \\ \boldsymbol{\ldots} & & & & \\ \boldsymbol{C_k} & a_{k1} & a_{k2} & \ldots & a_{ks} \end{array}$$

In this matrix, a_{ij} represents the number of vertices shared between cluster C_i and reference cluster R_j. The precision and recall of the clustering are defined as follows:

$$Precision = \frac{\sum_{i\in[1,k]} max_j\{a_{ij}\}}{\sum_{i\in[1,k]}\sum_{j\in[1,s]} a_{ij}}, \quad Recall = \frac{\sum_{i\in[1,k]} max_j\{a_{ij}\}}{n}$$

Subsequently, the F1 score for clustering is defined as:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

The F1 score ranges from 0 to 1, where a higher value indicates greater similarity between the clustering $\boldsymbol{C}$ and the reference $\boldsymbol{R}$.

Table 4.1: Metrics' applicability to types of graphs.

Metric	Directed	Weighted	Unconnected
Degree Dist.	✓	●†	✓
Diameter	✓	✓	✓‡
Eccentricity	✓	✓	✓‡
APSP	✓	✓	✓‡
Betweenness Cent.	✓	✓	✓
Closeness Cent.	✓	✓	✓
Eigenvector Cent.	✓*	✓	✓
Katz Cent.	✓	✓	✓
#Communities	✗	✓	✓
LCC	✓	●†	✓
MCC	✓	●†	✓
GCC	✓	●†	✓
Clustering F1 Sim	✗	✓	✓
PageRank	✓	✓	✓
Min-cut/Max-flow	✓	✓	✓‡
GNN	✓	✓	✓

$*$ For directed graphs, the left eigenvector is used. A left eigenvector is an eigenvector satisfies $X_L\boldsymbol{A} = \lambda_L X_L$, where a right (by default) eigenvector satisfies $\boldsymbol{A}X_R = \lambda_R X_R$

† Weight not used, same as unweighted.

‡ In unconnected graphs, pairwise distance can be infinite, and min-cut max-flow can be zero if two terminals selected are in different communities. I exclude these pairs in the evaluation.

4.1.2.5 Application-level Metrics

In this section, I discuss metrics that are used in applications.

PageRank. PageRank, initially designed to rank web pages [113], is a foundational algorithm for Google's search engine. The underlying concept suggests that pages linked by numerous important pages bear greater significance. PageRank computation typically employs the power method. Each page (vertex) is assigned an initial score and iteratively calculates a new score by adding up $1/k$ of the scores of pages linked to it, where k represents the number of outgoing links from the source page. Eventually, the computation converges, and the score of each page indicates its importance within the network. The primary distinction between PageRank and eigenvector centrality (§ 4.1.2.3) lies in PageRank's specificity for web-page ranking, incorporating $1/k$ factor and additional parameters like damping factor [40] for better robustness and accuracy, while eigenvector centrality is more suitable for general graph analysis, not necessarily involve directed or weighted graphs.

Min-cut and Max-flow. In graph theory, a cut refers to the partitioning of a graph's vertices into two disjoint subsets [21]. A minimum s-t cut, or min-cut, represents the cut with the smallest total weight of edges that disconnect the source vertex s from the sink vertex t. The maximum flow, or max-flow, denotes the maximum amount of flow that can traverse from the source vertex s to the sink vertex t, where the edge weight represents the flow capacity. The max-flow and min-cut problems are equivalent, as the maximum flow a network can accommodate is constrained by the network's narrowest intersection, which is the min-cut. Min-cut and max-flow can be applied to identify bottlenecks in water networks, road networks, or electrical networks [119, 24].

Graph Neural Networks (GNNs). GNNs [127] are neural networks that operate on graphs. GNNs learn from the graph structure by aggregating information from neighboring vertices or edges and feeding the information to multi-layer perception (MLP) layers for training. Some famous GNN models include Graph Convolutional Network (GCN), Graph

Attention Network (GAT), and ChebNet [85, 53, 140]. GNN can be used for classification or prediction on vertex, edge, or graph-level tasks [160, 92, 54].

I summarize the graph metrics discussed and their applicability to different types of graphs in table 4.1.

4.1.3 Graph Sparsification Algorithms

Graph sparsification is to approximate a given graph by a graph with fewer vertices or edges. In this work, I consider graph sparsification algorithms that keep the same vertices of the original graph and only remove edges. This is because most of the metrics discussed in the previous section are vertex-centric; for example, distance metrics are for each vertex or each pair of vertices; centrality metrics are about the ranking of vertices; PageRank is the vertex ranking; min-cut and max-flow are also vertex pairwise.

In this section, I discuss graph sparsification algorithms evaluated in this work; they constitute the most widely used and representative sparsification algorithms.

4.1.3.1 ***Random*** Sparsifier

The simplest way to sparsify the graph is by randomly sampling a subset of edges to keep it in the sparsified graph. This is referred to as the *Random* sparsifier. It samples all edges in the graph with equal probability and thus can be used to preserve vertex-relative (distribution-based and ranking-based) properties. *Random* sparsifier is employed in GraphSAGE for neighbor sampling [73].

4.1.3.2 ***K-Neighbor*** Sparsifier

K-Neighbor sparsifier [124] selects k edges for each vertex, and if a vertex has less than k vertices, all of its edges are included. The edges are selected with probability proportional to their weights (uniform for unweighted graphs). It can be used in Laplacian smoothing [124]. *K-Neighbor* guarantees each vertex has at least k edges, so it can be applied if the downstream task requires high graph connectivity.

Table 4.2: Sparsifiers' applicability to types of graphs and characteristics. Note that all sparsifiers work for undirected, unweighted, and connected graphs because they are special cases of directed, weighted, and unconnected graphs, so they are not listed. Deterministic means whether the sparsifier generates the same sub-graph every time.

Sparsifier	**D?**	**W?**	**Un-C?**	**PRC**	**WC?**	**Det?**	**Complexity****
Random **(RN)**	✓	✓	✓	✓	✗	✗	$\mathcal{O}(\rho\vert\mathcal{E}\vert)$
K-Neighbor **(KN)**	✓*	✓	✓	✓‡	✗	✗	$\mathcal{O}(\vert\mathcal{E}\vert)$
Rank Degree **(RD)**	✓*	✓	✓	✓‡	✗	✗	$\mathcal{O}(\rho\vert\mathcal{E}\vert) - \mathcal{O}(\rho\vert\mathcal{E}\vert)log(\rho\vert\mathcal{E}\vert)$
Local Degree **(LD)**	✓*	✓	✓	✓‡	✗	✓	$\mathcal{O}(\vert\mathcal{E}\vert) - \mathcal{O}(\vert\mathcal{E}\vert log(\vert\mathcal{E}\vert))$
Spanning Forest **(SF)**	✗	✓	✓	✗	✗	✓	$\mathcal{O}(\vert\mathcal{E}\vert log(\vert\mathcal{V}\vert))$
t-Spanner **(SP-t)**	✗	✓	✓	✗	✗	✓	$\mathcal{O}(\vert\mathcal{V}\vert^2 log(\vert\mathcal{V}\vert))$
Forest Fire **(FF)**	✓	✓	✓†	✓‡	✗	✗	$\mathcal{O}(r\vert\mathcal{E}\vert)$
L-Spar **(LS)**	✓*	✓	✓	✓‡	✗	✓	$\mathcal{O}(k\vert\mathcal{E}\vert)$
G-Spar **(GS)**	✓*	✓	✓	✓	✗	✓	$\mathcal{O}(k\vert\mathcal{E}\vert)$
Local Similarity **(LSim)**	✓*	✓	✓	✓‡	✗	✓	$\mathcal{O}(\vert\mathcal{E}\vert)$
SCAN	✓*	✓	✓	✓	✗	✓	$\mathcal{O}(\vert\mathcal{E}\vert)$
ER	✗	✓	✓	✓	✓	✗	$\mathcal{O}(\vert\mathcal{E}\vert log(\vert\mathcal{V}\vert)^3)$

The header of each column is: **D?** means whether the sparsifier works on a **directed** graph. **W?** means whether the sparsifier works on a **weighted** graph. **Un-C?** means whether the sparsifier works on an **un-connected** graph. **PRC** is **prune rate control**, and it means whether the sparsifier has a fine-grain, coarse-grain, or no control over the prune rate. **WC?** means whether the sparsifier makes **weight changes** to in the sparsified graph. **Det?** means whether the sparsifier generates **deterministic** results across runs. $*$ Need to specify using in-degree or out-degree; in this work, I use out-degree.
$**$ $|\mathcal{V}|$ =#Vertices, $|\mathcal{E}|$ =#Edges, ρ =prune rate, r =burnt ratio, k =#minwise hash. It can be a range for some sparsifiers because different optimal algorithms can be used according to graph properties.
† Seeds are randomly selected, thus edges from communities with fewer vertices are less likely to be included.
‡ Subject to constraint. Indirect or coarser grain control or has an upper limit for prune rate.

4.1.3.3 *Rank Degree* Sparsifier

Rank Degree sparsifier [141] starts with selecting a random set of "seed" vertices. Subsequently, the vertices with edges to the seed vertices are ranked according to their degree in descending order. The edges connecting each seed vertex to its top-ranked neighbors are selected and incorporated into the sparsified graph. The recently added nodes in the graph serve as new seeds to search for additional edges. This process continues until the target sparsification limit is reached. *Rank Degree* biases to high-degree vertices, which are considered the hub vertices in a graph, so it excels at keeping edges incident to the important vertices in graphs.

4.1.3.4 *Local Degree* Sparsifier

Similar to the *Rank Degree* sparsifier, the *Local Degree* sparsifier [72] preserves edges incident to high-degree vertices, but in a deterministic manner. For each vertex, *Local Degree* incorporates edges to the top $deg(v)^\alpha$ neighbors ranked by their degree in descending order, where $\alpha \in [0, 1]$ controls the degree of sparsification. Another difference compared to *Rank Degree* is that *Local Degree* sparsifier makes sure each vertex will have at least one edge, so *Local Degree* sparsifier is a good choice when one desires to keep both graph connectivity and edges incident to important vertices.

4.1.3.5 *Spanning Forest*

A spanning tree is a subgraph that constitutes a tree (a connected graph without a cycle [18]) and includes all the vertices in the graph [17]. A *Spanning Forest* consists of multiple spanning trees. Kruskal's algorithm [87] and Prim's algorithm [118] can be used to construct a *Spanning Forest*. Although it is not strictly a sparsifier, as the prune rate cannot be controlled, *Spanning Forest* is included because it reduces the size of graphs and is a fundamental notion in graph theory. *Spanning Forest* is helpful when one strictly wants to keep the graph connectivity the same as the original graph.

4.1.3.6 *t-Spanner*

A spanner is a subgraph approximating the pairwise distances between vertices in the original graph. A *t-Spanner* is defined as a subgraph such that any pairwise distance is at most t times the distance in the original graph, which can be formally expressed as:

$$\forall u, v \in \boldsymbol{V}, d_{\boldsymbol{H}}(u, v) \leq t d_{\boldsymbol{G}}(u, v)$$

In this equation, $t(> 1)$ denotes the stretch factor. A greedy algorithm [27] is employed for constructing t-spanners. This algorithm starts with an empty edge set and then iteratively adds the edge e_{uv} if the distance $d_H(u, v)$ between the vertices u and v in the current graph exceeds t times the weight of e_{uv}. The process continues until all edges have been considered. In addition to strictly keeping the graph connectivity, *t-Spanner* provides a better guarantee on the pairwise distances between vertices and is a better choice than *Spanning Forest* when such a property is desired.

4.1.3.7 *Forest Fire*

The *Forest Fire* model is a generative model for graphs, originally proposed by Leskovec et al. [90]. The concept involves constructing the graph by adding one vertex at a time and forming edges to certain subsets of the existing vertices. When a new vertex u is added to the graph, it connects to an existing vertex v in the graph. Subsequently, it "spreads" from v to other vertices in the graph with a certain predefined probability, creating edges between v and the newly discovered vertices. This process assembles "burning" through edges probabilistically, hence the name *Forest Fire* [90].

4.1.3.8 Similarity-based sparsifiers

Similarity-based sparsifiers constitute a group of sparsifiers based on similarities between vertices measured by specific metrics.

Jaccard similarity [108] measures the similarity between two sets by computing the portion of shared neighbors between two nodes (u and v), as defined below:

$$JaccardSimilarity(u, v) = \frac{|\mathcal{N}(u) \bigcap \mathcal{N}(v)|}{|\mathcal{N}(u) \bigcup \mathcal{N}(v)|}$$

The *Jaccard score* of an edge is the Jaccard similarity between two constituent vertices of the edge. Once Jaccard scores are computed, they can be used to perform similarity-based sparsifications.

Global Sparsifiers. Global sparsifiers select edges based on similarity scores globally. *global Jaccard sparsifier* (*G-Spar*) sorts the Jaccard scores globally and then selects the edges with the highest similarity score. *SCAN* [150] uses structural similarity measures to detect clusters, hubs, and outliers. The SCAN similarity score is a modified version of the Jaccard score, defined as follows:

$$SCANSimilarity(u,v) = \frac{|\mathcal{N}(u) \bigcap \mathcal{N}(v)| + 1}{\sqrt{(deg(u)+1)(deg(v)+1)}}$$

Once the scores are computed, the edges in the sparsified graph are included from high-score edges to low-score edges.

Local Sparsifiers. Similarity scores can also be used to select edges in a local way. The *local Jaccard similarity sparsifier* (*L-Spar*) [125] includes d^c edges with the highest Jaccard scores incident to each vertex locally, where c is a parameter. The *Local Similarity* sparsifier works similarly to *L-Spar*, but it further ranks edges using the Jaccard score and computes $log(rank(edge))/log(deg(v))$ as the similarity score. Finally, *Local Similarity* sparsifier selects edges with the highest similarity scores.

The *L-Spar* and *Local Similarity* sparsifiers are particularly useful for preserving local structure in the graph, such as clustering. They can be applied to social network analysis and recommendation systems. By focusing on local similarities between vertices, these sparsifiers provide a more accurate representation of the original graph's local properties compared to other sparsifiers.

Table 4.3: Graph datasets information.

Category	Name	D?	W?	C?	#Nodes	#Edges	Density	source
Social Network	ego-Facebook	✗	✗	✓	4,039	88,234	1.08E-02	snap [91]
	ego-Twitter	✓	✗	✗	81,306	1,768,149	2.67E-04	snap [91]
Gene	human_gene2	✗	✓	✗	14,340	9,041,364	8.79E-02	SuiteSparse [51]
Community Network	com-DBLP	✗	✗	✓	317,080	1,049,866	2.09E-05	snap [91]
	com-Amazon	✗	✗	✓	334,863	925,872	1.65E-05	snap [91]
communication	email-Enron	✗	✗	✗	36,692	183,831	2.73E-04	snap [91]
Collaboration	ca-AstroPh	✗	✗	✗	18,772	198,110	1.12E-03	snap [91]
	ca-HepPh	✗	✗	✗	12,008	118,521	1.64E-03	snap [91]
Web	web-BerkStan	✓	✗	✗	685,230	7,600,595	1.62E-05	snap [91]
	web-Google	✓	✗	✗	875,713	5,105,039	6.66E-06	snap [91]
	web-NotreDame	✓	✗	✗	325,729	1,497,134	1.41E-05	snap [91]
	web-Stanford	✓	✗	✗	281,903	2,312,497	2.91E-05	snap [91]
GNN	Reddit	✗	✗	✓	232,965	57,307,946	2.11E-03	pyg [73]
	ogbn-proteins	✗	✗	✓	132,534	39,561,252	4.50E-03	ogb [136, 76]

The header of each column is: **D?** means whether the graph is **directed**. **W?** means whether the graph is **weighted**. **C?** means whether the graph is **connected**.

4.1.3.9 Effective Resistance (*ER*) Sparsifier

The concept of Effective Resistance (*ER*) is derived from the analogy of an electrical circuit and applied to a graph. In this context, edges represent resistors, and the effective resistance of an edge corresponds to the potential difference generated when a unit current is introduced at one end of the edge and withdrawn from the other.

I refer readers to [132] for the details of how *ER* is calculated. Once the effective resistance is calculated, a sparsified subgraph can be constructed by selecting edges with a probability proportional to their effective resistances. Notably, Spielman and Srivastava further proved that the quadratic form for Laplacian of such sparsified graphs is close to that of the original graph. Then the following inequality holds for the sparsified subgraph with high probability:

$$\forall \boldsymbol{x} \in \mathbb{R}^{|\mathcal{V}|} \quad (1-\epsilon)\boldsymbol{x}^T \boldsymbol{L}\boldsymbol{x} \leq \boldsymbol{x}^T \tilde{\boldsymbol{L}}\boldsymbol{x} \leq (1+\epsilon)\boldsymbol{x}^T \boldsymbol{L}\boldsymbol{x}$$

where $\tilde{\boldsymbol{L}}$ is the Laplacian of the sparsified graph, and $\epsilon > 0$ is a small number. The insight is that *ER* reflects the significance of an edge. *ER* is a spectral sparsifier, and it aims to preserve the quadratic form of the graph Laplacian. It can be applied to applications that rely on the quadratic form of graph Laplacian, such as min-cut/max-flow.

I list the sparsifiers discussed in the section and their applicability to types of graphs, features, and time complexity in table 4.2.

4.1.4 Datasets

Table 4.3 lists the graph datasets used in this work; I select graphs from various categories with different characteristics, sizes, and densities to ensure the diversity of graphs.

4.2 Experimental Setup

4.2.1 Graph Preparation

The graphs employed in this study are sourced from multiple graph dataset suites. I carry out essential pre-processing steps on all graphs to ensure their proper preparation for sparsifier execution and metric evaluation. The process can be summarized as follows:

1. I remove vertices with no edge incidence (i.e., isolated vertices), as they do not contribute to graph information and can induce noise in metric evaluations. Then, vertices are re-indexed to be zero-based and continuous.

2. For each directed graph, an undirected version is generated by symmetrizing each edge (i.e., adding a [dst, src] edge to the graph if it does not already exist). This ensures that sparsifiers that only operate on undirected graphs can function properly. Other sparsifiers are still applied to the original directed graphs.

4.2.2 Graph Sparsification

In this section, I cover additional information regarding the graph sparsifiers. When applying sparsifiers:

1. I sweep the prune rate from 0.1 to 0.9, with a step of 0.1. Some sparsifiers have a coarser prune rate granularity (e.g., *K-Neighbor*, *L-Spar*), and I attempt to align them with the specified prune rate. Some sparsifiers have a maximum prune rate (e.g., *Local Degree*, *K-Neighbor*), so I sweep up to their maximum prune rate. Certain sparsifiers have no control over the prune rate and only support a single prune rate (e.g., *Spanning Forest*, *t-Spanner*), and I retain them as is.

2. For non-deterministic sparsifiers, the inherent randomness in the algorithm produces different sub-graphs in each run. In such cases, I generate 10 graphs at each prune rate,

measure graph metrics using the mean value, and indicate their standard deviation in the results. For deterministic sparsifiers, I generate a single graph at each prune rate.

3. For the Effective Resistance sparsifier, since it is the only one that modifies edge weights, I consider two variants denoted as *ER-weighted* and *ER-unweighted*, respectively.

4.2.3 Graph Metrics

In this section, I cover additional information regarding the measurement of sparsifiers' quality on graph metrics.

4.2.3.1 Basic Metrics

Graph connectivity. I employ the source-destination pair unreachable ratio and the vertex isolated ratio to measure graph connectivity. The former represents the fraction of vertex pairs that do not have a path connecting them. The latter signifies the proportion of isolated vertices, meaning no edges are incident to them. Both of these ratios provide insights into the overall connectivity of a graph when assessing the effectiveness of sparsification methods.

Degree Distribution. I assess how closely the similarity of the degree distribution of the sparsified graphs and that of the original graph using the Bhattacharyya distance [35], defined as:

$$B_d(P, Q) = -ln\left(\sum_{x \in \mathcal{X}} \sqrt{P(x)Q(x)}\right)$$

where P and Q are two distributions. A value closer to 0 indicates a higher similarity in distribution. I evenly divide the discrete degree distribution into 100 bins for all graphs.

Quadratic Form Similarity. To evaluate this, I generate 100 vectors $\boldsymbol{x}$ with random entries. Next, I compute the quadratic form $\boldsymbol{x}^T \boldsymbol{L} \boldsymbol{x}$ for the original and the sparsified graphs. Then, I use the mean quadratic form ratio to assess the sparsification quality.

4.2.3.2 Distance Metrics

APSP and Eccentricity. The computation of the All-Pair-Shortest-Path (APSP) is time-consuming for large graphs. Therefore, I randomly sample 100,000 source-destination pairs, referred to as Some-Pair-Shortest-Path (SPSP), and report the average stretch factor, defined as the distance ratio between the same pair in the sparsified and the original graph. I exclude pairs belonging to different communities. Similarly, I randomly select 1000 vertices to represent the eccentricity of all vertices.

Diameter. Computing the true diameter requires performing APSP, which is impractical on large graphs. I employ an approximate diameter algorithm [55]. The algorithm starts with a randomly chosen source vertex, identifies a target vertex farthest from it, and iteratively repeats the process using the target vertex as the new source vertex. I validated the approximate diameter against the true diameter on small graphs and verified that they are closely aligned. To minimize potential bias introduced by the initial source vertex selection, each graph is assessed using 10 different randomly chosen seed vertices to obtain the mean diameter.

4.2.3.3 Centrality Metrics

I employ the top-k precision to evaluate the quality of centrality metrics. First, vertices are ranked according to their centrality scores. Then, the top-k vertices in the sparsified graphs are compared with those in the full graph. The proportion of overlapping vertices is referred to as the top-k precision. In this paper, I set k to 100 because, typically, only a small subset of vertices in graphs are critical, and accurately ranking them is more important.

Betweenness Centrality. Actual betweenness centrality calculation also requires computing APSP. In this paper, I adopt an approximate betweenness centrality algorithm proposed by Geisberger *et al.* [66]. The algorithm is sampling-based, and a higher sampling number achieves better estimation quality. I use a sampling number of 500 and compare it with exact betweenness on small graphs, confirming the results are closely aligned.

4.2.3.4 Application-level Metrics

Min-cut/Max-flow. I randomly sample 100,000 src-dst pairs and measure the min-cut/max-flow on both the original and sparsified graphs. Then, I use the mean stretch factor between the sparsified and the original graph to evaluate the sparsification quality.

GNN. For GNNs, I evaluate two models: GraphSAGE and ClusterGCN. The quality is measured in test accuracy or Area Under the Receiver Operating Characteristics [59] (AUROC). AUROC ranges from 0.5 to 1. A higher accuracy or AUROC indicates better GNN performance. For both GNN models, I train the network with sparsified graph and test on the full graph because 1) training is the most time-consuming part and is the most meaningful to apply sparsification, 2) testing on the full graph reveals how well the sparsified graph captures full graph's characteristics.

4.2.4 Software Framework

My software evaluation framework integrates several open-source libraries and my custom implementations. I use `NetworKit`[133] for multiple sparsifiers and `Laplacians.jl`[131] for the effective resistance sparsifier. I also implemented the *K-Neighbor*, *Rank Degree*, *L-Spar*, and *t-Spanner* algorithms.

For the evaluation metrics, I employ both `NetworKit` [133] and `graph-tool` [116] for implementations of several discussed distance, centrality, clustering, and min-cut/max-flow metrics. I use `PyG` [60] to implement the graph neural networks. Additionally, I implemented degree distribution and quadratic form evaluation.

The framework is open-sourced and extendable to incorporate more sparsification algorithms and graph metrics.

4.2.5 Hardware Platform

The experiments in this paper are performed on a server with an Intel Xeon Platinum 8380 CPU with 1 TB of memory. The graph neural networks run on an Nvidia A40 GPU with 48

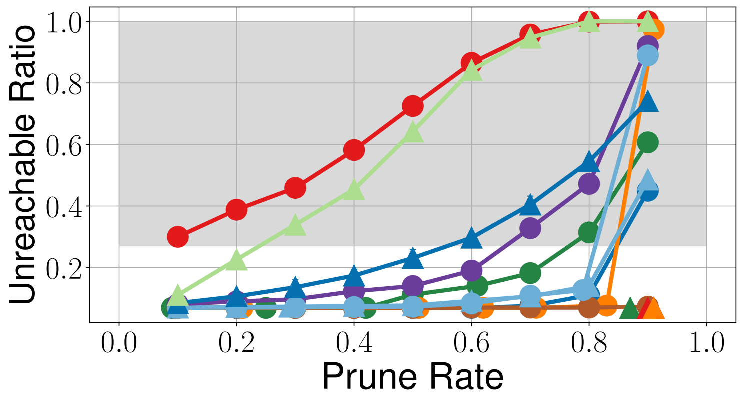
GB memory.

4.3 Results

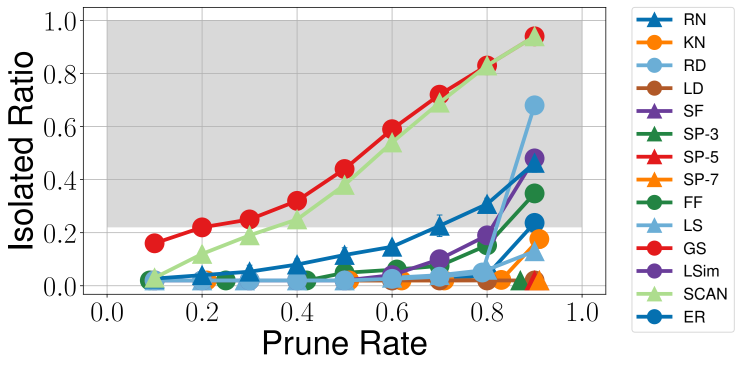
In this section, I evaluate the impact of various sparsifiers on the quality of graph metrics at different prune rates. I perform comprehensive experiments on all sparsifiers, graph metrics, and datasets discussed in this paper. Due to the extensive nature of the experiments (over 30,000 data points), I can only show a subset of performance results in the figures. The full results are available in the appendix. I adhere to the following rules to present the results without bias: (1) for readability, I only show a representative subset of sparsifiers for each graph metric, including those that perform well or poorly and those that yield interesting outcomes; (2) I always include *Random* as it serves as a naive sparsifier for comparison; (3) I select at least one representative graph for each graph metric and discuss any discrepancies observed in other graphs. I then compare sparsification times and briefly discuss the overhead associated with sparsification. Finally, I summarize the results and provide insights.

4.3.1 Basic Metrics

Figures 4.1a and 4.1b show the source-destination pair unreachable ratio and vertex isolated ratio, respectively. As the prune rate increases, the graph becomes more disconnected, leading to an increase in isolated vertices. *K-Neighbor* excels at preserving graph connectivity because it ensures that each vertex retains at least k edges. Two local sparsifiers, *Local Degree* and *Local Similarity*, also show strong performance since they both select edges to maintain locally, guaranteeing at least one edge for each vertex. *ER* performs well by retaining high-resistance edges, which are the low-redundancy edges crucial for maintaining graph connectivity. *Spanning Forest* and *t-Spanners* preserve the same level of connectivity as the original graph, as ensured by the algorithms. *Random* does not effectively preserve graph connectivity because it does not attempt to maintain edges critical for connectivity. *G-Spar* and *SCAN* retain edges connecting similar vertices on a global scale, and these edges are often intra-community edges that are not crucial for preserving connectivity, resulting in the poorest performance. The acceptable unreachable/isolated ratio can be customized according to specific applications. In this paper, I consider an increase of 20% or more in the unreachable/isolated ratio compared to the original graph as excessive (shown as the grey area in Figures 4.1a and 4.1b).



(a) Pair Unreachable Ratio



(b) Vertex Isolated Ratio

Figure 4.1: Graph Connectivity on ca-AstroPh.

Degree Distribution. Figure 4.2 illustrates the degree distribution on ogbn-proteins. A lower Bhattacharyya distance signifies a more similar degree distribution to the original graph. *Random* demonstrates the best performance in preserving the degree distribution. This is because *Random* treats all edges without bias, thus maintaining the same proportion of edges for all vertices and keeping a similar degree distribution. Most graphs exhibit a

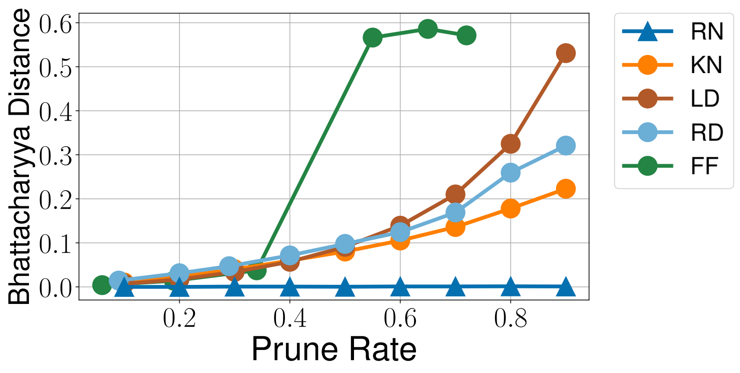


Figure 4.2: Degree distribution comparison on ogbn-proteins. Lower is better. *Random* performs the best, *Local Degree* and *Forest Fire* do not do well in preserving degree distribution.

power-law degree distribution, so some sparsifiers struggle to preserve degree distribution. For instance, *Local Degree* and *Rank Degree* retain edges connected to high-degree vertices. Conversely, *K-Neighbor* maintains up to K edges for all vertices, eliminating surplus edges from high-degree vertices. These biases negatively impact the preservation of the degree distribution. Among all sparsifiers, *Random* consistently performs well across all graphs, while *Local Degree*, *Rank Degree*, *K-Neighbor*, and *Forest Fire* under-perform on most graphs. The performance of other sparsifiers moderately fluctuates across graphs due to different graph characteristics.

Laplacian Quadratic Form. Figure 4.3 displays the Laplacian quadratic form similarity on com-Amazon. A value closer to 1 indicates better quality. From the figure, *ER-weighted* emerges as the clear winner. This is because the Laplacian quadratic form is the specific attribute *ER-weighted* is designed to preserve. Note that only *ER-weighted* possesses this property. *ER-unweighted*, along with other sparsifiers, exhibits no capability to preserve Laplacian quadratic form similarity at all, and they show the same pattern as *Random*. The pattern observed on com-Amazon is consistent across other undirected graphs. For directed graphs (not shown due to space limit), the Laplacian quadratic form ratio for *ER-weighted* is no longer guaranteed to be close to 1; this is because the symmetrization process deviates the graph's spectral property from that of the original directed graph. However, *ER-weighted*

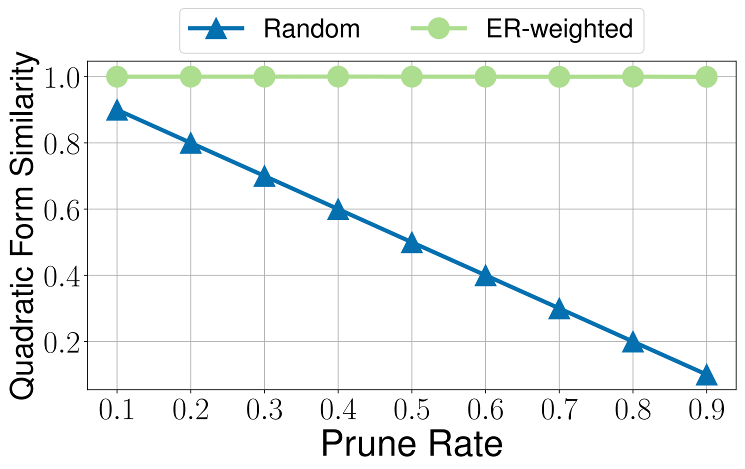


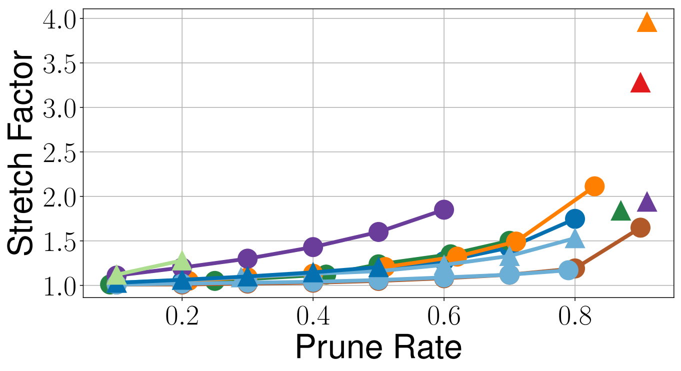
Figure 4.3: Laplacian quadratic form comparison of different sparsifiers on com-Amazon. Closer to 1 is better. *ER-weighted* performs the best. *Random* and other sparsifiers do not preserve Laplacian quadratic form.

still maintains a constant ratio and offers a better guarantee than other sparsifiers.

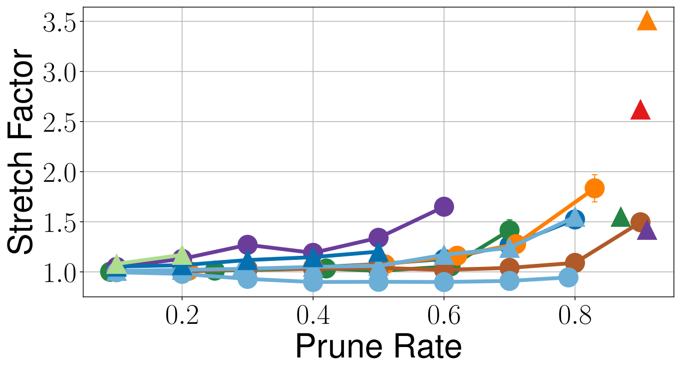
4.3.2 Distance Metrics

SPSP. A practical sparsifier should keep the mean stretch factor close to 1 while keeping the unreachable ratio relatively low. Figure 4.4a shows the mean stretch factor of 100,000 sampled source-destination pairs, with the constraint that the unreachable ratio is $< 20\%$ over that in the original graph (white area in figure 4.1a). This allows for a comparison of the mean stretch factor without a significant increase in the number of unreachable pairs. *Local Degree* and *Rank Degree* demonstrate the best performance in preserving distances while maintaining a low unreachable ratio. This is because both of them bias towards preserving edges of high-degree vertices, which are typically hub vertices in the graph and often lie along many shortest paths.

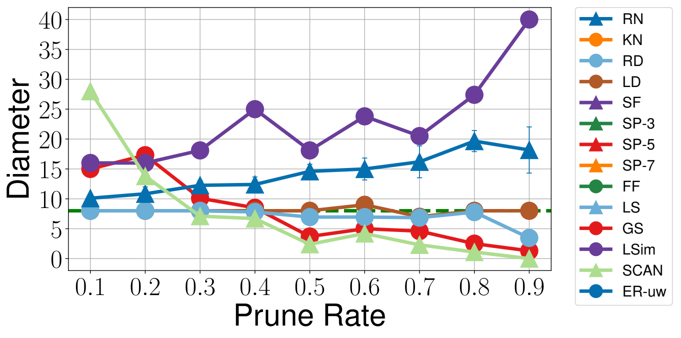
L-Spar, *ER-unweighted*, *Forest Fire*, and *K-Neighbor* also exhibit strong performance due to their ability to maintain graph connectivity. Conversely, *G-Spar* and *SCAN* perform poorly as they rapidly increase the unreachable ratio and have a higher stretch factor. Although *Spanning Forest* and *t-Spanners* have a relatively high stretch factor, they guarantee the



(a) Adjusted SPSP Stretch Factor



(b) Adjusted Eccentricity Stretch Factor



(c) Diameter

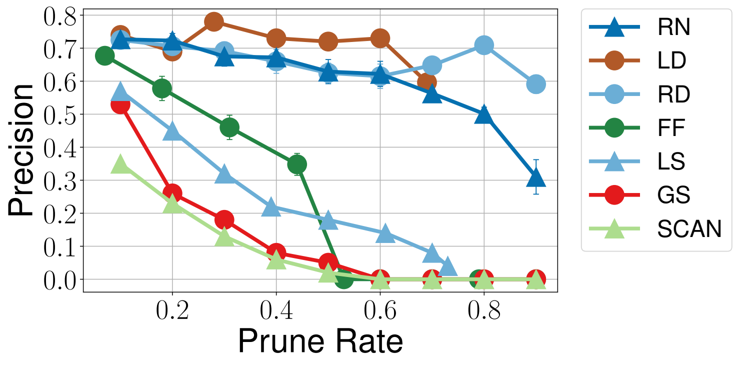
Figure 4.4: (a) Adjusted SPSP stretch factor of sparsifiers on ca-AstroPh with the constraint of acceptable pair unreachable ratio. (b) Adjusted eccentricity stretch factor of sparsifiers on ca-AstroPh with the constraint of acceptable vertex isolated ratio. (c) Diameter comparison on ego-Facebook. For the stretch factor, closer to 1 is better. For graph diameter, closer to ground truth (green line) is better. *Rank Degree* and *Local Degree* have the best performance. *G-Spar* and *SCAN* do not perform well.

connectivity of the original graph, allowing them to maintain the unreachable ratio. *t-Spanners* fulfill the guarantee that the stretch factor is at most t but empirically show a higher mean stretch factor than *Local Degree*. *t-Spanners* is useful when connectivity is paramount, and a slightly higher stretch factor is tolerable.

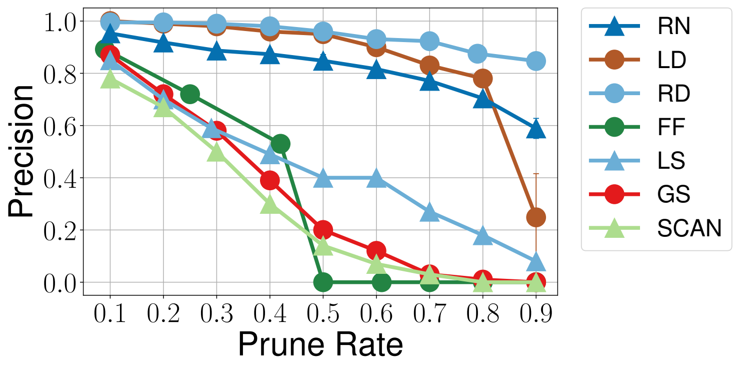
Eccentricity. Figure 4.4b presents the performance of sparsifiers with the vertex isolation ratio is $< 20\%$ higher than that in the original graph (white area in figure 4.1b). *Local Degree* and *Rank Degree* best preserve eccentricity while keeping the unreachable ratio low. *L-Spar*, *ER-unweighted*, *Forest Fire*, and *K-Neighbor* also show strong performance due to their ability to maintain graph connectivity. *G-Spar* and *SCAN* perform poorly compared to other sparsifiers. *Spanning Forest* and *t-Spanners* have a relatively high stretch factor but guarantee the graph connectivity. Additionally, *t-Spanners* provide a theoretical upper bound on the stretch factor, making them suitable for certain scenarios.

Diameter. Figure 4.4c presents the diameters of various sparsified graphs at various prune rates. The green dashed line (8) indicates the diameter measured on the full graph as ground truth. I observe that *Local Degree* and *Rank Degree* perform the best, consistent with their strong performance in preserving distance. *G-Spar*, *SCAN*, and *Local Similarity* perform poorly compared to other sparsifiers.

In general, distance-related metrics are consistent across graphs. Some graphs (e.g., com-Amazon) have a lower average degree, causing the unreachable ratio or vertex isolation ratio to increase faster than in other graphs. *Local Degree* and *Rank Degree* consistently demonstrate the best performance for all distance-related metrics; however, *Local Degree* more effectively maintains the connectivity. *G-Spar* and *SCAN* always under-perform because they both tend to keep intra-community edges. This leads to a more disconnected graph and a high unreachable/isolation ratio.



(a) Betweenness Centrality



(b) Closeness Centrality

Figure 4.5: Top-100 precision for Betweenness and Closeness centrality. Higher is better. (a) Betweenness centrality on com-DBLP. (b) Closeness centrality on ca-AstroPh. *Local Degree*, *Rank Degree*, and *Random* have the best performance. *L-Spar*, *G-Spar*, *SCAN*, and *Forest Fire* do not perform well.

4.3.3 Centrality Metrics

Betweenness and Closeness Centrality. Figure 4.5a and 4.5b display the top-100 precision of betweenness centrality on com-DBLP and closeness centrality on ca-AstroPh. *Local Degree* and *Rank Degree* exhibit the best performance. This is because the top-scored vertices are typically hub vertices, and as explained in § 4.3.2, both *Local Degree* and *Rank Degree* preserve edges incident to high-degree vertices, thus maintaining the betweenness and closeness ranking of hub vertices. *Random* uniformly samples edges without bias and preserves the relative ranking to some extent. *G-Spar* and *SCAN* perform poorly as they aggressively disconnect graphs. I consistently observe *Local Degree*, *Rank Degree*, and *Random* perform well, and *G-Spar* and *SCAN* perform poorly across graphs.

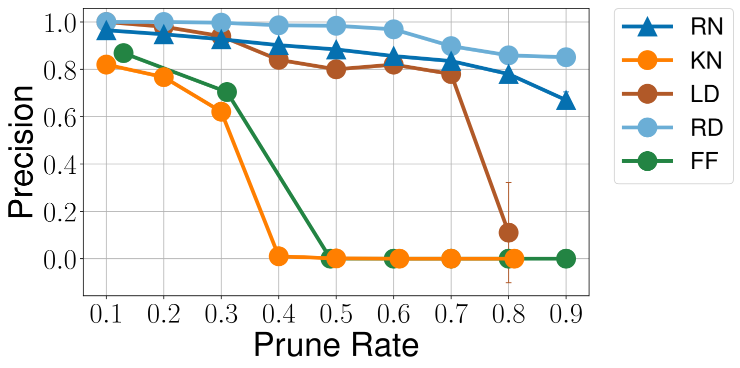


Figure 4.6: Eigenvector centrality top-100 precision comparison on email-Enron. Higher is better. *Rank Degree* and *Random* have the best performance. *Forest Fire* and *K-Neighbor* do not perform well.

Eigenvector Centrality. Figure 4.6 presents the top-100 precision of eigenvector centrality on email-Enron. *Rank Degree* achieves the best performance because it retains edges connected to high-degree vertices. Although eigenvector centrality is not directly linked to degree, high-degree vertices have a higher probability of being directly or indirectly (via n-hop neighbors) connected to important vertices. In comparison, *Local Degree* performs worse than *Rank Degree* since it only considers the degree of immediate neighbors and may disconnect vertices from vital vertices located more than 1-hop away. *Random* shows strong performance due to its unbiased nature, which helps preserve relative ranking. Both *Forest Fire* and *K-Neighbor* under-perform in preserving eigenvector centrality.

Katz Centrality. Figure 4.7 illustrates the top-100 precision of Katz centrality on ego-Twitter. *Random* demonstrates the most effective performance. This is because *Random* proportionally maintains the number of edges relative to the original degree for all vertices. Thus, the graph's hop structure closely resembles its original state. Empirically, *K-Neighbor* and *ER-unweighted* also exhibit strong performance. *Local Degree* and *Rank Degree* do not perform well since they solely focus on degree, thereby only accounting for immediate

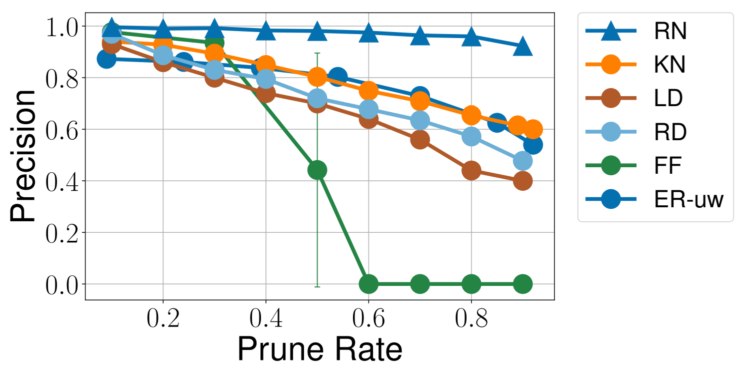


Figure 4.7: Katz centrality top-100 precision comparison of different sparsifiers on ego-Twitter. Higher is better. *Random* has the best performance. *Forest Fire* does not perform well.

neighbors. Therefore, vertices with low-degree immediate neighbors but high k-hop ($k > 1$) neighbors are severely penalized. Minor fluctuations in sparsifiers' relative performance on certain graphs can be attributed to the variation in the attenuation factor α. Overall, the performance is consistent across graphs.

In summary, *Local Degree*, *Rank Degree*, and *Random* consistently excel in centrality-related metrics. This is because *Local Degree* and *Rank Degree* retain edges connected to hub vertices, and centrality metrics seek important vertices in the graph, which often correspond to hub vertices. Conversely, *Random* maintains edges without bias, thus effectively preserving the relative vertex ranking.

4.3.4 Clustering Metrics

Number of Communities. I employ the widely recognized Louvain method [37] for community detection, assuming the number of communities is unknown, and use the number detected in the original graph as the ground truth. Figure 4.8 presents a comparison of community numbers on com-DBLP, with the green dashed line representing the ground truth; the closer to it, the better. As the prune rate increases, the graph becomes increasingly disconnected, and the number of communities consistently rises. *Local Degree* and *K-Neighbor* excel in maintaining the community number relatively close to the ground truth because it

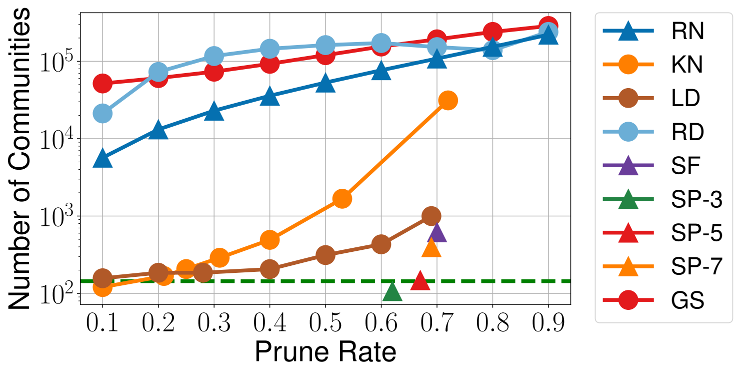
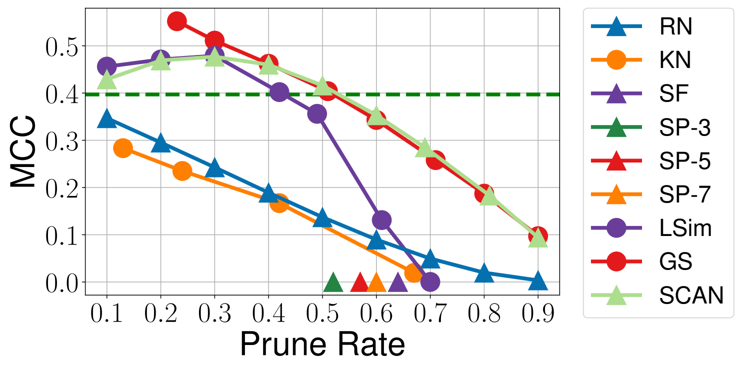


Figure 4.8: Number of communities comparison on com-DBLP. Closer to the green line is better. *Local Degree*, *Spanning Forest*, and *t-Spanners* have the best performance. *G-Spar*, *Rank Degree*, and *Random* do not perform well.

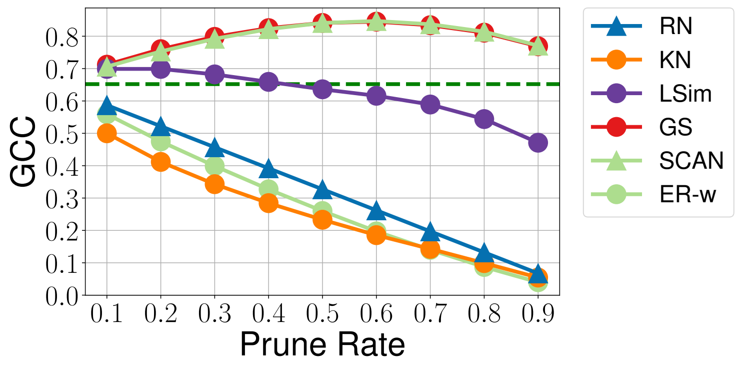
preserves connectivity. *Spanning Forest* and *t-Spanners* also demonstrate strong performance, surpassing *Local Degree* at equivalent prune rates, as they ensure connectivity remains identical to the original graph. Unlike *Local Degree*, *Rank Degree* struggles to preserve the community number because it sparsifies globally without guaranteeing connectivity preservation. In various graphs, *Local Degree*, *Spanning Forest*, and *t-Spanners* consistently outperform.

Clustering Coefficient. Figure 4.9 compares clustering coefficients on com-Amazon and human_gene2. I use the mean clustering coefficient (MCC) to evaluate the local clustering coefficient (LCC), as it represents the average LCC of all vertices. The green dashed lines indicate the MCC and GCC of the original graph. Generally, most sparsifiers exhibit decreasing MCC and GCC as the prune rate rises, with only *Local Similarity*, *SCAN*, and *G-Spar* exhibiting slight increases in MCC at lower prune rates. None of the sparsifiers demonstrate outstanding performance in preserving MCC and GCC, as they all degrade linearly with respect to the prune rate. *Spanning Forest* and *t-Spanners* consistently have an MCC of 0 due to the absence of loops in the graph. Clustering coefficient results vary across different graphs, with graph categories and directedness significantly impacting sparsifier performance. Overall, no sparsifier proves effective in preserving clustering coefficients.

Clustering F1 Similarity. Relying solely on the number of communities to evaluate



(a) Mean clustering coefficient.



(b) Global clustering coefficient.

Figure 4.9: Clustering coefficients comparison. Closer to the green line is better. (a) shows the MCC on com-Amazon (b) shows the GCC on human_gene2. No sparsifier is effective in preserving the clustering coefficient.

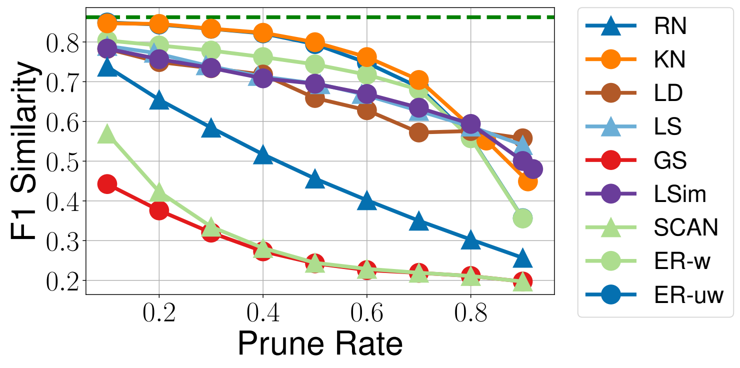


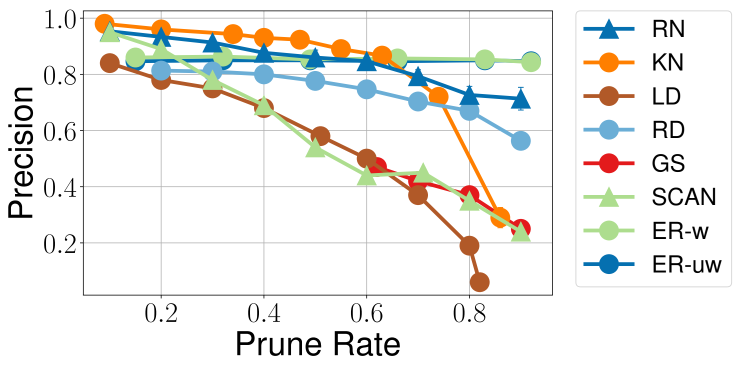
Figure 4.10: Clustering F1 similarity comparison of different sparsifiers on ca-HepPh. Higher is better. *ER-unweighted*, *ER-weighted*, *K-Neighbor*, *Local Degree*, *L-Spar*, and *Local Similarity* perform the best. *SCAN* and *G-Spar* underperform.

clustering quality is insufficient. Therefore, I employ the clustering F1 score to measure clustering similarity (see § 4.1.2.4). Figure 4.10 shows the clustering F1 similarity comparison on ca-HepPh, with F1 similarity ranging from 0 (worst) to 1 (best). The green dashed line represents the clustering F1 similarity when applying clustering algorithms twice on the original graph; it is not 1 due to the inherent randomness in the clustering algorithm. For all sparsifiers, F1 similarity decreases as the prune rate increases. *K-Neighbor* exhibits the best overall performance, while *Local Similarity*, *Local Degree*, and *L-Spar* also demonstrate strong results. These sparsifiers share a focus on local edges, and locally similar vertices are more likely to belong to the same community. Empirically, I also observe that *ER-weighted* and *ER-unweighted* perform well, potentially due to *ER*'s preservation of low-redundant edges, which are often crucial in clustering algorithms. In contrast, *G-Spar* and *SCAN* perform poorly in preserving clustering similarity. Across graphs, *K-Neighbor*, *Local Degree*, *Local Similarity*, *L-Spar*, *ER-unweighted*, and *ER-weighted* consistently rank as top performers, while *G-Spar* and *SCAN* persistently underperform.

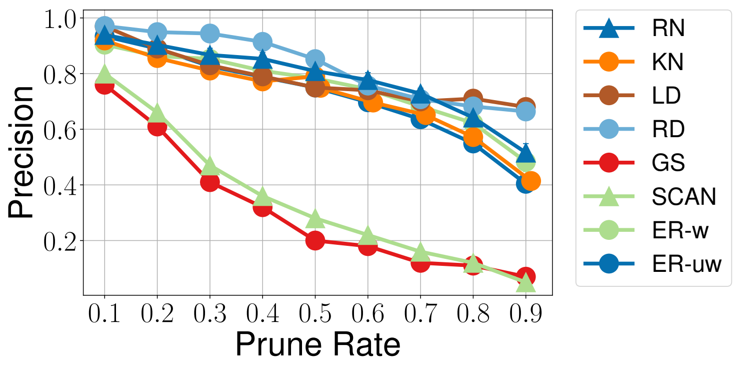
4.3.5 High-level Metrics

PageRank. Figures 4.11a and 4.11b present the top-100 precision of PageRank on web-Google and ego-Facebook, respectively. Note that web-Google is a directed graph and *ER* only supports undirected graphs. Thus, I symmetrize the graph before performing *ER*. Sparsifiers that work on directed graphs are applied directly.

As illustrated in Figure 4.11a, *ER-unweighted* and *ER-weighted* demonstrate high precision and consistency at various prune rates. On all web networks (web-NotreDame, web-BerkStan, web-Google, web-Stanford), the performance of *ER* remains similar in that precision is almost constant at different prune rates. However, *ER* does not always achieve the best performance at low prune rates. For some graphs, the precision of *ER* remains constant but at a lower level. This can be due to the symmetrizing process altering the original graph's information. The more symmetrical the original graph is, the less influence will be introduced. *K-Neighbor*



(a) PageRank Centrality on web-Google



(b) PageRank Centrality on ego-Facebook

Figure 4.11: PageRank centrality. Higher precision is better. (a) PageRank centrality on web-Google. *K-Neighbor* and *Random* perform the best at a low prune rate, *ER-weighted* and *ER-unweighted* perform the best at a high prune rate. *Local Degree*, *G-Spar*, and *SCAN* do not perform well. (b) PageRank centrality on ego-Facebook. *Rank Degree* has the best performance. *G-Spar* and *SCAN* underperform.

also shows good performance at low prune rates. In contrast, *G-Spar*, *SCAN*, and *Local Degree* fail to preserve PageRank effectively.

Figure 4.11b reveals *ER* sparsifier's performance on unweighted graphs, using ego-Facebook as an example. *Rank Degree*, *Local Degree*, *Random*, *K-Neighbor*, *ER-unweighted*, and *ER-weighted* all exhibit similar performance in preserving PageRank. *G-Spar* and *SCAN* continue to underperform. In comparison to directed graphs, *ER* no longer exhibits almost constant precision at varying prune rates on undirected graphs. *Rank Degree* and *K-Neighbor* consistently perform well on both directed and undirected graphs. *Local Degree* displays a significant discrepancy in performance between directed and undirected graphs, excelling in undirected graphs but consistently underperforming in directed ones. *G-Spar* and *SCAN* show poor performance consistently.

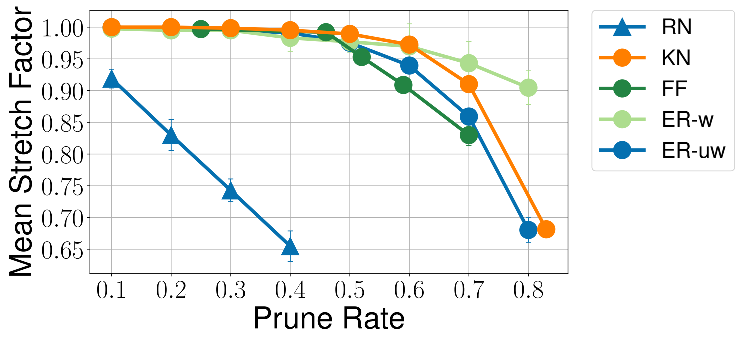
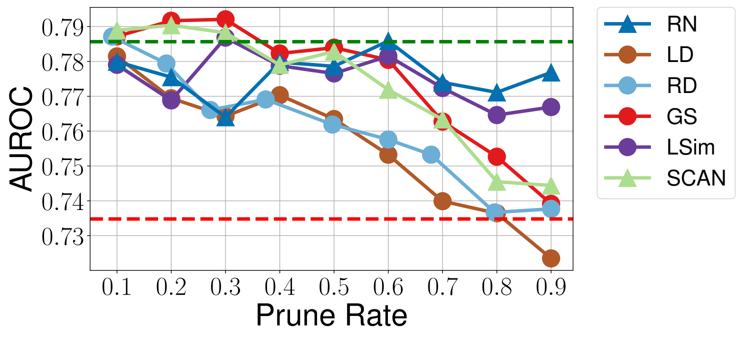


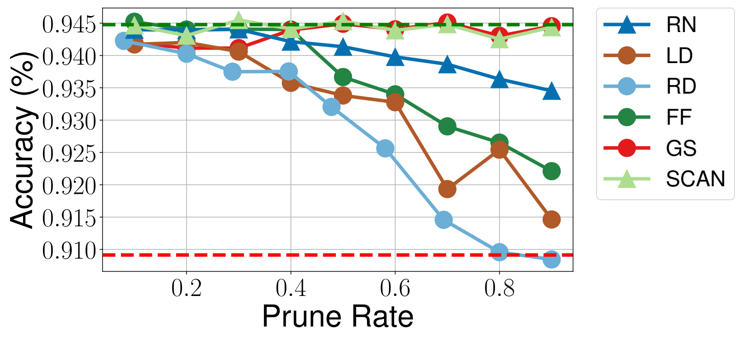
Figure 4.12: Adjusted Mean Stretch Factor for min-cut/max-flow with the constraint of acceptable unreachable ratio on ca-HepPh. Closer to 1 is better. *ER-weighted* has the best performance.

Min-cut/Max-flow. Figure 4.12 presents the mean stretch factor on ca-HepPh, with the constraint that the unreachable ratio remains $< 20\%$ higher than in the original graph. A mean stretch factor closer to 1 means better performance. *ER-weighted* shows the best performance. This can be attributed to *ER* being a spectral sparsifier, which preserves the spectral properties of graphs [132]. Min-cut/max-flow methods are also closely related to the graph spectrum. Flow-based graph partitioning [112] employs the Fiedler vector [61] (eigenvector corresponding to the second smallest eigenvalue of the graph Laplacian). One can intuitively think of *ER* as retaining high-resistance (low-redundant) edges in the graph, typically found in the critical (narrowest) section of the max-flow problem. *ER-weighted* significantly outperforms its unweighted counterpart *ER-unweighted*, as *ER-weighted* effectively compensates for the weights of other edges when removing edges. *K-Neighbor* and *Forest Fire* show good empirical performance as well. In contrast, *G-Spar* and *SCAN* underperform, and other sparsifiers exhibit similarly mediocre results. The outcomes for min-cut/max-flow are consistent across graphs, with *ER-weighted* as the top performer, followed by *K-Neighbor* and *Forest Fire*.

GNN. Figures 4.13a and 4.13b show the performance of sparsifiers on two distinct GNN



(a) GraphSAGE comparison of different sparsifiers on ogbn-proteins



(b) ClusterGCN comparison of different sparsifiers on Reddit

Figure 4.13: GNN comparison of different sparsifiers. Higher AUROC and accuracy are better. The green line represents the inference results on the model trained by the full graph. The red line represents the inference results on the model trained with no graph (MLP only). (a) is evaluated with the GraphSAGE on ogbn-proteins. (b) is evaluated with the ClusterGCN on Reddit.

models. GNN performance is measured using AUROC and vertex classification accuracy. The green dashed line represents performance on the full graph, while the red dashed line represents performance on the empty graph (a graph with no edges). I include the empty graph to demonstrate the performance of GNN models based solely on vertex features without any graph structural information. On the GraphSAGE model, *Random* and *Local Similarity* perform the best; *G-Spar* and *SCAN* perform well at low prune rates but deteriorate rapidly at higher rates. However, on the CluterGCN model, *G-Spar* and *SCAN* perform well at all prune rates. *Local Degree* and *Rank Degree* consistently underperform compared to other sparsifiers on both models. Due to the complexity of GNN algorithms, it is challenging to draw straightforward conclusions. Overall, the performance of sparsifiers on GNNs differs from model to model, which may be due to the inherent characteristics of GNN workloads.

4.3.6 Sparsification Time

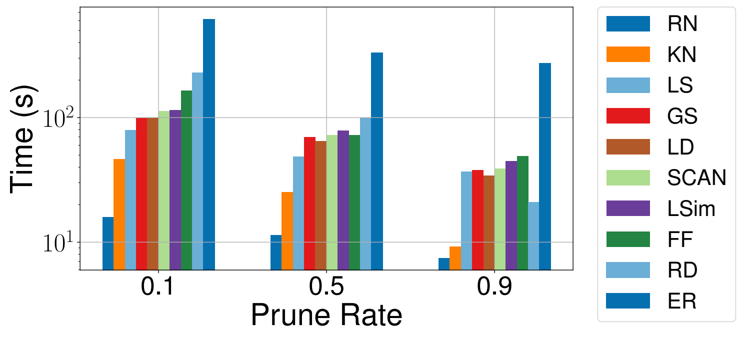


Figure 4.14: Sparsification time comparison on ogbn-proteins

Figure 4.14 shows the sparsification time of different sparsifiers at different prune levels. For all sparsifiers, sparsification time decreases as the prune rate increases; this is expected because the higher the prune rate, the fewer edges need to be picked. Across sparsifiers, the sparsification time is also different. *Random* and *K-Neighbor* are the sparsifiers with the lowest overhead due to their low algorithmic complexity. *L-Spar*, *G-Spar*, *Local Degree*, *SCAN*, *Local Similarity*, *Forest Fire*, and *Rank Degree* show similar latency. *ER* is the most complex algorithm. In the figure, the time for *ER* is only for sampling. I do not include the computation time of the effective resistance because it is a one-time cost. The computation of effective resistance takes 990 seconds for ogbn-proteins. And the execution time of *ER* is approximately an order of magnitude higher than that of other sparsifiers. However, depending on the application, a high-cost sparsifier like *ER* can still be useful if it preserves the desired graph properties and the sparsification overhead is less than the time that can be saved in performing the downstream task on the sparsified graph.

4.3.7 Summary of Results and Insights

Overall, The performance of all sparsifiers degrades as the prune rate increases. Usually, I observe that the relative performance of sparsifiers is consistent across prune rates, meaning superior sparsifiers at low prune rates will remain superior at high prune rates, and the performance gap between the superiors and inferiors will be larger. On some occasions, the performance of a sparsifier has an elbow point, beyond which the performance drops sharply. This is because some sparsifiers cannot maintain certain properties beyond the elbow prune rate. For example, in figure 4.6, the performance of *Local Degree* dropped abruptly when increasing the prune rate from 0.8 to 0.9 because the number of edges is so low that it cannot maintain the graph connectivity anymore.

To make sparsification effective, the selection of the sparsification algorithm should preserve the graph property/properties on which the downstream application is based. I summarize what each sparsifier preserves as below.

- ***Random***: preserves relative (distribution-based or ranking-based) properties, for example, degree distribution and top centrality rankings. It struggles to preserve absolute (valued-based) properties, such as the number of communities, clustering coefficient, and min-cut/max-flow.

- ***K-Neighbor***, ***Spanning Forest***, and ***t-Spanners***: preserves graph connectivity; keeps pair unreachable ratio and vertex isolated ratio low.

- ***Rank Degree*** and ***Local Degree***: preserves graph connectivity and edges to high-degree vertices (hub vertices). Perform well on distance metrics (APSP, eccentricity, diameter) and centrality metrics.

- ***Forest Fire***: simulates the evolution of graphs and does not strictly stick to the original graph. Empirically, it does not excel at any of the metrics evaluated.

- ***G-Spar*** and ***SCAN***: Empirically perform well in preserving ClusterGCN accuracy.
- ***L-Spar*** and ***Local Similarity***: preserves the edge to similar vertices, thus preserving clustering similarity.
- ***ER***: preserves the spectral properties of the graph, specifically the quadratic form of the graph Laplacian. It performs well in preserving min-cut/max-flow results.

4.4 Related Work

ML-based sparsifiers are a group of sparsifiers that use machine learning-related techniques to sparsify graphs. SparRL [146] proposes a graph sparsification framework enabled by deep reinforcement learning. NeuralSparse [159] presents a supervised graph sparsification technique to improve performance in graph neural networks (GNN). Instead of focusing on saving execution time by performing graph sparsification, NeuralSparse aims to remove task-irrelevant edges from the graph, thus improving the accuracy of the downstream GNNs. DropEdge [123] presents a method very close to the random sparsifier but samples a random set of edges for each training epoch in a graph convolutional network (GCN); the goal is both to reduce message passing overhead and reduce over-fitting with the full graph input.

4.5 Conclusion

This study provides a comprehensive evaluation of 12 graph sparsification algorithms, analyzing their performance in preserving 16 essential graph metrics across 14 real-world graphs with diverse characteristics. The findings revealed that no single sparsifier excels in preserving all graph properties, and it is important to select appropriate sparsification algorithms based on the downstream task. This study contributes to the broader understanding of graph sparsification algorithms, and I provided insights to guide future work in effectively integrating graph sparsification into graph algorithms to optimize computational efficiency without significantly compromising output quality. New applications can be broken down into one or more graph properties and sparsification algorithms can be chosen with the heuristic elaborated in this work. The open-source framework implemented the 12 sparsification algorithms and 16 graph properties evaluated in this work and provides an easy-to-use interface to bridge the two parts. The framework offers a valuable resource for ongoing evaluations of emerging sparsification algorithms, graph metrics, and growing graph data.

CHAPTER 5

A Power Efficient GCN Accelerator with Multiple Dataflows

With the rapid development of deep learning in the last decade, neural networks are now widely adopted in many applications such as image recognition [88], object detection [121], and machine translation [28]. However, traditional neural networks are limited to handling Euclidean data [70] such as one-dimensional (1D) text streams and two-dimensional (2D) images, and do not generalize well on non-Euclidean data such as graphs [126] and manifolds [44]. Graph Neural Networks (GNNs) take a further step to explore graph-structured data. Compared to Euclidean data, graphs have better expressiveness so that GNNs can learn from the latent information of nodes and the connections between nodes. This extends the application scope of deep learning to a broader range of applications [161, 138] such as natural science [31].

Among different types of GNNs, Graph Convolutional Networks (GCN) is one of the most prominent algorithms [86]. Motivated by Convolutional Neural Networks (CNNs), GCNs generalize convolution to graph-structured data. GCN solves CNN's limitation of only being applicable to regular Euclidean data [161]. GCN is composed of two phases - aggregation and combination. Aggregation collects information from neighboring nodes and edges. It works on the input graph and often suffers from irregular memory accesses. Combination uses multi-layer perceptron (MLP) to further process the aggregated results by multiplying them with the trained weight matrices, which have regular memory accesses. GCN has many variations [162, 74, 47], and has developed into a big algorithm family.

Due to the inherent irregular memory accesses in GCNs, CPUs and GPUs cannot make good use of their massive computing resources. Thus, several works are proposed to enhance resource utilization. HyGCN [152] uses dedicated processing engines for the aggregation and combination phases to alleviate the memory irregularity in the aggregation phase while exploiting the regularity in the combination phase. EnGN [94] applies edge reorganization to compress the sparse adjacency matrix and uses a degree-aware vertex cache to store hot nodes. AWB-GCN [67] observes that real-world graph datasets have power-law distributions, and it optimizes Processing Elements' (PE) utilization by performing workload balancing among PEs. ReGNN [45] dynamically computes and reuses the aggregated features of redundant neighbor sets to reduce memory accesses. GCoD [155] and I-GCN [68] try to improve graph regularity by rearranging the adjacency matrix permutation.

Although prior works performed various optimizations to enhance resource utilization, they use a fixed dataflow and are not flexible enough to run different GCNs efficiently. Firstly, real-world datasets span a wide range of sizes and densities. They require different aggregation dataflows based on the input dataset characteristics. Secondly, the order of aggregation and combination phases can be altered when the aggregation function is linear (see §5.1). While this results in better performance, the order must be respected with non-linear aggregation functions. Thus, it is important to support different dataflows and orderings to achieve both efficiency and flexibility.

In this paper, I make the following contributions:

- I perform quantitative and qualitative analysis on three widely used GCN algorithms: vanilla GCN, GS-mean, and GS-max with five real-world datasets. I show that the GCN algorithms and input dataset characteristics affect the choice of phase ordering and dataflow for the best performance.

- I propose PEDAL, an accelerator for GCN inference. PEDAL features three dataflows and supports both orderings of the aggregation and combination phases, achieving both efficiency and flexibility.
- I train a decision tree with 400 synthetic datasets to automatically and accurately choose the best dataflow and phase ordering for a GCN algorithm.
- I evaluate the performance of PEDAL using a cycle-accurate simulator and measure its power and area using RTL synthesis. I show PEDAL achieves 144.5×, 9.36×, and 2.55× speedup compared to CPU, GPU, and HyGCN, and also 8856×, 1606×, 8.4× and 1.78× better power efficiency compared to CPU, GPU, HyGCN, and EnGN.

To the best of my knowledge, this is the **first work** that explores different dataflows and execution orders for the GCN workload and exploits this knowledge to choose the best dataflow according to the input graph and GCN algorithm.

5.1 Background

GCN uses a convolutional layer to collect information for training and inference. While CNN performs convolution on Euclidean data, GCN takes a graph (non-Euclidean data) as the input. The input graph's nodes (or edges) have a vector of features containing information for training and inference. For example, in a social network, each node represents a user with features like age, gender, etc [83].

Unlike the Euclidean data, where neighbors are spatially close in the memory (multi-dimensional matrices), neighbors of graph-structured data are located apart. This results in irregular memory accesses and imposes new challenges on the processors. This section gives a brief background on GCN. Table 5.1 lists the notations and acronyms used in this paper.

Category	Notation & Acronyms	Note
Dimension	N	Number of nodes in the graph
	F1	The feature dimension for the 1st layer
	F2	The feature dimension for the 2nd layer
Matrix	A	Adjacency matrix, dimension N x N
	X	Feature matrix, dimension N x F1, each row is a Feature vector (transposed) for the corresponding node.
	W	Weight matrix, dimension F1 x F2
Acronyms	**AC**	Short for Aggregation+Combination order
	CA	short for Combination+Aggregation order
	IP-AC	Short for Inner-Product, AC order dataflow
	IP-CA	Short for Inner-Product, CA order dataflow
	RW-AC	Short for Row-Wise, AC order dataflow
	RW-CA	Short for Row-Wise, CA order dataflow
Others	$\mathcal{N}(v)$	Neighbors of node v
	$d(M)$	density of matrix M
	$NNZ(M)$	number of non-zeroes of matrix M

Table 5.1: Notation and acronyms used in this paper

5.1.1 GCN Model

A GCN is composed of multiple layers. In each layer, feature information from the neighbors is aggregated (aggregation phase) and multiplied by a weight matrix (combination phase) and becomes the feature information for the next layer. The aggregation function can be sum, mean, max, min, Long Short Term Memory (LSTM), or other more complicated functions. The combination phase uses an MLP layer with a trainable weight matrix to transform the aggregated features and reduce the dimension of the output features.

Each layer in GCN propagates node or edge information to its one-hop neighborhood. Thus, an N-layer network effectively propagates features to its N-hop neighbors. Usually, a couple of layers is enough as the information from closer neighbors is more important than remote ones. Figure 5.1 shows an example of a vanilla GCN layer. Other variances of GCN algorithms have a similar model but with different aggregation functions.

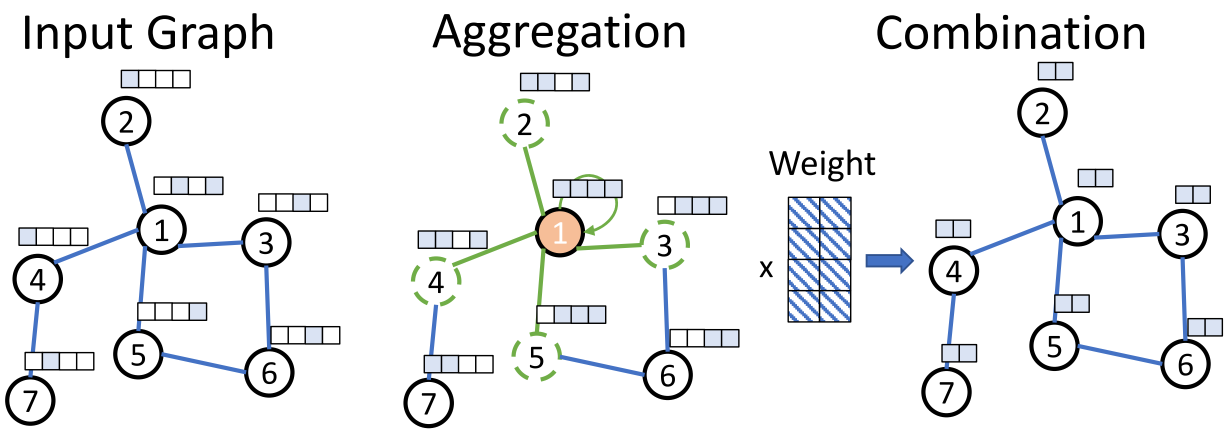


Figure 5.1: An example of the Vanilla GCN layer with N=7 nodes, and F1=4 and F2=2 features. White cells are zeros. A self-loop retains the feature vector of the node for aggregation.

5.1.2 Rich Diversity in GCN Models

I observe that state-of-the-art GCN models and popular input datasets come with diverse aggregation functions and input feature densities. An efficient GCN processor must be flexible enough to exploit different characteristics.

Table 5.2 shows the aggregation and combination functions of the three GCN models used in this work. I call the original GCN proposed in [86] vanilla GCN. Vanilla GCN takes the mean value of all neighboring nodes' features and multiplies the aggregated results with a weight matrix through an MLP layer. GraphSAGE [74] introduced neighbor sampling to vanilla GCN. GS-mean and GS-max are two variations of GraphSAGE that use *mean* and *max* for aggregation functions, respectively. In this work, I use a sampling number of 25 to be consistent with the original GraphSAGE algorithm [74].

Algorithm	Aggregation	Combination
Vanilla GCN [86]	$B = mean(\mathcal{N}(H^l))$	$X = ReLU(BW)$
GS-mean [74]	$B = mean(\mathcal{N}(H^l))$	$\sigma(W_l \cdot Concat(B, h^l))$
GS-max [74]	$B = max_{j \in N(h^l)} \sigma(W_l^1 \cdot h_j^l)$	$\sigma(W_l^2 \cdot Concat(B, h^l))$

Table 5.2: Aggregation and combination operations of GCN models [23].

I observe a variety of input datasets with different input sizes and feature matrix densities.

Table 5.3 shows the information of these datasets. Cora and CiteSeer have relatively small input graphs with a sparse feature matrix. PubMed has a medium-sized input graph with a 10% dense feature matrix, and Reddit and Ogbn-products have a large input graph with a dense matrix.

Dataset Name	#Vertices	#Edges	F1	*d(X)*	X size
Cora (CR) [86]	2708	10566	1433	1.27%	385KB
CiteSeer (CS) [86]	3327	9104	3703	0.85%	820KB
PubMed (PB) [86]	19717	88648	500	10%	7.5MB
Reddit (RD) [74]	232965	114.6M	602	100%	535MB
Ogbn-Prod (OP) [75]	2449029	123.7M	100	99%	925MB

Table 5.3: Datasets information. All datasets contain a single graph; all graphs are unweighted, undirected, and symmetrical. Non-zeros in the feature matrix are stored in 32-bit fixed point.

5.1.3 Phase Orderings

The original GCN model performs the aggregation phase before the combination phase. This is similar to CNNs, where convolution is performed before feeding the results to fully connected layers. However, prior works [67, 94] have observed that reordering the phases - performing combination before aggregation - can significantly reduce the operation count. This is because the combination reduces the feature dimension, and by executing the combination phase first, the matrix multiplication in the aggregation is performed on a smaller dimension. In this work, I refer to the original order of performing **A**ggregation phase before **C**ombination phase as the **AC** order and the reverse order as the **CA** order.

Figure 5.2 shows the total number of arithmetic operations for **AC** order and **CA** order for vanilla GCN and GraphSAGE with different datasets. On average, **CA** order achieves 93% operation count reduction for GCN and GS-mean. **CA** order does not apply to GS-max, which uses a non-linear function for aggregation. Lower operation count makes **CA** order preferable. However, it is only applicable when the alternation of the order does not affect the correctness of the output.

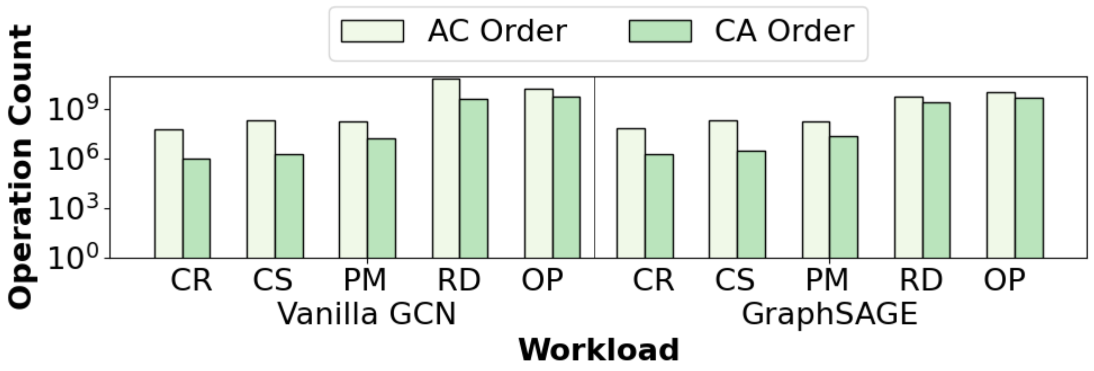


Figure 5.2: Operation count for vanilla GCN, GS-mean, and GS-max models in **AC** and **CA** orders. **CA** order does not apply to GS-max.

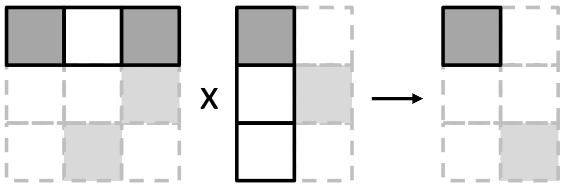
To ensure correctness, the aggregation function must be linear, meaning that $Agg(a, b) \times c == Agg(a \times c, b \times c)$, where $\times c$ is the combination operation. For example, addition and mean functions are linear operations because $(a + b) \times c == (a \times c + b \times c)$, while max and min functions are not linear because $max(a, b) \times c! = max(a \times c, b \times c)$.

5.1.4 Aggregation Dataflow

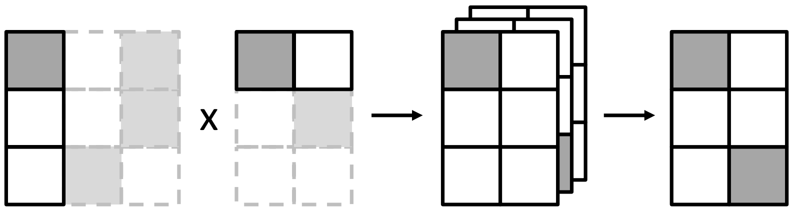
The aggregation phase is essentially the multiplication of the adjacency matrix and the feature matrix. There are three widely used matrix multiplication methods: inner-product (IP), outer-product (OP), and row-wise (RW) as the candidates. Figure 5.3 shows inner-product, outer-product, and row-wise matrix-matrix multiplication.

Inner-product performs element-wise operation with a row and a column of two matrices on matching indices. It exploits output data reuse because each output is written only once, but suffers from bad input reuse due to repeated reading of the second matrix for each row in the first matrix. Besides, for very sparse matrices, the odds of having matching indices are very low and can become a major overhead.

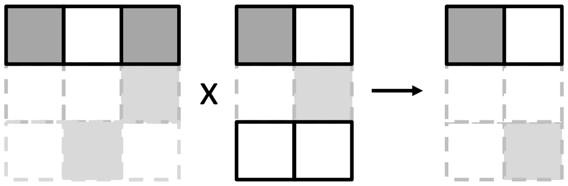
Outer-product performs pair-wise multiplication with a column and a row of two matrices and generates a partial matrix of the same size as the final result. Outer-product enjoys input data reuse because both input matrices are read only once, but it generates N partial result matrices and needs element-wise merging of all the partial matrices to get the final result.



(a) Inner Product



(b) Outer Product



(c) Row-wise

Figure 5.3: Inner product, outer product, and row-wise dataflows.

Row-wise takes a row from the first matrix, uses its indices to retrieve the corresponding rows from the second matrix, and reduces multiple rows to one using the aggregation function. Row-wise has good output data reuse and avoids the index matching overhead in the inner-product. The downside of row-wise is bad input data reuse as the second matrix will be repetitively read, and its access pattern depends on the first matrix, causing irregular memory accesses.

Out of the three aggregation dataflows, outer-product is unsuitable as it requires merging partial results to get final results, which impedes the pipelining of the aggregation and combination phases. The choice between inner-product and row-wise is explained in §5.2.3.

General-purpose architectures are ill-suited for efficiently executing GCN workloads, and prior accelerators cannot adapt to a large design space of GCN workloads. Therefore, designing an accelerator architecture that can support diverse GCN models, different phase orderings, and aggregation dataflows is crucial to optimize performance and power efficiency.

5.2 Proposed Design

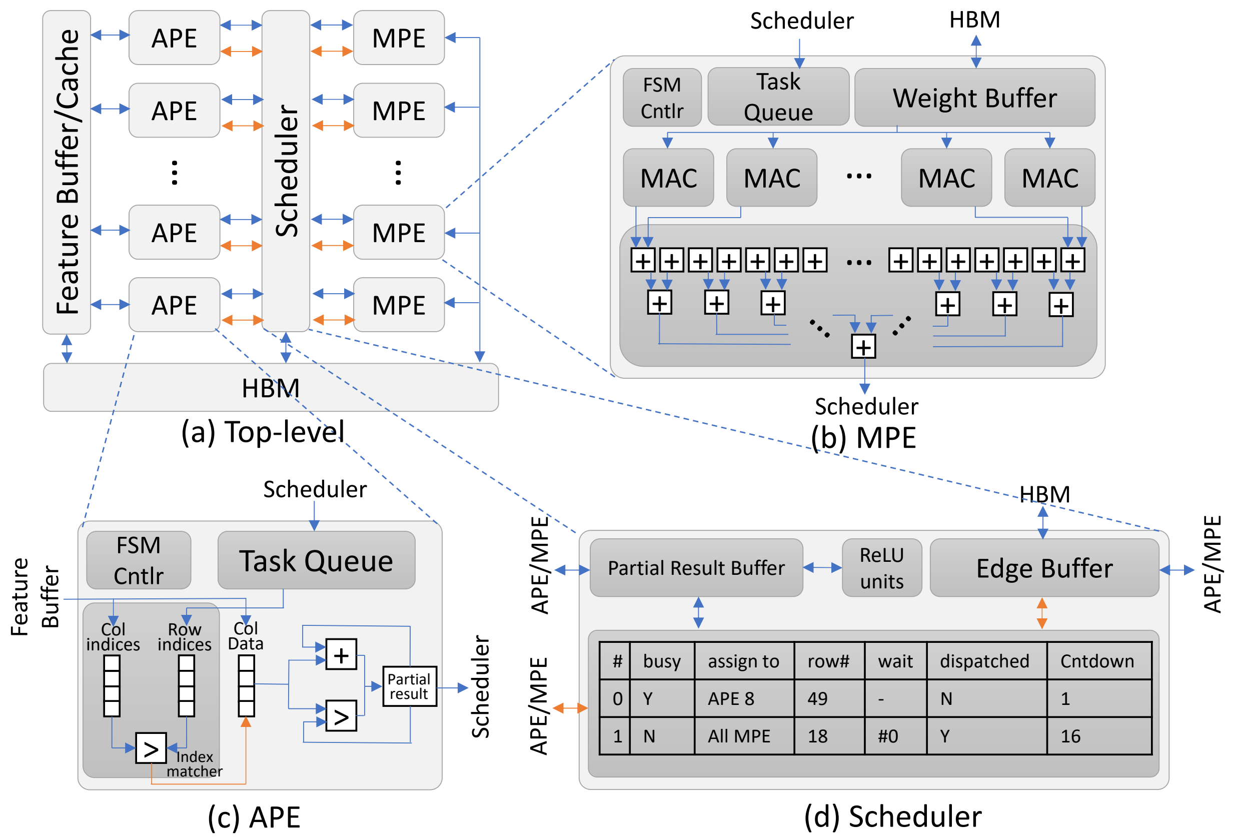
5.2.1 PEDAL Architecture

In this section, I present the architecture design of PEDAL.

Top-level. Figure 5.4(a) shows the top-level PEDAL architecture. PEDAL has two types of Processing Elements (PEs) - Aggregation Processing Elements (APEs) for aggregation and MAC Processing Elements (MPEs) for combination. PEDAL has 32 APEs and 16 MPEs, an 8 MB feature buffer, a scheduler, and a backend HBM memory system. APEs and MPEs are connected to the scheduler, which controls the task assignment and intermediate result movement among APEs and MPEs. The feature buffer is connected to all APEs. It has 32 banks and can be used as a user-managed scratchpad or a user-transparent cache. The scheduler has a 2 MB edge buffer and a 512 KB partial result buffer. Each MPE has a 32 KB weight buffer. All MPEs, the scheduler, and the feature buffer are connected to the backend HBM memory system.

APE. APEs are used to execute aggregation operations. Figure 5.4(c) shows the architecture of an APE. It has a task queue to receive tasks from the scheduler, a Finite-State Machine (FSM) controller to execute the tasks, an index matcher that matches row indices with column indices for the inner product, and an accumulator and a comparator. In this work, the computing units in APEs are simplified to only an adder and a comparator for addition and max/min operations. This minimizes the area and power consumption while allowing APEs to perform a handful of the most popular algorithms like GCN, GS-mean, and GS-max. Other computing units can be added to APEs if desired to enable other operations.

Index matcher. The index matcher finds the intersection of two sorted arrays. It keeps two circular queues of 32 Column and Row indexes. A naive implementation compares the top element of queues and returns them if they are equal. Otherwise, it pops the smaller one.



#	busy	assign to	row#	wait	dispatched	Cntdown
0	Y	APE 8	49	-	N	1
1	N	All MPE	18	#0	Y	16

Figure 5.4: PEDAL architecture. (a) is the top-level architecture, (c), (b), and (d) are the details inside APE, MPE, and the scheduler module, respectively. The blue lines in the figures are the data path, and the orange lines are the control path.

In the worst case, this implementation needs to pop all elements of the queues sequentially. To decrease this overhead, the index matcher is equipped with 2×8 comparators. They compare the top elements with the 8 top indices of the other queues. In one cycle, the index matcher pops as many indexes from one queue as its top element becomes smaller than or equal to the other. Compared to the naive design, this design increases the performance by 3.97× while only adding 6% area overhead for 16 parallel comparators.

MPE. MPEs are used to execute the combination phase, which is the matrix multiplication of the feature and the weight matrices. Figure 5.4(b) shows the architecture of an MPE. It contains a task queue that receives jobs from the scheduler and an FSM Controller for controlling the execution; instead of accessing a unified weight buffer for all MPEs, each

MPE has a private weight buffer. The columns of the weight matrix are evenly distributed to MPEs, and each MPE will compute with the assigned portion of the weight matrix. Each MPE has 64 Multiply-Accumulate (MAC) units and a hierarchical adder tree with 63 adders to reduce the MAC results. Finally, the result is sent back to the partial result buffer in the scheduler.

Scheduler. The scheduler is responsible for assigning tasks to APEs and MPEs and keeping track of the status of each task. For example, tasks assigned to multiple APEs or MPEs can retire only when all PEs finish. The scheduler also monitors the dependencies between aggregation and combination phases. It only dispatches tasks that have no outstanding dependencies. Finally, the scheduler takes care of the data movement between APEs and MPEs when the results of one phase are needed in another phase. Figure 5.4(d) shows the architecture of the scheduler. It has an edge buffer for the adjacency matrix, a partial result buffer for the intermediate results from APEs and MPEs, and a schedule table that keeps track of the status of each task.

5.2.2 PEDAL Dataflows

Decoupling Aggregation and MAC PEs enables PEDAL to support diverse dataflows. This work features two ways of performing the aggregation phase: Inner-product (IP) and Row-Wise (RW), as well as two different computation orders: **AC** order and **CA** order as discussed in § 5.1. It gives us four different dataflows, namely **IP-AC**, **IP-CA**, **RW-AC**, and **RW-CA**. The four dataflows are described in detail below:

IP-AC. In **IP-AC**, the feature matrix is assigned to APEs column-wisely. Each APE is equipped with an equal-sized portion of the feature buffer (256KB). When the feature matrix is too big to be loaded into the feature buffer, it will be split into chunks of columns, and PEDAL loads the next chunk once the previous one is done. I assign as many columns as possible to fill up the feature buffer of APEs, eliminating workload imbalance from uneven distribution of non-zeros. The weight matrix is dense, so I assign an equal number of columns to each MPE.

Each row of the adjacency matrix is an aggregation task, and each row of the aggregated feature matrix is a combination task. The scheduler is responsible for scheduling, dispatching, tracking, and retiring tasks. Each aggregation task is broadcast to all APEs, and each APE will perform aggregation on the adjacency matrix row with the assigned feature matrix columns using inner-product. Once an APE finishes a task, it will send the task id and partial results to the scheduler. When all APEs finish a task, the scheduler retires the aggregation task and dispatches the corresponding combination task to MPEs. MPEs perform inner-product with the rows of the aggregated feature matrix and columns of the weight matrix.

IP-CA. **IP-CA** reverses the order of aggregation and combination to reduce the operation count. However, performing in CA order leads to a crucial issue: the combination phase generates the intermediate matrix row by row, while aggregation in the inner-product requires all rows indicated by the adjacency matrix at once, which will not be available at the time needed. Thus, I forfeit the **IP-CA** dataflow as it impedes the pipelining of aggregation and combination and hurts the performance.

RW-AC. **RW-AC** is an alternative way of performing GCN algorithms in AC order. In this dataflow, the feature buffer is used as a unified cache that is transparent to users. Similar to **IP-AC**, each row of the adjacency matrix is an aggregation task, but instead of broadcasting to all APEs, row-wise assigns each task to one APE. The APE retrieves the rows from the feature matrix based on the column indices of non-zeros in the adjacency matrix row. Each APE works independently from the other APEs and receives a new task upon finishing one, thus dynamically achieving workload balancing. The combination phase is the same as in **IP-AC** and is pipelined with the aggregation phase.

RW-CA. **RW-CA** performs the combination phase first to shrink the feature dimension and then performs the aggregation in a row-wise manner. While **IP-AC** and **RW-AC** dataflow *pull* the neighboring nodes' features, **RW-CA** *pushes* the feature of a node to all its neighbors. This is because the combination phase generates the intermediate feature matrix row by row, and it is not feasible to *pull* features from neighbors as they may not be ready yet. In **RW-CA**, the combination is performed the same way as in **IP-AC** and **RW-AC**. When a row of the intermediate feature matrix is generated, it is broadcast to all APEs. Each row of the adjacency matrix is an aggregation task, and it is evenly split into slices and assigned to all APEs. Each APE aggregates the feature to the slice assigned, so there is no memory contention across APEs. The feature buffer is evenly allocated to each APE (256KB) and used as a cache.

5.2.3 Choosing the Right Dataflow

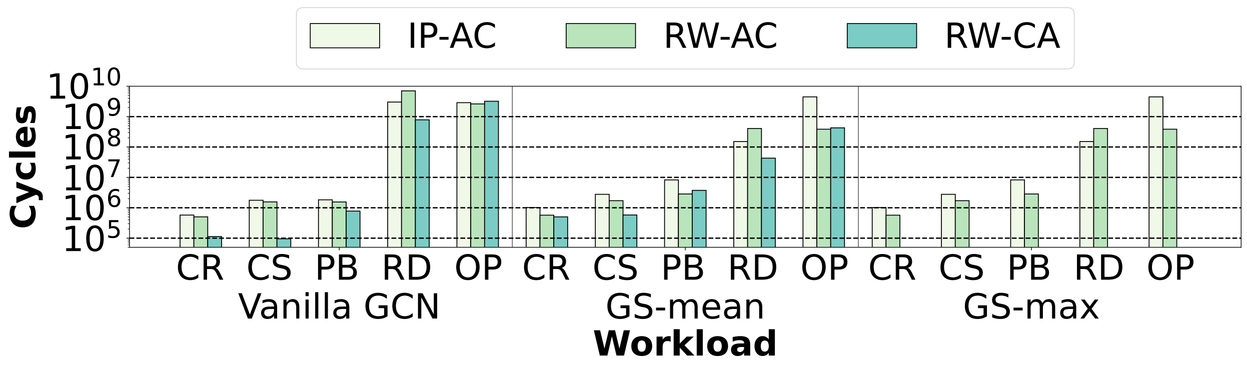


Figure 5.5: Performance of **IP-AC**, **RW-AC**, and **RW-CA**.

Figure 5.5 shows the performance of each dataflow for vanilla GCN, GS-mean, and GS-max with five real-world datasets. The choice of dataflow (**IP-AC**, **RW-AC**, or **RW-CA**) significantly affects the execution time. The best dataflow must be picked for each GCN algorithm and dataset pair. A simple approach is to find the number of arithmetic operations and compare them for different dataflows. However, this approach does not take memory stalls into account. For example, while the operation count of vanilla GCN and Reddit dataset is the same for **IP-AC** and **RW-AC** dataflows, the high cache miss rate of **RW-AC** dataflow results in a longer execution time compared to **IP-AC**. The simple approach chooses the right dataflow in only 73% of the evaluated GCN model and input dataset pairs. A better approach is needed to make educated decisions based on the dataset characteristics and the GCN model. I use N, NNZ(A), NNZ(X), and F1 as dataset characteristics, which are the input dataset metadata.

With the complexity of so many dataset parameters, GCN variances, and execution orders, a decision tree is needed to choose the best dataflow. I created 400 synthetic datasets where N ranges from 1K to 1M, NNZ(A) from 2K to 200M, F1 from 100 to 3K, and NNZ(X) from 1K to 3B. I pick these parameters because they reflect the sizes and densities of the input graphs and input features. These ranges are large enough to cover all the real-world datasets evaluated in this paper. Besides, the synthetic datasets are generated such that non-zeros in adjacency matrices have power-law distribution, and non-zeros in feature matrices have Gaussian distribution based on observations from the real-world datasets. I build a decision tree using scikit-learn [115], which uses an optimized version of CART (Classification and Regression Trees). I use the synthetic datasets to train the decision tree and use it to predict the best dataflow on real-world datasets.

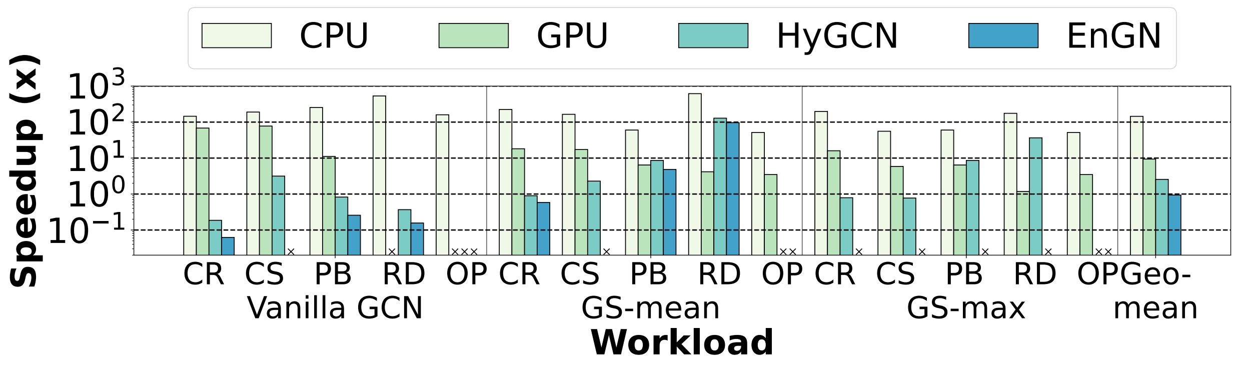
	Compute Unit	On-chip Memory	Off-chip Memory	Area(mm^2)	Power(W)
CPU	80 cores @ 2.1GHz	96MB	256 GB/s DDR4	- (14nm)	125
GPU	10496 Shading Units @ 1.4GHz	16.25MB	936.2 GB/s	628 (8nm)	350
HyGCN	16 SIMD cores @ 1GHz and 32x128 systolic array	22.1MB	256 GB/s HBM	7.8 (12nm)	6.7
EnGN	128x16 arrays @ 1GHz	1.6MB	256 GB/s HBM	3.54 (14nm)	3.87
PEDAL	32 APEs and 16 MPEs @ 1GHz	11MB	256 GB/s HBM	4.05 (12nm)	2.04

Table 5.4: Architecture configuration comparison of CPU, GPU, HyGCN, EnGN, AWB-GCN and PEDAL

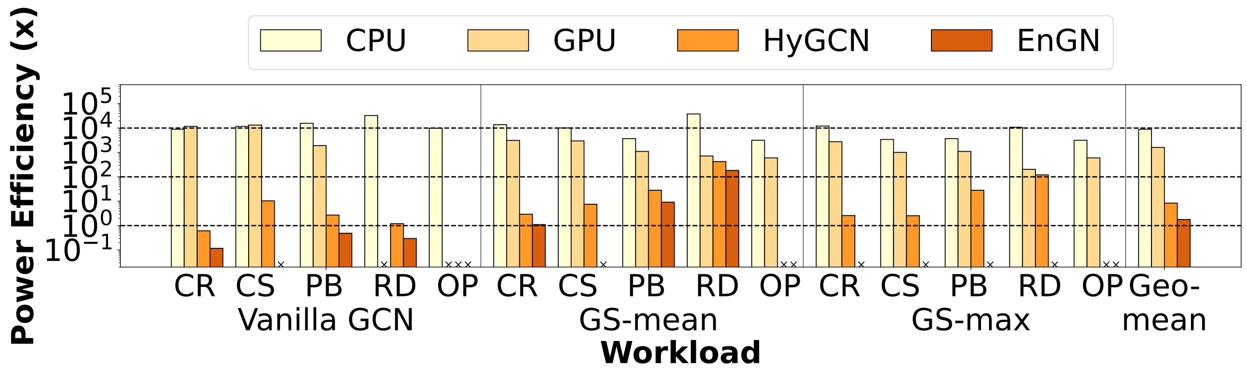
5.3 Evaluation

5.3.1 Experimental Setup

Baseline. I evaluate the CPU performance on Intel Xeon Gold 6230 CPU, and GPU performance on NVIDIA GPU with Ampere architecture. I implement the baseline on the state-of-the-art PyTorch Geometric [60] library. I also compare PEDAL with two prior GNN accelerators: HyGCN [152] and EnGN [94] using the reported performance numbers.



(a) Speedup.



(b) Power efficiency.

Figure 5.6: Speedup and power efficiency of PEDAL compared to CPU, GPU, HyGCN and EnGN. Power efficiency is measured by power-delay product. × markers mean missing data points due to GPU Out-Of-Memory or prior accelerators not reporting for some datasets or not supporting some GCN models.

Table 5.4 shows the configurations of PEDAL and baseline architectures.

PEDAL simulation. I build a cycle-accurate simulator in Python and C++ to measure the computation cycles of PEDAL. The simulator is event-based and controlled by a state machine to enforce dependencies. The memory access trace is recorded and fed to Ramulator [84] for memory access latency. Ramulator includes both cache and HBM memory as a hierarchical memory system. I implement the design in RTL and use Design Compiler to synthesize with a commercial 12nm CMOS library at 1GHz clock frequency. I use eDRAM for on-chip memory of PEDAL, HyGCN, and EnGN, and analyze with CACTI [107].

GCN algorithms and datasets. I evaluate the vanilla GCN algorithm and two variations of GraphSAGE: GS-mean and GS-max in this work. The details of the algorithms can be found in Table 5.2. Table 5.3 shows the datasets used in this work. I cover graph size from small to large, and with the feature matrix from sparse to dense to thoroughly compare PEDAL and prior works. I use the same hidden dimension (128) as in HyGCN for layer 1.

5.3.2 Decision Tree Accuracy

I used 80% of the synthetic datasets to train the decision tree and test on the remaining 20% and achieved 90% accuracy. Then, I apply the decision tree to the real-world datasets, and it selects the best dataflow with 93.3% accuracy. The only mistake happens on the vanilla GCN and OP dataset in which the mispredicted dataflow (**IP-AC**) is only 9% slower than the best dataflow (**RW-AC**). I calculate the execution time ratio between the decision tree selected dataflow and the best dataflow on the synthetic test set. The average ratio is 1.047, meaning that the decision tree selected dataflow has an execution time expectation less than 5% higher than the best dataflow.

5.3.3 Speedup and Power Efficiency

Figure 5.6a shows the performance of PEDAL compared to CPU, GPU, and prior accelerators. The best dataflow is used for each GCN algorithm and dataset pair.

On average, PEDAL outperforms CPU and GPU by 144.5× and 9.36×. Compared to prior accelerators, despite having less computing resources, PEDAL achieves 2.55× speedup over HyGCN. Compared to EnGN, PEDAL also supports non-linear aggregation functions (e.g., GS-max) while achieving similar performance for linear aggregation functions.

Compared to prior works, where thousands of PEs are used for better performance, PEDAL uses only 32 APEs and 16 MPEs to achieve similar or better performance in most cases. Figure 5.6b shows the power efficiency of PEDAL. Power efficiency is measured using the power-delay product. On average, PEDAL achieves 8856×, 1606×, 8.4× and 1.78× better power efficiency than CPU, GPU, HyGCN, and EnGN, respectively. PEDAL is conservative on adding an excessive amount of PEs because a) too many APEs to access the feature buffer will cause serialization issue, b) too many APEs will cause cache thrashing to the capacity-limited feature buffer, c) an appropriate ratio of APEs and MPEs is important to load balance between aggregation and combination. By employing a lower number of PEs, PEDAL achieves lower power consumption while keeping a comparable performance, thus having better power efficiency.

5.3.4 Power and Area Breakdown

Module	Components	Power	Area
APE	Accumulator	0.2%	0.07%
	Index Matcher	7.7%	1.80%
	Controller	0.7%	0.20%
	TaskQueue	1.04%	2.98%
MPE	Adder Tree	2.4%	6.42%
	MAC	42.8%	11.41%
	Controller	0.45%	0.31%
	Weight Buffer	0.9%	12.91%
	Task Queue	0.53%	1.48%
Feature Buffer	Buffer	40%	49.93%
Scheduler	Partial Results	0.07%	0.40%
	Edge Buffer	2.9%	11.83%
	Controller	0.06%	0.07%
	ReLU	0.15%	0.00%

Table 5.5: Power and Area breakdown

PEDAL has an average power consumption of 2.04W, which is 69.6% and 47.3% lower than HyGCN and EnGN, respectively (Table 5.4). Compared to HyGCN with general-purpose SIMD units, PEDAL customizes processing elements and requires less computing power. Besides, the limited feature buffer size of EnGN increases the miss rate drastically for large datasets, such as Reddit. This results in higher eDRAM power consumption compared to PEDAL that uses 8MB of feature buffer. The total area of PEDAL is 4.05 mm^2, which is 48.1% smaller than HyGCN and 14.4% higher than EnGN, respectively. Table 5.5 lists each component's power and area breakdown.

5.3.5 Discussion

PEDAL supports multiple dataflows and phase orderings. Figure 5.2 shows that operation distributions vary in different dataflows, making either APEs or MPEs the bottleneck; solely adding more resources to the architecture can only help certain dataflow but not all. Besides, PEDAL also needs to support Row-Wise mode, where the feature buffer is used as a unified cache. Adding too many APEs requires increasing the size of the feature buffer to ensure access latency. Either of the solutions is too expensive for the potential performance gain of this design.

5.4 Conclusion

In this work, I present PEDAL, a power-efficient accelerator for GCN inference supporting multiple dataflows. To accommodate different input graph sizes and densities, as well as GCN variants with different aggregation functions, PEDAL features multiple dataflows, namely **IP-AC**, **RW-AC**, and **RW-CA**, to support performing GCN inference in both phase orderings efficiently. I evaluate the performance of PEDAL using a cycle-accurate simulator and do RTL synthesis to get power and area. PEDAL achieves 144.5×, 9.36×, and 2.55× speedup compared to CPU, GPU, and HyGCN respectively, and 8856×, 1606×, 8.4× and 1.78× better power efficiency compared to CPU, GPU, HyGCN and EnGN respectively.

CHAPTER 6

Conclusion And Future Work

Emerging applications are increasingly important parts of our life, and it is crucial to improve their execution efficiency and scalability. This thesis focuses on video transcoding and graph algorithms, performs hardware characterization to find the bottleneck, and uses software-hardware co-design to improve their performance.

The graph is a data structure that can effectively model the complicated relationship between entities. Graphs are widely used in everyday life and scientific research; for example, social networks and road networks are represented using graphs, and molecules in chemistry and biology, and particles in physics are also described using graphs. The inherent irregular memory access pattern and the growing size of real-world graphs make it challenging to run graph-based algorithms in today's general-purpose hardware like CPUs and GPUs. It is crucial to speed up the execution of graph algorithms through both software optimization and hardware design.

This dissertation presented CPU characterization on video transcoding, revealing how the bottlenecks change with respect to software parameters. Then, it presented a software solution and a hardware design to speed up graph algorithms. On the software side, I used graph sparsification to substitute the full graph with a much smaller sparsified graph to achieve speedup. I comprehensively studied how different graph sparsification algorithms perform in preserving graph properties. On the hardware side, I designed an accelerator for GCNs, which supports multiple dataflows, achieving both flexibility and efficiency when executing different GCNs and input graphs.

More specifically, first, I performed CPU characterization on video transcoding to understand the hardware bottlenecks and how they change with different software parameters. The characterization helps future hardware optimization for specific applications. Guided by the characterization results, the work used Graphite, AutoFDO, and hardware-aware scheduler and achieved an average speedup of 4.42%, 4.66%, and 3.72%, respectively.

Second, graph sparsification is used to substitute the full graph with a sparsified graph. The sparsified graph has much fewer edges and, thus, is much smaller in size. A sparsified graph is considered a good delegate of the full graph if the results of the downstream tasks are close to that of the full graph. The lack of understanding of how different graph sparsification algorithms affect different graph properties makes it hard to make an informed choice of the appropriate sparsification algorithm. I conducted a comprehensive benchmark on 12 graph sparsification algorithms, explored their performance in preserving 16 essential graph properties on 14 real-world graphs, and gave insights into how to choose the best sparsification algorithm for different downstream tasks.

Last, I presented PEDAL, a power-efficient GCN accelerator. This work observed prior accelerators only support one dataflow, which does not execute all GCNs at the best efficiency. PEDAL proposed an accelerator that supports three dataflows, considering both efficiency and flexibility. PEDAL also used a decision tree to automatically choose the best dataflow for a given GCN model and input graph. PEDAL achieved an average speedup of 144.5×, 9.36× and 2.55× compared to CPU, GPU and HyGCN, respectively, and achieved an average power efficiency by 8856×, 1606×, 8.4×, and 1.78× compared to CPU, GPU, HyGCN, and EnGN, respectively.

While the software and hardware solutions presented in this thesis significantly reduced the amount of work and improved execution efficiency, there are further research directions can be explored to speed up the graph algorithms even more.

Understanding the execution cost and bottleneck of graph sparsification. Chapter 4 revealed the performance of 12 sparsification algorithms on preserving different graph properties. However, when it comes to the cost of performing these sparsification algorithms, it only briefly compared the cost using sparsification time. More benchmarking can be performed to understand the execution time and memory footprint for each sparsification algorithm and how they are related to the prune rates and input graph characteristics. It is also important to know the sparsification overhead compared to the execution time of the downstream tasks for the end-to-end speedup. Finally, there are opportunities to develop dedicated hardware for some of the time-consuming but well-performed sparsification algorithms and integrate it as part of the graph algorithm accelerator.

Support for more operations to extend the applicability to broader GNNs. Chapter 5 presented an accelerator that supports multiple dataflows to achieve both execution efficiency and flexibility. To achieve optimal power efficiency, the accelerator only included the most popular operations. This limits the accelerator to certain types of Graph Convolutional Networks, which is a subset of the Graph Neural Network family. More operations can be added to the accelerator, extending it to more GNN models. The design can also be modularized to quickly design an accelerator for a specific GNN model with minimal changes.

APPENDIX A

Pseudo Code for Sparsification Algorithms

This appendix provides pseudo-code to some of the sparsification algorithms as a supplement to Chapter 4.

A.1 *Rank Degree* Sparsifier

Algorithm 1 *Rank Degree* sparsifier

```
1: procedure RankDegreeSparsifier($\boldsymbol{G}$)
2:     Input: $\boldsymbol{G}$: Graph to sparsify
3:     Input: $\rho$: $0 < \rho \leq 1$ selects top $\rho * \#$neighbors for each vertex
4:     Output: $\boldsymbol{H}$: Sparsified graph
5:
6:    seeds = [$u_1$, $u_2$, ..., $u_s$]        ▷ selects s vertices uniformly at random
7:    $\mathcal{V}_H = \varnothing$, $\mathcal{E}_H = \varnothing$        ▷ initialize vertex and edge sets in $\boldsymbol{H}$
8:    while —$\mathcal{V}_H$— ¡ —$\mathcal{V}_G$— do
9:        new_seeds=$\varnothing$
10:        for all $u \in$ seeds do
11:            neighs=getNeighborsOf($u$)
12:            ranks = {}        ▷ dictionary, key is vertex, value is degree
13:            for all $v \in$ neighs do
14:                ranks[$v$] = getDegreeOf($v$)
15:            sort ranks by value
16:            select top k=$\rho \times$len(neighbors), $v_1, ..., v_k$
17:            new_seeds = new_seeds $\bigcup$ [$v_1, ..., v_k$]
18:            $\mathcal{E}_H = \mathcal{E}_H \bigcup [(u, v_1), ..., (u, v_k)]$
19:        seeds=new_seeds
20:     $\boldsymbol{H} = \{Vertex(\mathcal{V}_{\boldsymbol{G}}), Edge(\mathcal{E}_H)\}$
21:     return $\boldsymbol{H}$
```

A.2 *Local Degree* Sparsifier

Algorithm 2 Local degree score

```
1: procedure getEdgeScore($G$)
2:     Input: $G$: Graph to calculate edge scores
3:     Output: Scores: An array of edge scores for each edge
4:
5:     scores = [0, ..., 0]                                   ▷ scores is an array of length #edges
6:     for all $v_i \in \mathcal{V}$ do                              ▷ iterate all vertex v in $\mathcal{V}$
7:         $d_i$ = degree($v_i$)
8:         neighbor_degree = {}                    ▷ dictionary, key is edge id, value is degree
9:         for all $v_j \in$ Neighbor($v_i$) do              ▷ find degrees of all neighboring vertex
10:            $d_j$ = degree($v_j$)
11:            eid = $e_{ij}$.edgeID
12:            neighbor_degree[eid] = $d_j$
13:
14:        sort(neighbor_degree)                                                ▷ sort by degree
15:
16:        last_rank, last_degree, num_same = 0
17:        neighbor_rank = {}                          ▷ dictionary, key is edge id, value is rank
18:        for all item $\in$ neighbor_degree do       ▷ compute the rank of neighbors by degree
19:            eid = item.key
20:            $d_j$ = item.value
21:            if $d_j$ == last_degree then                          ▷ same rank for same degree
22:                num_same++
23:            else
24:                last_rank += num_same
25:                num_same = 1
26:                last_degree = $d_j$
27:            neighbor_rank[eid] = lask_rank
28:
29:        for all item $\in$ neighbor_degree do
30:            eid = item.key
31:            rank = item.value
32:            s = 1.0 - log(rank)/log($d_i$)
33:            // score for an edge can be updated multiple times, take the max score
34:            scores[eid] = max(scores[eid], s)
35:    return scores
```

A.3 *t-Spanner*

Algorithm 3 Greedy algorithm for t-spanner construction

```
1: procedure CONSTRUCT T-SPANNER($G, t$)
2:     Input: $G$: Original graph
3:     Input: $t$: stretch factor, must be an odd number ¿ 1
4:     Output: $H$: t-spanner graph
5:
6:     $H \leftarrow (V, \varnothing)$                ▷ Init $H$ to have the same set of vertices, and no edges
7:     for all $e_{uv} \in E$ in non-decreasing order do
8:         if $d_H(u, v) > tw(u, v)$ then
9:             add $e_{uv}$ to $H$
10:    return $H$
```

A.4 *Forest Fire*

The *Forest Fire* model can be described more formally as follows (modified from [90]):

1. A new vertex $\boldsymbol{u}$ chooses an existing vertex $\boldsymbol{v}$ uniformly at random, and forms an edge to it.

2. Two random numbers x and y are generated geometrically distributed with means $p/(1-p)$ and $rp/(1-rp)$, where p is the forward burning probability, and r is the backward burning ratio.

3. Vertex $\boldsymbol{v}$ selects x outgoing edges and y incoming edges that are not visited yet, if there are not enough unvisited edges, select all. For undirected edges, every edge can be both an outgoing and incoming edge. Let $\boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_{x+y}$ denote the other end of the selected edges.

4. Vertex $\boldsymbol{u}$ forms edges to the $\boldsymbol{w}_1, \boldsymbol{w}_1, ..., \boldsymbol{w}_{x+y}$.

5. Repeat (3) and (4) recursively to each of the $\boldsymbol{w}_1, \boldsymbol{w}_2, ..., \boldsymbol{w}_{x+y}$, until no edge can be added.

Algorithm 4 Forest Fire Score

```
procedure getEdgeScore($G$)
    Input: $G$: Graph to calculate edge scores
    Input: bp: The probability a neighbor vertex is burnt, from 0.0 to 1.0
    Input: targetBurnRatio: In total targetBurnRatio * m edges will be burnt
    Output: Scores: An array of edge scores for each edge

    burnt_count = 0                         ▷ keep track of total number of burnt edges
    scores = [0, ..., 0]                    ▷ scores is an array of length #edges
    burnt = [0, ..., 0]                     ▷ burnt is an array of length #edges

    while burnt_count ¡ targetBurnRatio * numberOfEdges($G$) do
        visited = [false, ..., false]       ▷ visited is an array of length #vertices
        vertexQ = []                        ▷ a queue for vertex to be visited
        vertexQ.add(randomVertex($G$))      ▷ pick a random starting vertex

        while vertexQ is not empty do
            $u$ = vertexQ.pop()
            visted[$u$.id] = True
            neighs = getAllUnvisitedVertices()
            while neighs is not empty do
                r = randNum()               ▷ r is a random float from 0.0 to 1.0
                if r $\leq$ bp then
                    break                   ▷ decides to burn the vertex, not propagate further
                $v$ = pickRandom(neighs)    ▷ pick a random vertex to propagate fire
                remove(neighs, $v$)         ▷ pick a random vertex to propagate fire
                vertexQ.add($v$)
                eid = getEdgeID(u, v)
                burnt_count++
    max_burnt = max(burnt)
    scores = burnt/max_burnt                ▷ normalize burnt_count to be the scores
    return scores
```

A.5 Similarity-based Sparsifiers

Algorithm 5 G-Spar

```
1:  procedure G-Spar($G$)
2:      Input: $G$: Graph to calculate edge scores
3:      Output: $H$: G-Spar sparsified graph
4:
5:      scores = {}        ▷ scores is a dictionary, key is edge id, value is edge score
6:      for all e $\in E$ do
7:          eid = e.id
8:          score = JaccardScore($e$)
9:          scores[eid] = score
10:     sort scores by value
11:     pick top s% edges to form $H$
12:     return $H$
```

Algorithm 6 L-Spar

```
1:  procedure L-Spar($G$)
2:      Input: $G$: Graph to calculate edge scores
3:      Input: $c$: Exponent parameter
4:      Output: $H$: L-Spar sparsified graph
5:
6:      for all $v \in \mathcal{V}$ do
7:          $d$ = degreeOf($v$)
8:          $\mathcal{E}'$=getEdgesOf($v$)
9:          scores = {}        ▷ scores is a dictionary, key is edge id, value is edge score
10:         for all e $\in \mathcal{E}'$ do
11:             eid = e.id
12:             score = JaccardScore($e$)
13:             scores[eid] = score
14:         sort scores by value
15:         add top $d^c$ edges to $H$
16:     return $H$
```

Algorithm 7 Edge Triangle Count

```
1: procedure EdgeTriangleCount($\boldsymbol{G}$)
2:     Input: $\boldsymbol{G}$: Graph to calculate triangle edge scores
3:     Output: triangle_count: Triangle count for each edge
4:
5:     triangle_count = [0, ..., 0]                                    ▷ array of length #edges
6:     incident_triangle_count = [None, ..., None]                     ▷ array of length #vertices
7:
8:     for all $u \in \boldsymbol{V}$ do                                          ▷ first vertex in triangle
9:         for all $v \in$ getNeighborsOf($u$) do
10:            incident_triangle_count[$v$] = 0              ▷ mark all neighboring vertices not None
11:        for all $v \in$ getNeighborsOf($u$) do                         ▷ second vertex in triangle
12:            for all $w \in$ getNeighborsOf($w$) do                      ▷ third vertex in triangle
13:                if incident_triangle_count[$w$] is not None then              ▷ triangle found
14:                    // count triangles to the vertices first, each triangle is counted 3 times
15:                    if $u \geq v$ then
16:                        incident_triangle_count[$v$]++
17:                    if $u \geq w$ then
18:                        incident_triangle_count[$w$]++
19:
20:        // add local triangle count to global, reset local triangle count
21:        for all $v \in$ getNeighborsOf($u$) do
22:            eid = getEdgeId(u, v)
23:            if incident_triangle_count[$v$] > 0 then
24:                triangle_count[eid]+=incident_triangle_count[$v$]
25:            incident_triangle_count[$v$] = None
26:    return triangle_count
```

The Edge Triangle Count is not a standalone sparsification algorithm. It is listed here because it is used to calculate the local similarity score and the SCAN structural similarity score.

Algorithm 8 Local similarity score

```
1: procedure LocalSimilarityScore($G$)
2:     Input: $G$: Graph to calculate triangle edge scores
3:     Output: scores: Local similarity scores for each edge
4:
5:     scores = [0, ..., 0]                                  ▷ array of length #edges
6:
7:     triangle_count = EdgeTriangleCount($G$)
8:     for all $u \in V$ do
9:         neighbors_sims = {}                               ▷ dictionary, key is edge id, value is similarity
10:        $d_u$ = getDegreeOf($u$)
11:        for all $v \in$ getNeighborsOf($u$) do
12:            $d_v$ = getDegreeOf($v$)
13:            eid = getEdgeId($u$, $v$)
14:            sim = triangle_count[eid]/($d_u$+$d_v$-triangle_count[eid])
15:            scores[eid] = max(scores[eid], sim)
16:    return scores
```

Algorithm 9 SCAN Structural Similarity Score

```
1: procedure SCANStructuralSimilarityScore($G$)
2:     Input: $G$: Graph to calculate triangle edge scores
3:     Output: scores: SCAN structural similarity scores for each edge
4:
5:     scores = [0, ..., 0]                                  ▷ array of length #edges
6:
7:     triangle_count = EdgeTriangleCount($G$)
8:     for all $u \in V$ do
9:         neighbors_sims = {}                               ▷ dictionary, key is edge id, value is similarity
10:        $d_u$ = getDegreeOf($u$)
11:        for all $v \in$ getNeighborsOf($u$) do
12:            $d_v$ = getDegreeOf($v$)
13:            eid = getEdgeId($u$, $v$)
14:            sim = (triangle_count[eid]+1)/$\sqrt{(d_u+1)*(d_v+1)}$
15:            scores[eid] = sim
16:    return scores
```

A.6 Effective Resistance ($\boldsymbol{ER}$) Sparsifier

I briefly summarize the derivation of the effective resistance. Interested readers should refer to [132] for more details.

I first define the following notations:

$\mathbb{R}$: real number.

G: Input Graph, in this write-up, G must be symmetrical (undirected).

$|\mathcal{V}|$: Number of Vertices in G.

$|\mathcal{E}|$: Number of Edges in G.

$\boldsymbol{A}$: $\in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, Adjacency Matrix of G.

$\boldsymbol{D}$: $\in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, Degree Matrix of G, where i^{th} diagonal entry is the degree of i^{th} vertex, if the graph is weighted, then it's the sum of all edge weights related to vertex i.

$\boldsymbol{L}$: $\in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{V}|}$, Laplacian Matrix of G, $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$.

$\boldsymbol{B}$: Incidence Matrix, $\in \mathbb{R}^{|\mathcal{E}|\times|\mathcal{V}|}$. Each row in $\boldsymbol{B}$ represents an edge, where the head vertex is -1, the tail vertex is 1, and all others are 0s. The head and tail of an undirected edge are randomly assigned.

$\boldsymbol{W}$: Weight Matrix, $\in \mathbb{R}^{|\mathcal{E}|\times|\mathcal{E}|}$, is a diagonal matrix, and each diagonal entry represents an edge weight. If the graph is unweighted, then W becomes an Identity Matrix $\boldsymbol{I}$.

χ_u: A unit vector of length $|\mathcal{V}|$, where only the u^{th} element is 1, others are 0s.

R_{uv}: The effective resistance of edge uv.

The derivation of the effective resistance is as follows:

$$\boldsymbol{L} = \boldsymbol{B}^T\boldsymbol{W}\boldsymbol{B} \qquad \text{(proof omitted)} \tag{A.1a}$$

According to Kirchhoff's law, the current flow in is always the same as the current flow out of the vertex,

$$\boldsymbol{B}^T\boldsymbol{i} = \boldsymbol{c_{ext}} \tag{A.1b}$$

According to Ohm's law,

$$\boldsymbol{i} = \boldsymbol{W}\boldsymbol{B}\nu \tag{A.1c}$$

Combing eq. (A.1a), (A.1b), and (A.1c),

$$\boldsymbol{B}^T\boldsymbol{W}\boldsymbol{B}\nu = \boldsymbol{L}\nu = \boldsymbol{c_{ext}} \tag{A.1d}$$

Let $\boldsymbol{L}^+$ be the pseudo-inverse of $\boldsymbol{L}$, because Laplacian matrix is positive semi-definite, and doesn't have an inverse

$$\nu = \boldsymbol{L}^+\boldsymbol{c_{ext}} \tag{A.1e}$$

Now set $\boldsymbol{c_{ext}} = \boldsymbol{\chi_u} - \boldsymbol{\chi_v}$, then eq. (A.1e) can be written as

$$\nu = \boldsymbol{L}^+(\boldsymbol{\chi_u} - \boldsymbol{\chi_v}) \tag{A.1f}$$

Multiply both sides by $(\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T$,

$$(\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T\nu = (\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T\boldsymbol{L}^+(\boldsymbol{\chi_u} - \boldsymbol{\chi_v}) \tag{A.1g}$$

Notice that $(\boldsymbol{\chi_u} - \boldsymbol{\chi_v})$ is equivalent to the transpose of i^{th} row in $\boldsymbol{B}$, denoted by $\boldsymbol{B}[i]$, thus,

$$\boldsymbol{B}[i]^T\nu = \boldsymbol{B}[i]\boldsymbol{L}^+\boldsymbol{B}[i]^T \tag{A.1h}$$

Eq. (A.1h) applies to every i, thus can generalized to

$$\boldsymbol{B}^T\nu = \boldsymbol{B}\boldsymbol{L}^{+}\boldsymbol{B}^T \tag{A.1i}$$

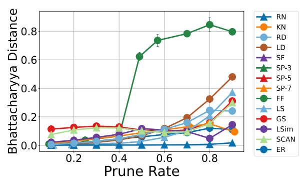
The l.h.s. of eq. (A.1g) is the voltage difference between u and v, which can be used to represent the effective resistance of the edge connecting u and v. Thus, the effective resistance is defined as

$$\begin{aligned}
R_{uv} &= (\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T\boldsymbol{L}^{+}(\boldsymbol{\chi_u} - \boldsymbol{\chi_v}) \\
&= (\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T\boldsymbol{L}^{+}\boldsymbol{L}\boldsymbol{L}^{+}(\boldsymbol{\chi_u} - \boldsymbol{\chi_v}) \\
&= (\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T\boldsymbol{L}^{+}\boldsymbol{B}^T\boldsymbol{W}\boldsymbol{B}\boldsymbol{L}^{+}(\boldsymbol{\chi_u} - \boldsymbol{\chi_v}) \\
&= ((\boldsymbol{\chi_u} - \boldsymbol{\chi_v})^T\boldsymbol{L}^{+}\boldsymbol{B}^T\boldsymbol{W}^{1/2})(\boldsymbol{W}^{1/2}\boldsymbol{B}\boldsymbol{L}^{+}(\boldsymbol{\chi_u} - \boldsymbol{\chi_v})) \\
&= ||\boldsymbol{W}^{1/2}\boldsymbol{B}\boldsymbol{L}^{+}(\boldsymbol{\chi_u} - \boldsymbol{\chi_v})||_2^2
\end{aligned} \tag{A.1j}$$

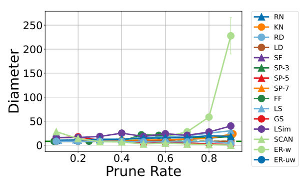
APPENDIX B

Full Results for Sparsification Benchmark

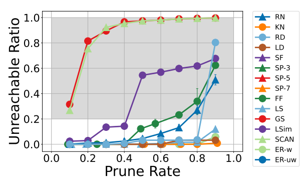
I only showed a subset of the results in Chapter 4. This appendix presents the full results generated in the sparsifiers benchmark. Each page shows one dataset, and the sub-captions note which metric each subgraph measures. Some figures are missing, and there are three possible reasons: 1) Some metrics are supported for directed graphs, they are Clustering F1 Similarity, Number of Communities, and Modularity; 2) Some experiments couldn't finish within 24 hours, especially on time-consuming metrics like Eigenvector Centrality and large graphs; 3) Some experiments run out of memory and triggered OOM kill by the OS.



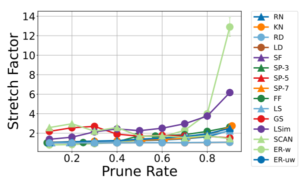
(a) Degree Distribution



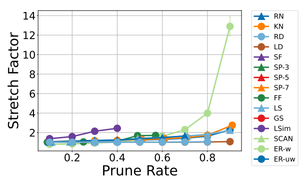
(b) Diameter



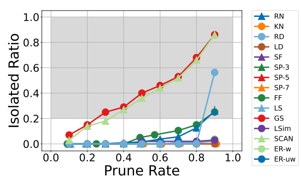
(c) Average SPSP Unreachable Ratio



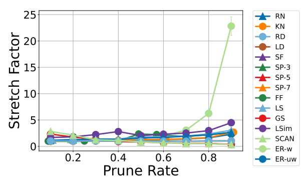
(d) SPSP Stretch Factor



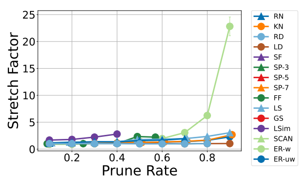
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



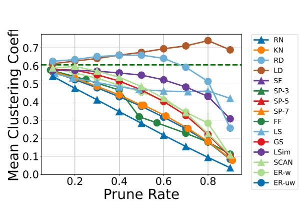
(f) Vertex Isolated Ratio



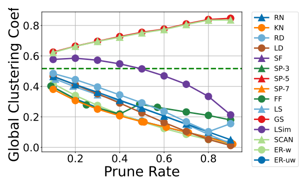
(g) Eccentricity Stretch Factor



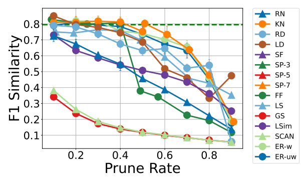
(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



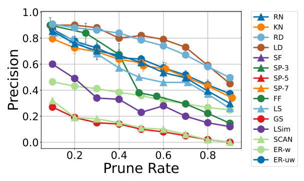
(i) Mean Clustering Coefficient



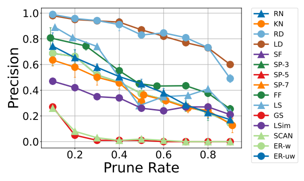
(j) Global Clustering Coefficient



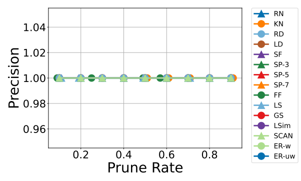
(k) Clustering F1 Similarity



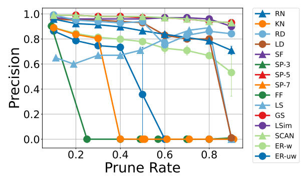
(l) Betweenness Centrality Precision



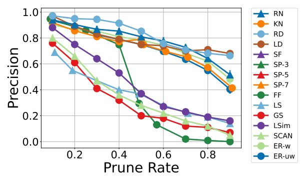
(m) Closeness Centrality Precision



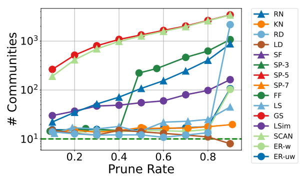
(n) Katz Centrality Precision



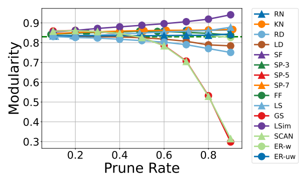
(o) Eigenvector Centrality Precision



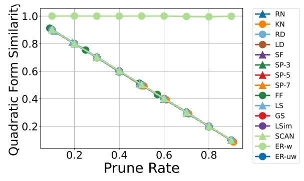
(p) PageRank Precision



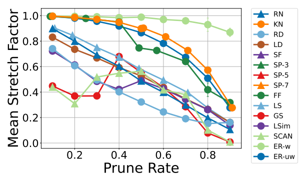
(q) Number of Communities



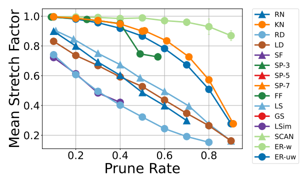
(r) Modularity



(s) Quadratic Form Similarity

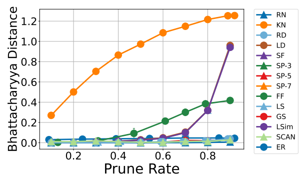


(t) Max Flow Stretch Factor

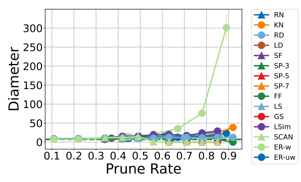


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

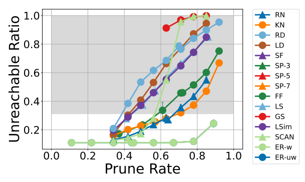
Figure B.1: Metric Evaluation on ego-Facebook



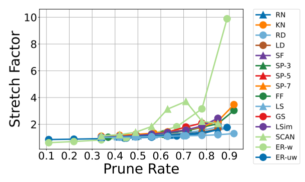
(a) Degree Distribution



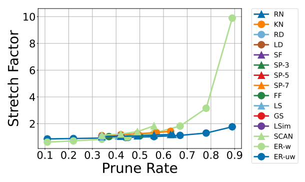
(b) Diameter



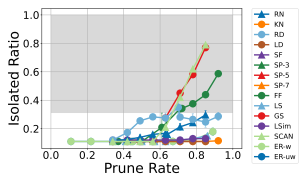
(c) Average SPSP Unreachable Ratio



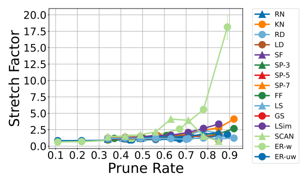
(d) SPSP Stretch Factor



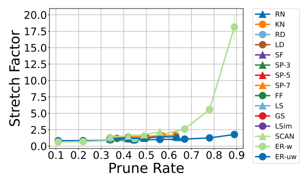
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



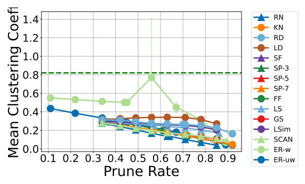
(f) Vertex Isolated Ratio



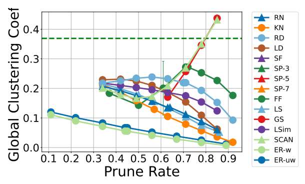
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



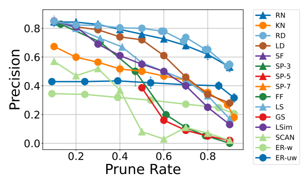
(i) Mean Clustering Coefficient



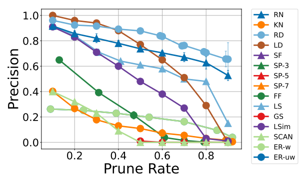
(j) Global Clustering Coefficient

Unsupported for directed graph

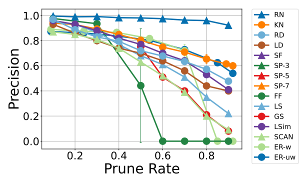
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



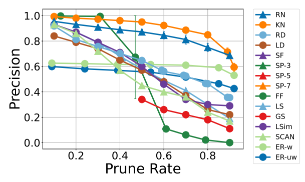
(m) Closeness Centrality Precision



(n) Katz Centrality Precision

Timed out after 24 hours

(o) Eigenvector Centrality Precision



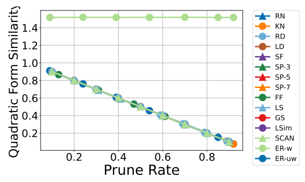
(p) PageRank Precision

Unsupported for directed graph

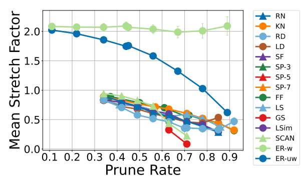
(q) Number of Communities

Unsupported for directed graph

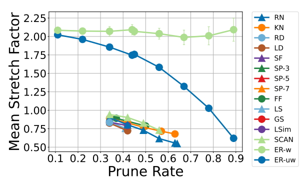
(r) Modularity



(s) Quadratic Form Similarity

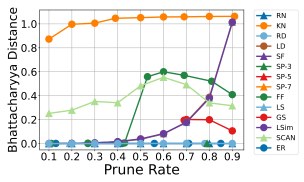


(t) Max Flow Stretch Factor

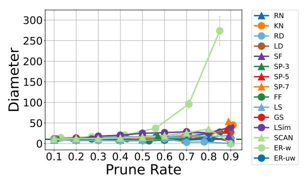


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

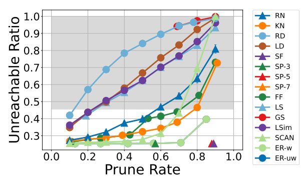
Figure B.2: Metric Evaluation on ego-Twitter



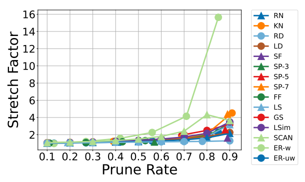
(a) Degree Distribution



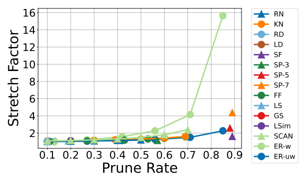
(b) Diameter



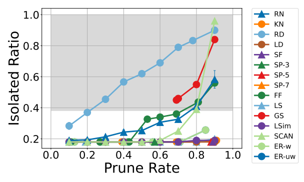
(c) Average SPSP Unreachable Ratio



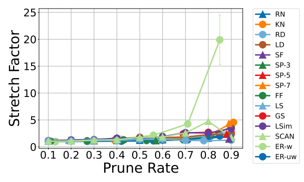
(d) SPSP Stretch Factor



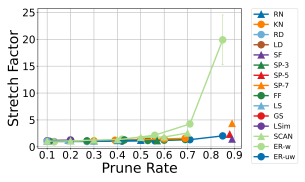
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



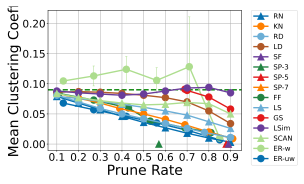
(f) Vertex Isolated Ratio



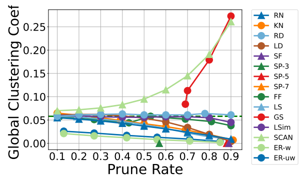
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



(i) Mean Clustering Coefficient



(j) Global Clustering Coefficient

Unsupported for directed graph

(k) Clustering F1 Similarity

Timed out after 24 hours

(l) Betweenness Centrality Precision

Timed out after 24 hours

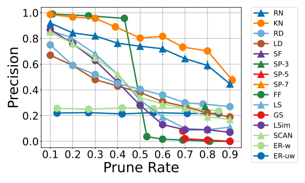
(m) Closeness Centrality Precision

Timed out after 24 hours

(n) Katz Centrality Precision

Timed out after 24 hours

(o) Eigenvector Centrality Precision



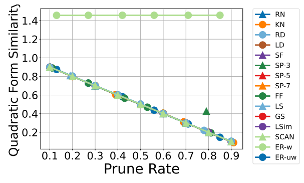
(p) PageRank Precision

Unsupported for directed graph

(q) Number of Communities

Unsupported for directed graph

(r) Modularity



(s) Quadratic Form Similarity

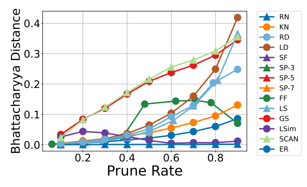
Timed out after 24 hours

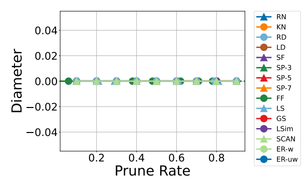
(t) Max Flow Stretch Factor

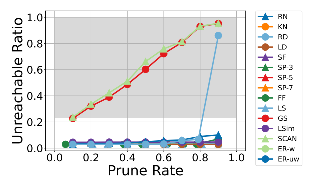
Timed out after 24 hours

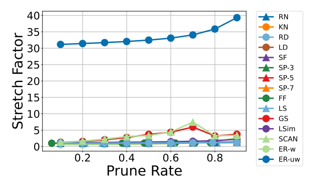
(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

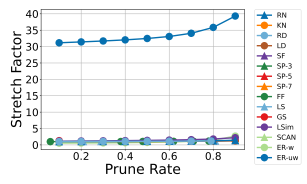
Figure B.3: Metric Evaluation on soc-Pokec

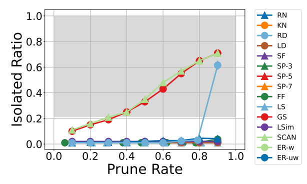

(a) Degree Distribution

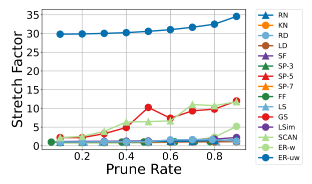

(b) Diameter

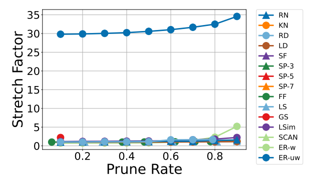

(c) Average SPSP Unreachable Ratio

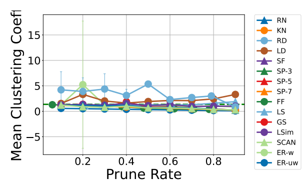

(d) SPSP Stretch Factor

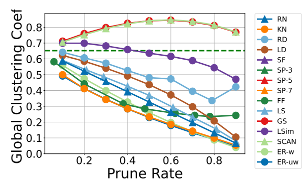

(e) SPSP Stretch Factor with Unreachable Ratio Constraint

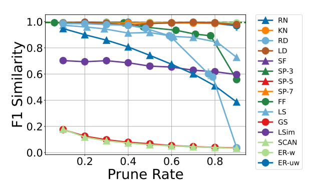

(f) Vertex Isolated Ratio

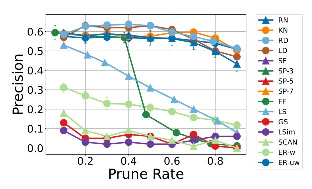

(g) Eccentricity Stretch Factor

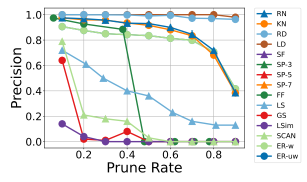

(h) Eccentricity Stretch Factor with Isolated Ratio Constraint

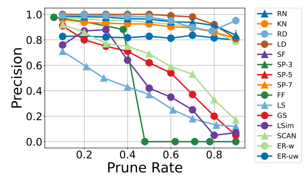

(i) Mean Clustering Coefficient

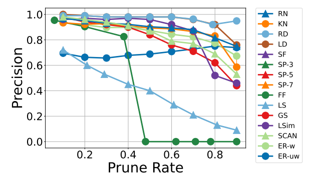

(j) Global Clustering Coefficient

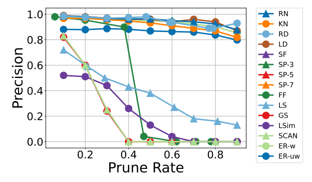

(k) Clustering F1 Similarity

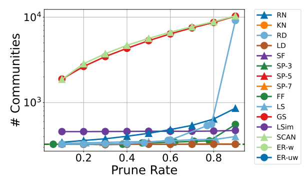

(l) Betweenness Centrality Precision

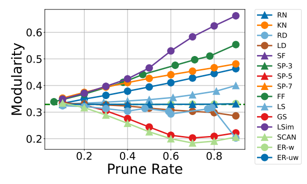

(m) Closeness Centrality Precision

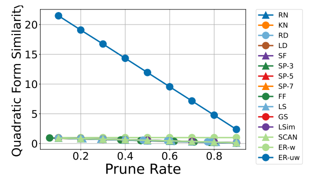

(n) Katz Centrality Precision

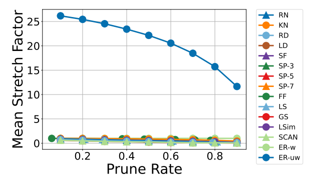

(o) Eigenvector Centrality Precision


(p) PageRank Precision


(q) Number of Communities


(r) Modularity


(s) Quadratic Form Similarity


(t) Max Flow Stretch Factor

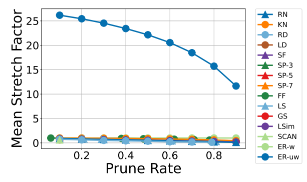
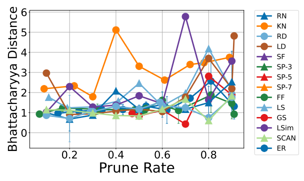
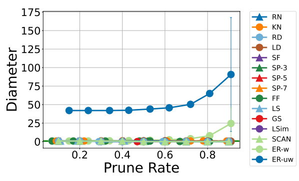

(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

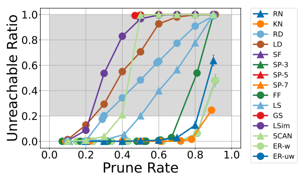
Figure B.4: Metric Evaluation on human_gene2



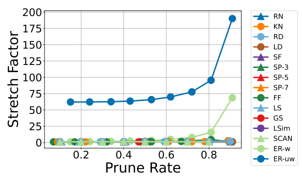
(a) Degree Distribution



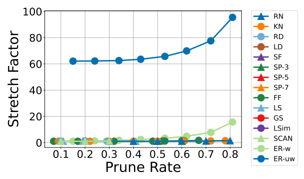
(b) Diameter



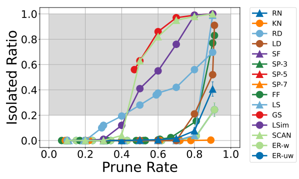
(c) Average SPSP Unreachable Ratio



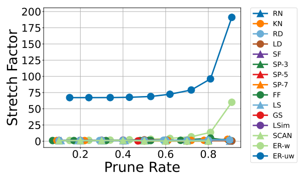
(d) SPSP Stretch Factor



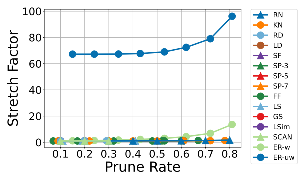
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



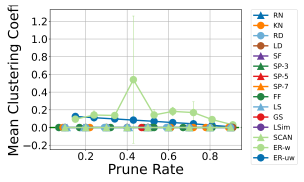
(f) Vertex Isolated Ratio



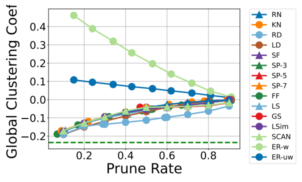
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



(i) Mean Clustering Coefficient



(j) Global Clustering Coefficient

Unsupported for directed graph

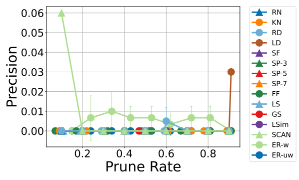
(k) Clustering F1 Similarity

Timed out after 24 hours

(l) Betweenness Centrality Precision

Timed out after 24 hours

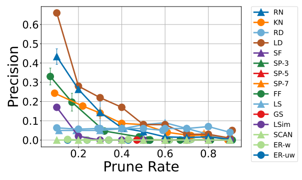
(m) Closeness Centrality Precision



(n) Katz Centrality Precision

Timed out after 24 hours

(o) Eigenvector Centrality Precision



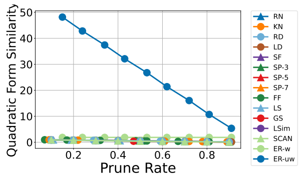
(p) PageRank Precision

Unsupported for directed graph

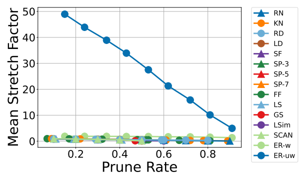
(q) Number of Communities

Unsupported for directed graph

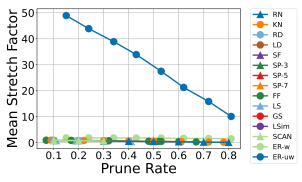
(r) Modularity



(s) Quadratic Form Similarity

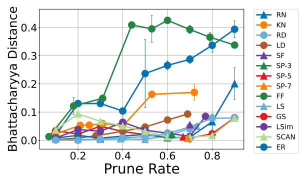


(t) Max Flow Stretch Factor

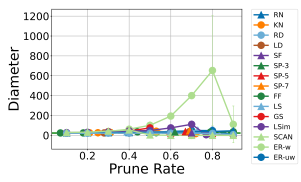


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

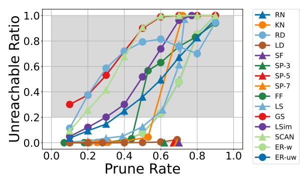
Figure B.5: Metric Evaluation on cage14



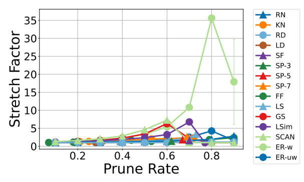
(a) Degree Distribution



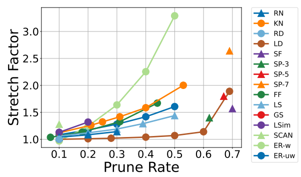
(b) Diameter



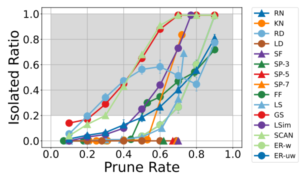
(c) Average SPSP Unreachable Ratio



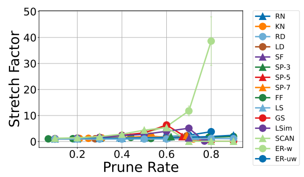
(d) SPSP Stretch Factor



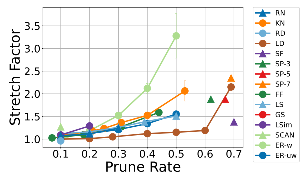
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



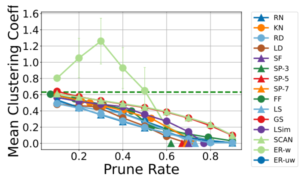
(f) Vertex Isolated Ratio



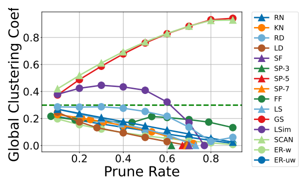
(g) Eccentricity Stretch Factor



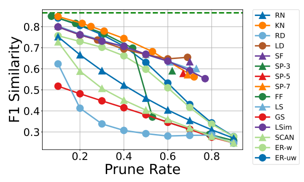
(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



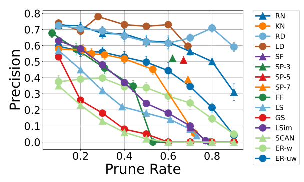
(i) Mean Clustering Coefficient



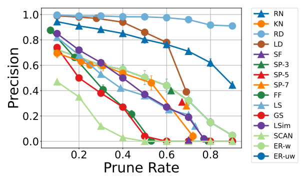
(j) Global Clustering Coefficient



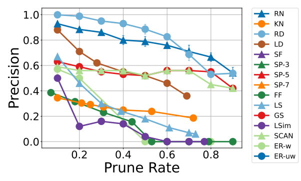
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



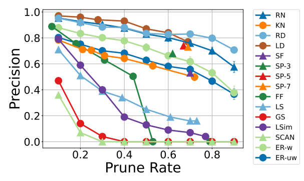
(m) Closeness Centrality Precision



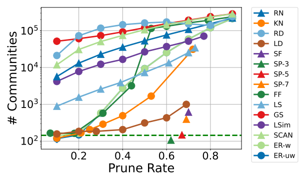
(n) Katz Centrality Precision

Timed out after 24 hours

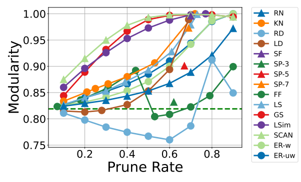
(o) Eigenvector Centrality Precision



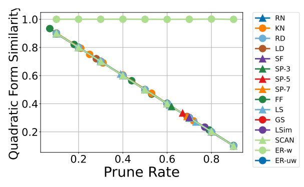
(p) PageRank Precision



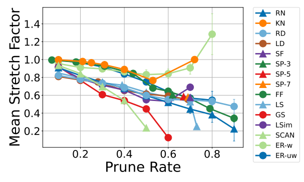
(q) Number of Communities



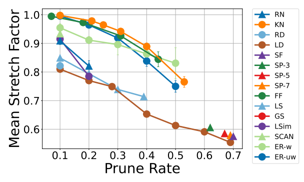
(r) Modularity



(s) Quadratic Form Similarity

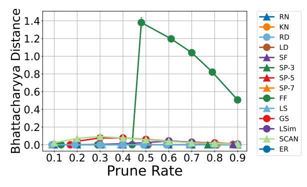


(t) Max Flow Stretch Factor

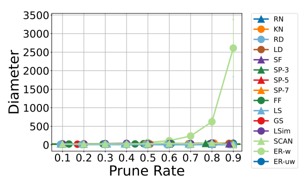


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

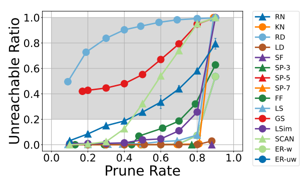
Figure B.6: Metric Evaluation on com-DBLP



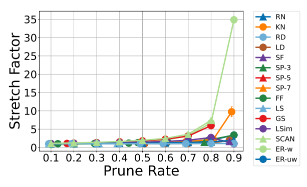
(a) Degree Distribution



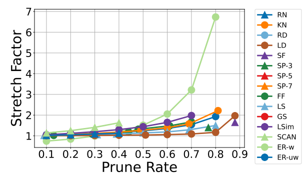
(b) Diameter



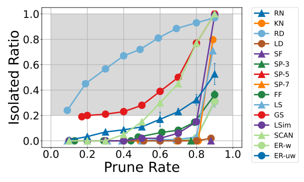
(c) Average SPSP Unreachable Ratio



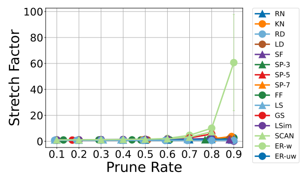
(d) SPSP Stretch Factor



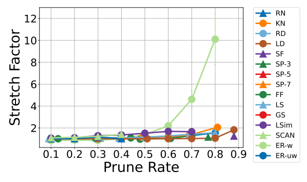
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



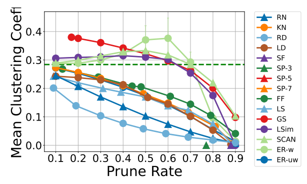
(f) Vertex Isolated Ratio



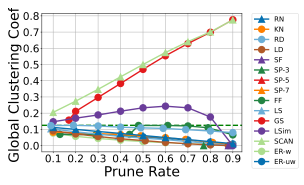
(g) Eccentricity Stretch Factor



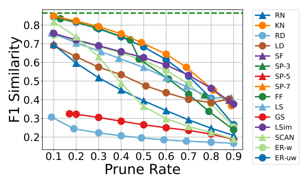
(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



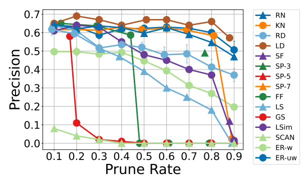
(i) Mean Clustering Coefficient



(j) Global Clustering Coefficient



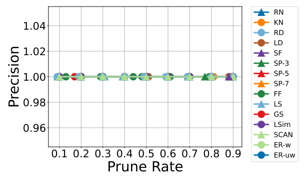
(k) Clustering F1 Similarity



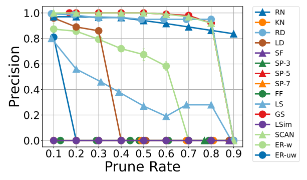
(l) Betweenness Centrality Precision

Timed out after 24 hours

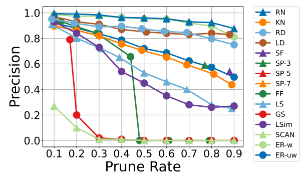
(m) Closeness Centrality Precision



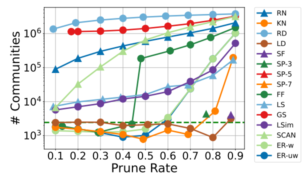
(n) Katz Centrality Precision



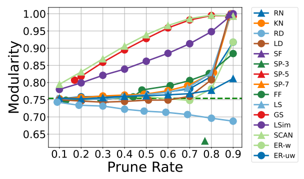
(o) Eigenvector Centrality Precision



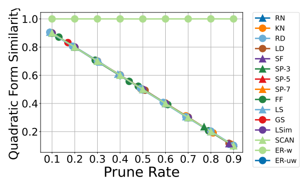
(p) PageRank Precision



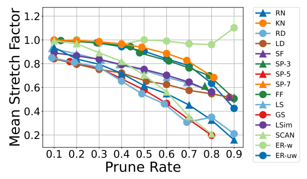
(q) Number of Communities



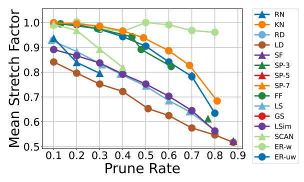
(r) Modularity



(s) Quadratic Form Similarity

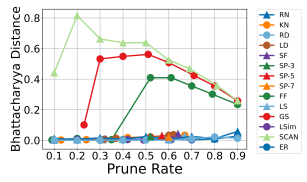


(t) Max Flow Stretch Factor

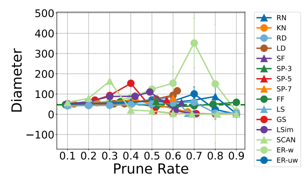


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

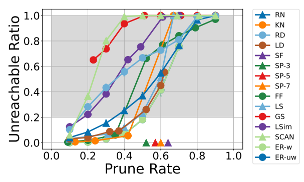
Figure B.7: Metric Evaluation on com-LiveJournal



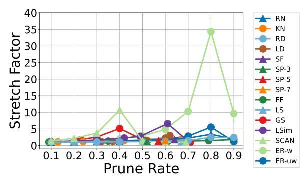
(a) Degree Distribution



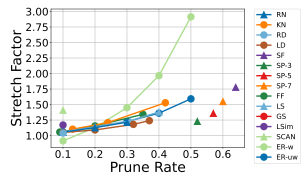
(b) Diameter



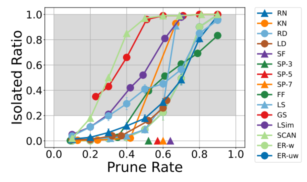
(c) Average SPSP Unreachable Ratio



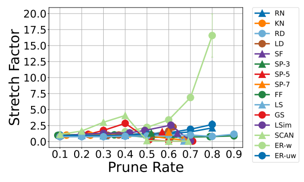
(d) SPSP Stretch Factor



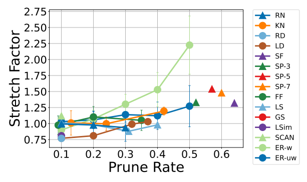
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



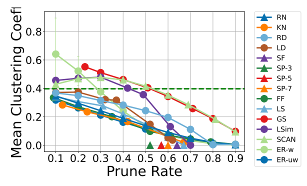
(f) Vertex Isolated Ratio



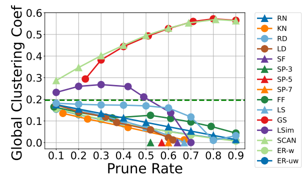
(g) Eccentricity Stretch Factor



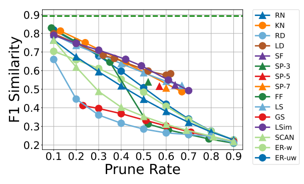
(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



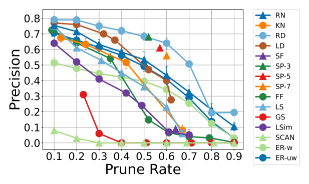
(i) Mean Clustering Coefficient



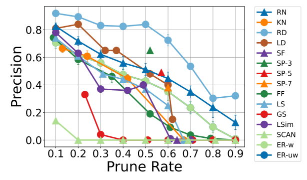
(j) Global Clustering Coefficient



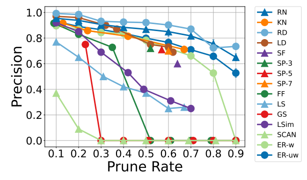
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



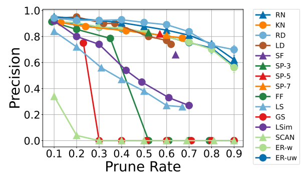
(m) Closeness Centrality Precision



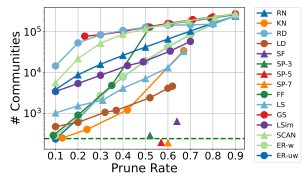
(n) Katz Centrality Precision

Timed out after 24 hours

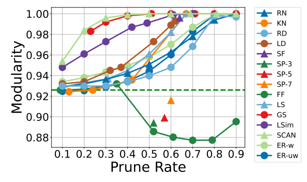
(o) Eigenvector Centrality Precision



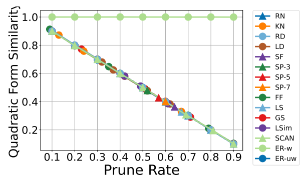
(p) PageRank Precision



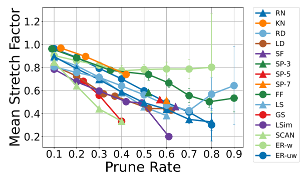
(q) Number of Communities



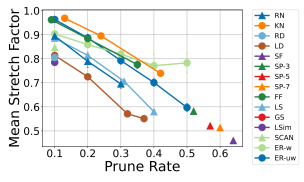
(r) Modularity



(s) Quadratic Form Similarity

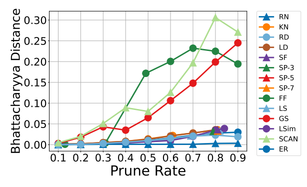


(t) Max Flow Stretch Factor

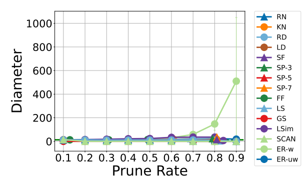


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

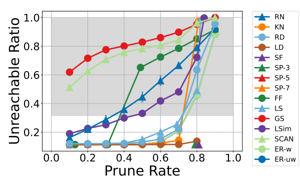
Figure B.8: Metric Evaluation on com-Amazon



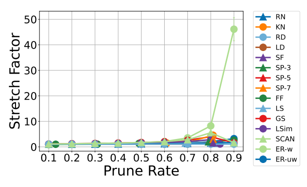
(a) Degree Distribution



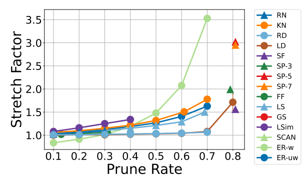
(b) Diameter



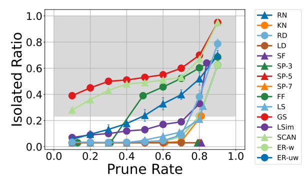
(c) Average SPSP Unreachable Ratio



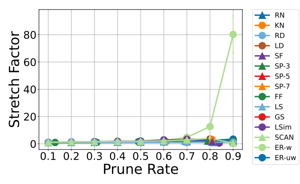
(d) SPSP Stretch Factor



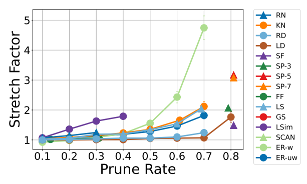
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



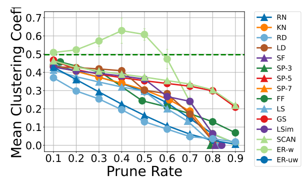
(f) Vertex Isolated Ratio



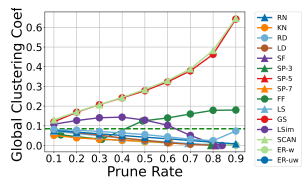
(g) Eccentricity Stretch Factor



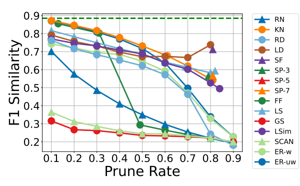
(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



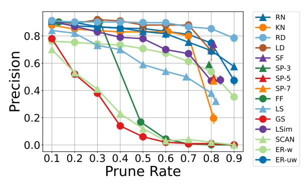
(i) Mean Clustering Coefficient



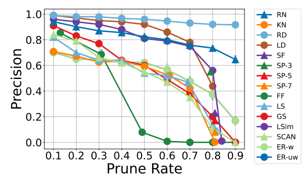
(j) Global Clustering Coefficient



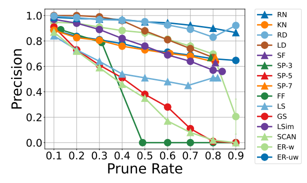
(k) Clustering F1 Similarity



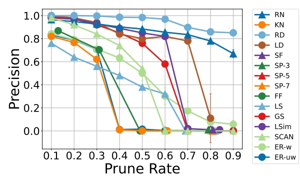
(l) Betweenness Centrality Precision



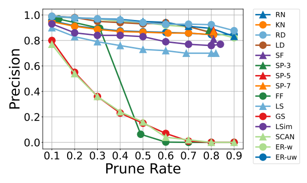
(m) Closeness Centrality Precision



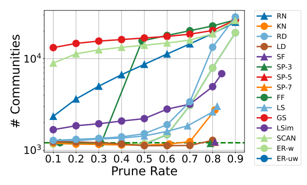
(n) Katz Centrality Precision



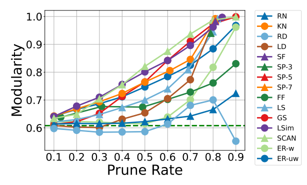
(o) Eigenvector Centrality Precision



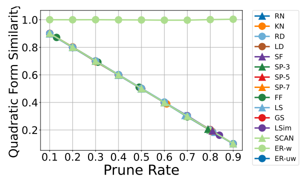
(p) PageRank Precision



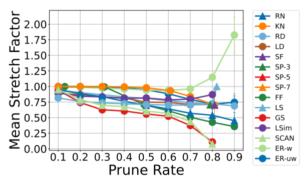
(q) Number of Communities



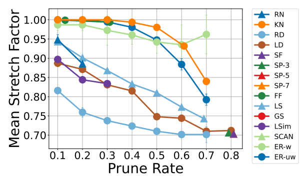
(r) Modularity



(s) Quadratic Form Similarity

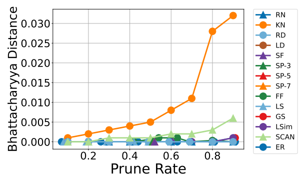


(t) Max Flow Stretch Factor

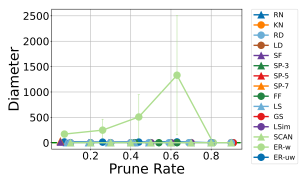


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

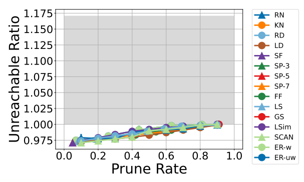
Figure B.9: Metric Evaluation on email-Enron



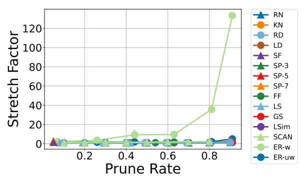
(a) Degree Distribution



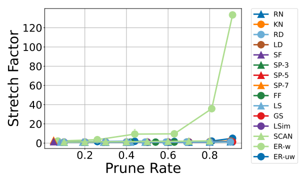
(b) Diameter



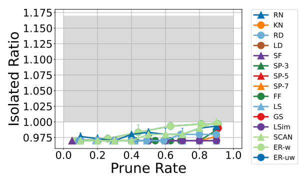
(c) Average SPSP Unreachable Ratio



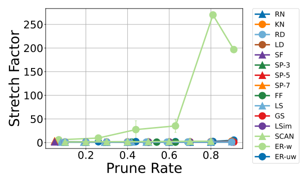
(d) SPSP Stretch Factor



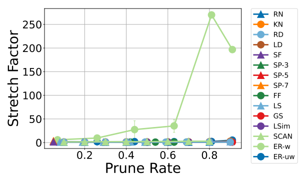
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



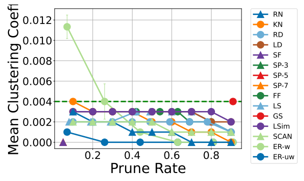
(f) Vertex Isolated Ratio



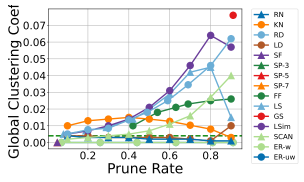
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



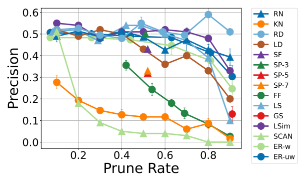
(i) Mean Clustering Coefficient



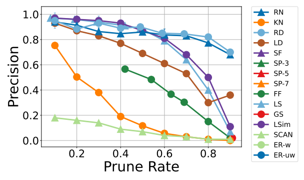
(j) Global Clustering Coefficient

Unsupported for directed graph

(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



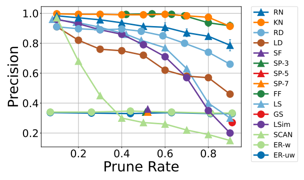
(m) Closeness Centrality Precision

Timed out after 24 hours

(n) Katz Centrality Precision

Timed out after 24 hours

(o) Eigenvector Centrality Precision



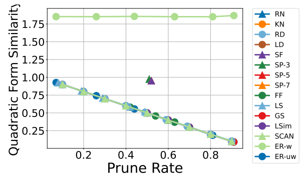
(p) PageRank Precision

Unsupported for directed graph

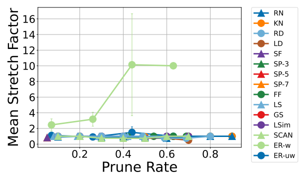
(q) Number of Communities

Unsupported for directed graph

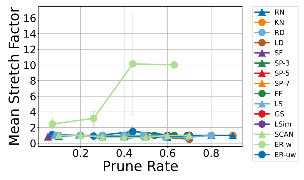
(r) Modularity



(s) Quadratic Form Similarity

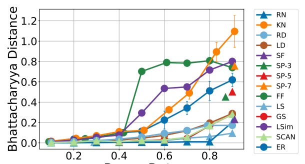


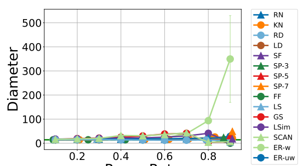
(t) Max Flow Stretch Factor

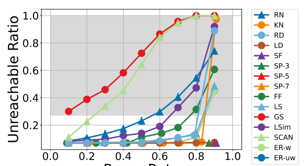


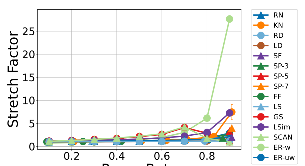
(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

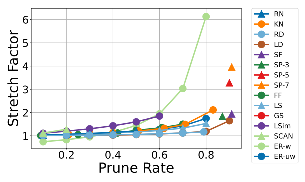
Figure B.10: Metric Evaluation on wiki-Talk

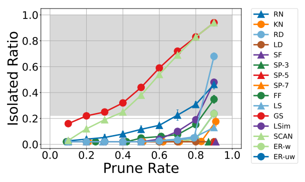

(a) Degree Distribution

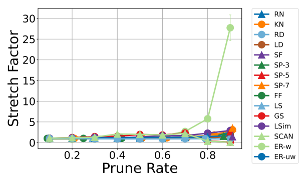

(b) Diameter

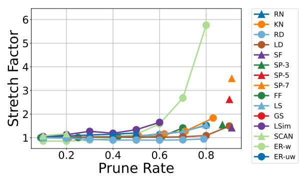

(c) Average SPSP Unreachable Ratio

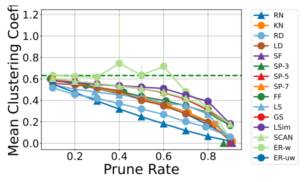

(d) SPSP Stretch Factor

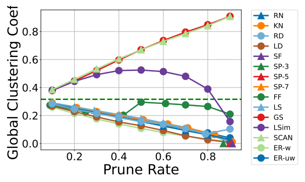

(e) SPSP Stretch Factor with Unreachable Ratio Constraint

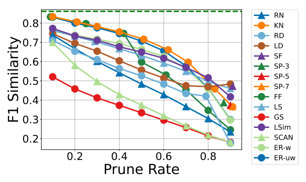

(f) Vertex Isolated Ratio

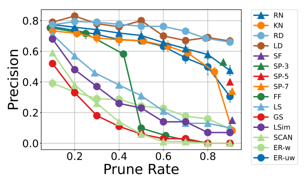

(g) Eccentricity Stretch Factor

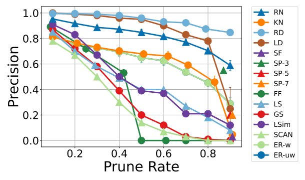

(h) Eccentricity Stretch Factor with Isolated Ratio Constraint

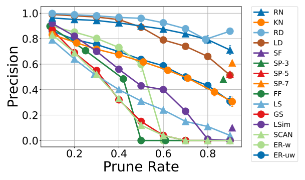

(i) Mean Clustering Coefficient

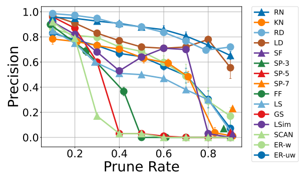

(j) Global Clustering Coefficient

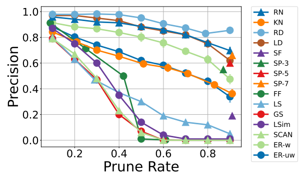

(k) Clustering F1 Similarity

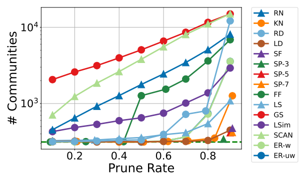

(l) Betweenness Centrality Precision

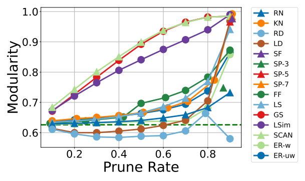

(m) Closeness Centrality Precision

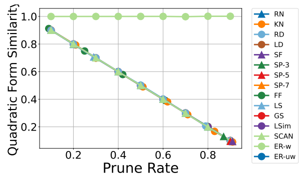

(n) Katz Centrality Precision

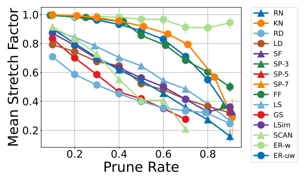

(o) Eigenvector Centrality Precision


(p) PageRank Precision


(q) Number of Communities


(r) Modularity


(s) Quadratic Form Similarity


(t) Max Flow Stretch Factor

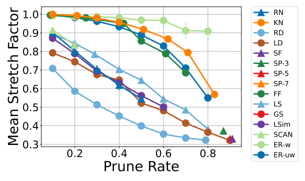
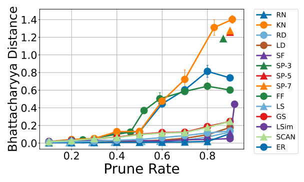
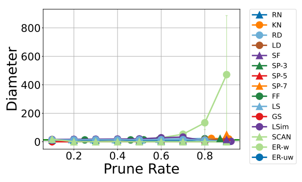
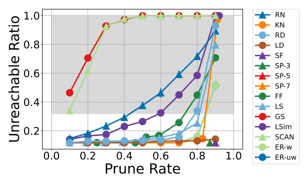
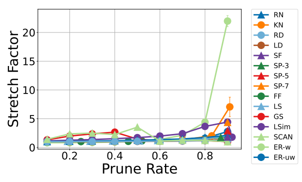

(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

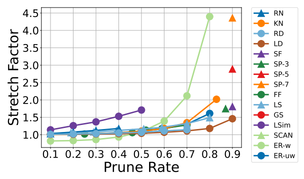
Figure B.11: Metric Evaluation on ca-AstroPh

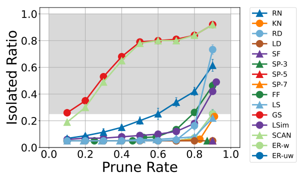

(a) Degree Distribution

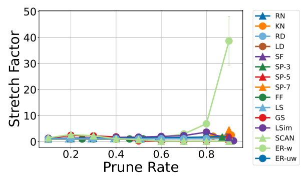

(b) Diameter

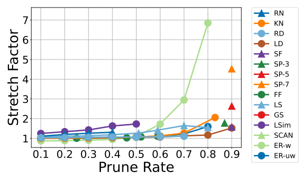

(c) Average SPSP Unreachable Ratio

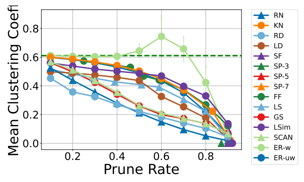

(d) SPSP Stretch Factor

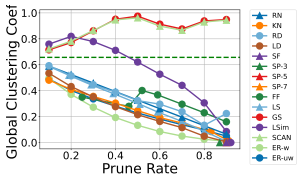

(e) SPSP Stretch Factor with Unreachable Ratio Constraint

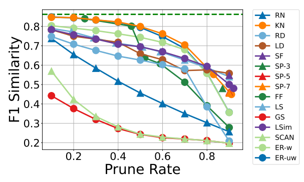

(f) Vertex Isolated Ratio

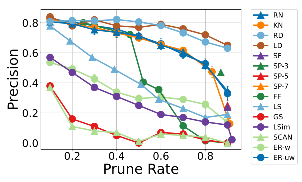

(g) Eccentricity Stretch Factor

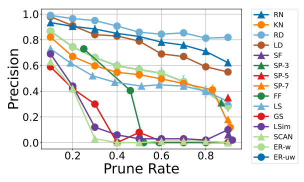

(h) Eccentricity Stretch Factor with Isolated Ratio Constraint

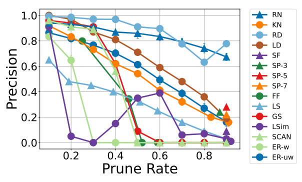

(i) Mean Clustering Coefficient

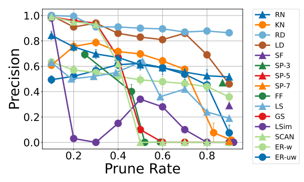

(j) Global Clustering Coefficient

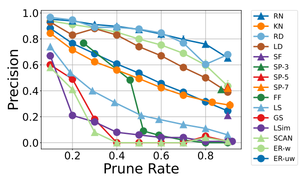

(k) Clustering F1 Similarity

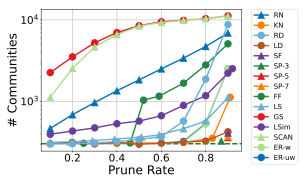

(l) Betweenness Centrality Precision

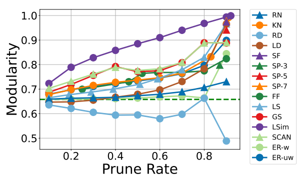

(m) Closeness Centrality Precision

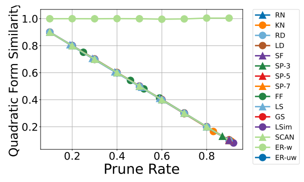

(n) Katz Centrality Precision

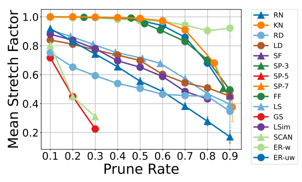

(o) Eigenvector Centrality Precision


(p) PageRank Precision


(q) Number of Communities


(r) Modularity


(s) Quadratic Form Similarity


(t) Max Flow Stretch Factor

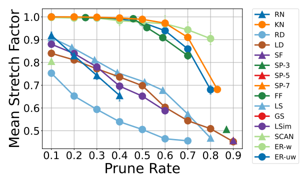
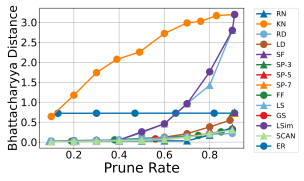
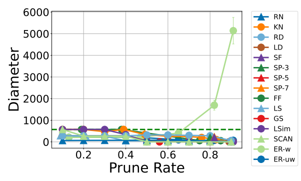

(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

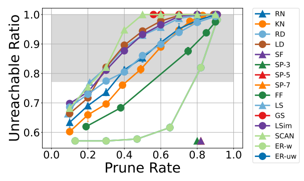
Figure B.12: Metric Evaluation on ca-HepPh



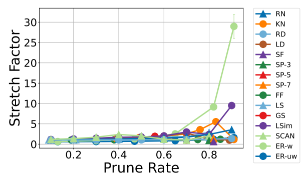
(a) Degree Distribution



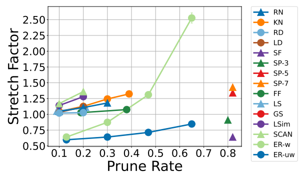
(b) Diameter



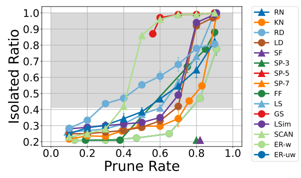
(c) Average SPSP Unreachable Ratio



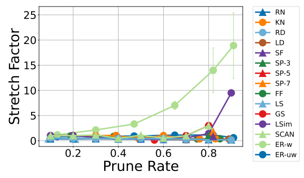
(d) SPSP Stretch Factor



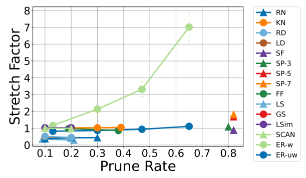
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



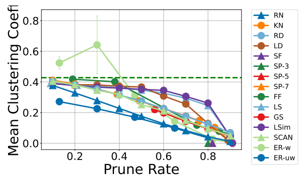
(f) Vertex Isolated Ratio



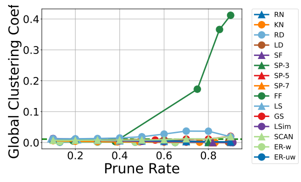
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



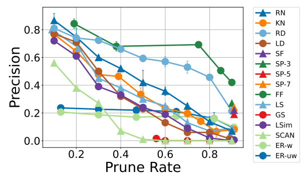
(i) Mean Clustering Coefficient



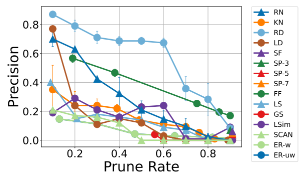
(j) Global Clustering Coefficient

Unsupported for directed graph

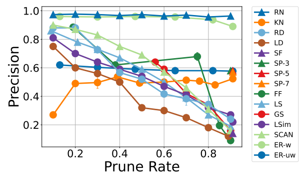
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



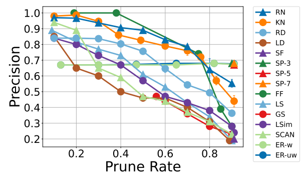
(m) Closeness Centrality Precision



(n) Katz Centrality Precision

Timed out after 24 hours

(o) Eigenvector Centrality Precision



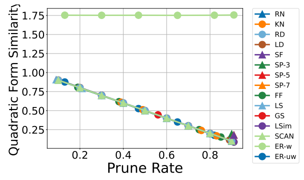
(p) PageRank Precision

Unsupported for directed graph

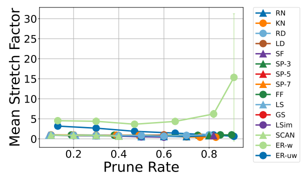
(q) Number of Communities

Unsupported for directed graph

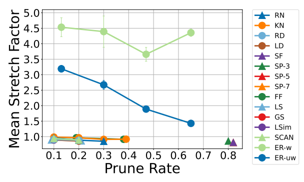
(r) Modularity



(s) Quadratic Form Similarity

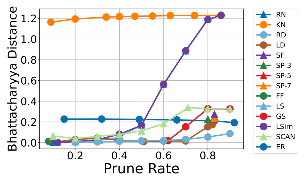


(t) Max Flow Stretch Factor

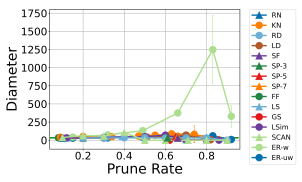


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

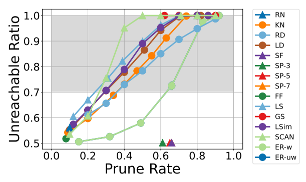
Figure B.13: Metric Evaluation on web-BerkStan



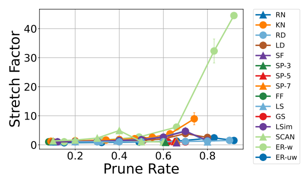
(a) Degree Distribution



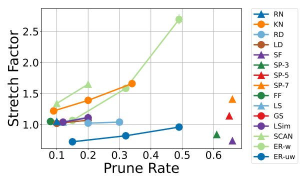
(b) Diameter



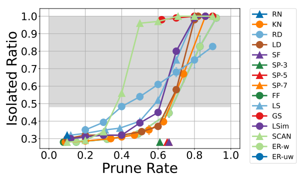
(c) Average SPSP Unreachable Ratio



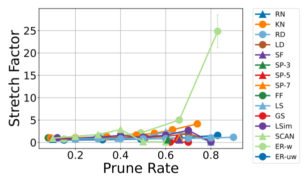
(d) SPSP Stretch Factor



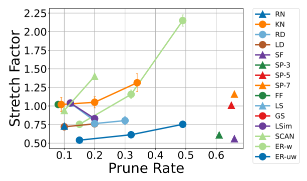
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



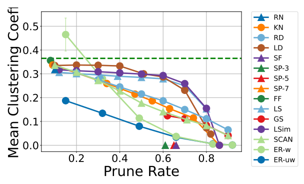
(f) Vertex Isolated Ratio



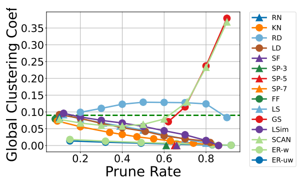
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



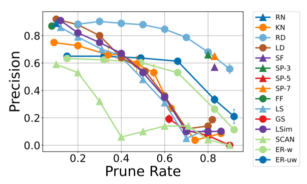
(i) Mean Clustering Coefficient



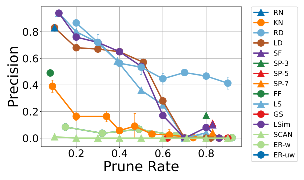
(j) Global Clustering Coefficient

Unsupported
for
directed graph

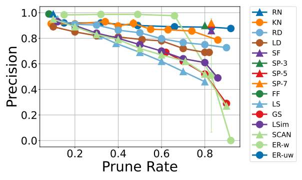
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



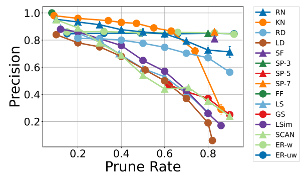
(m) Closeness Centrality Precision



(n) Katz Centrality Precision

Timed out
after
24 hours

(o) Eigenvector Centrality Precision



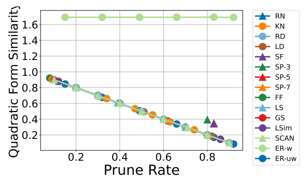
(p) PageRank Precision

Unsupported
for
directed graph

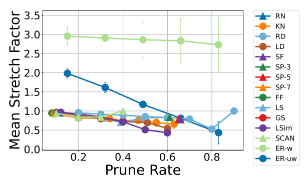
(q) Number of Communities

Unsupported
for
directed graph

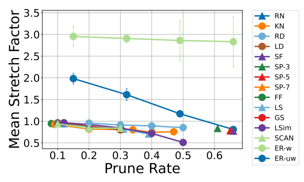
(r) Modularity



(s) Quadratic Form Similarity

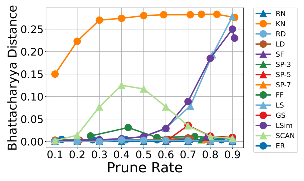


(t) Max Flow Stretch Factor

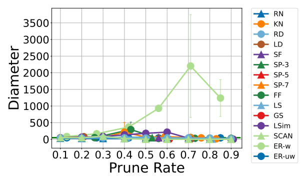


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

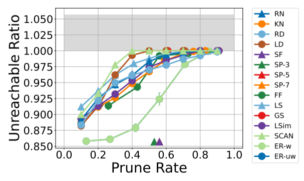
Figure B.14: Metric Evaluation on web-Google



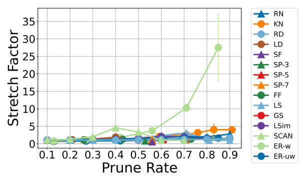
(a) Degree Distribution



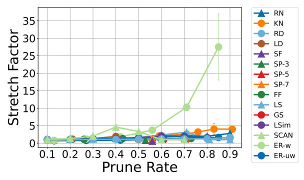
(b) Diameter



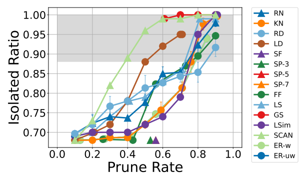
(c) Average SPSP Unreachable Ratio



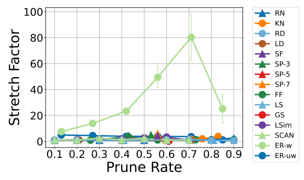
(d) SPSP Stretch Factor



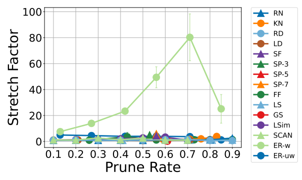
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



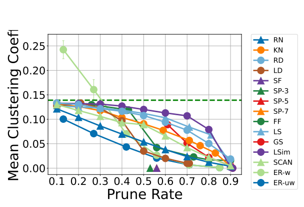
(f) Vertex Isolated Ratio



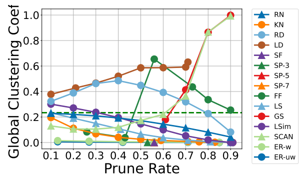
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



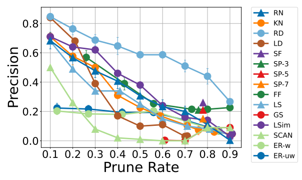
(i) Mean Clustering Coefficient



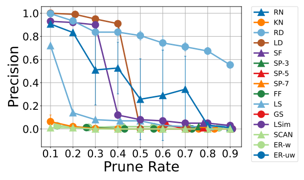
(j) Global Clustering Coefficient

Unsupported for directed graph

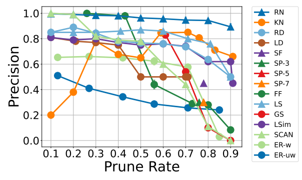
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



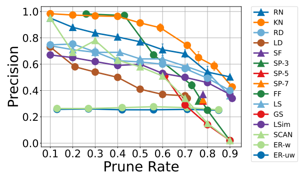
(m) Closeness Centrality Precision



(n) Katz Centrality Precision

Timed out after 24 hours

(o) Eigenvector Centrality Precision



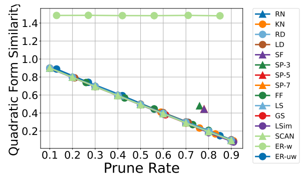
(p) PageRank Precision

Unsupported for directed graph

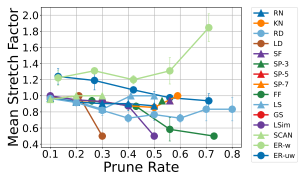
(q) Number of Communities

Unsupported for directed graph

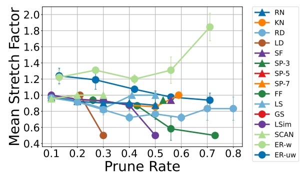
(r) Modularity



(s) Quadratic Form Similarity

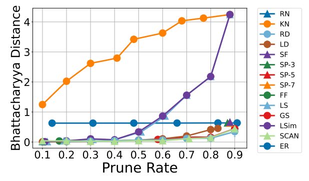


(t) Max Flow Stretch Factor

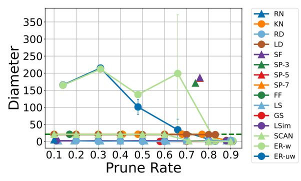


(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

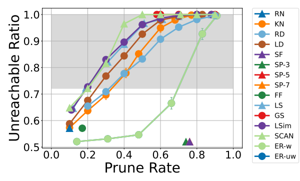
Figure B.15: Metric Evaluation on web-NotreDame



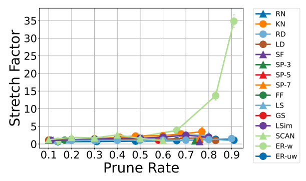
(a) Degree Distribution



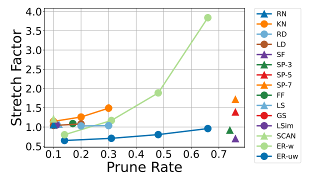
(b) Diameter



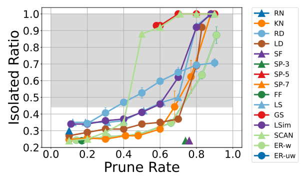
(c) Average SPSP Unreachable Ratio



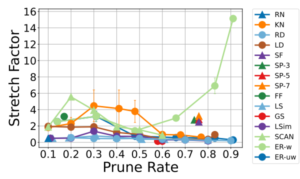
(d) SPSP Stretch Factor



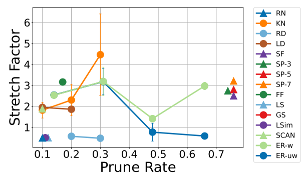
(e) SPSP Stretch Factor with Unreachable Ratio Constraint



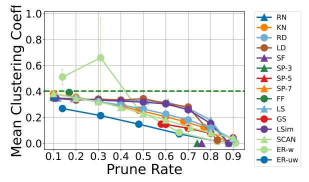
(f) Vertex Isolated Ratio



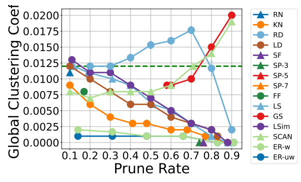
(g) Eccentricity Stretch Factor



(h) Eccentricity Stretch Factor with Isolated Ratio Constraint



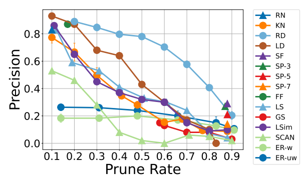
(i) Mean Clustering Coefficient



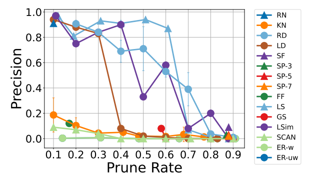
(j) Global Clustering Coefficient

Unsupported
for
directed graph

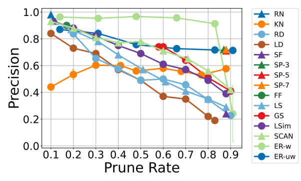
(k) Clustering F1 Similarity



(l) Betweenness Centrality Precision



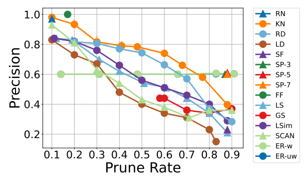
(m) Closeness Centrality Precision



(n) Katz Centrality Precision

Timed out
after
24 hours

(o) Eigenvector Centrality Precision



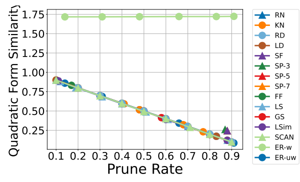
(p) PageRank Precision

Unsupported
for
directed graph

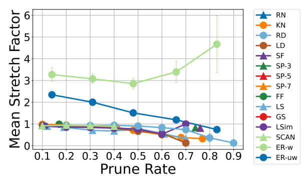
(q) Number of Communities

Unsupported
for
directed graph

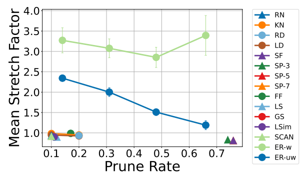
(r) Modularity



(s) Quadratic Form Similarity

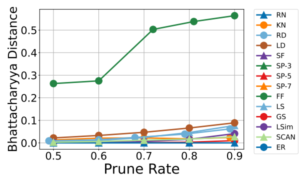


(t) Max Flow Stretch Factor



(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

Figure B.16: Metric Evaluation on web-Stanford



(a) Degree Distribution

Out
of
Memory

(b) Diameter

Timed out
after
24 hours

(c) Average SPSP Unreachable Ratio

Timed out
after
24 hours

(d) SPSP Stretch Factor

Timed out
after
24 hours

(e) SPSP Stretch Factor with Unreachable Ratio Constraint

Out
of
Memory

(f) Vertex Isolated Ratio

Out
of
Memory

(g) Eccentricity Stretch Factor

Out
of
Memory

(h) Eccentricity Stretch Factor with Isolated Ratio Constraint

Out
of
Memory

(i) Mean Clustering Coefficient

Out
of
Memory

(j) Global Clustering Coefficient

Timed out
after
24 hours

(k) Clustering F1 Similarity

Timed out
after
24 hours

(l) Betweenness Centrality Precision

Timed out
after
24 hours

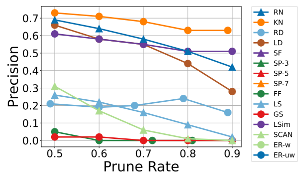
(m) Closeness Centrality Precision

Timed out
after
24 hours

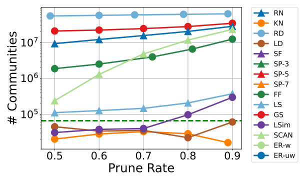
(n) Katz Centrality Precision

Timed out
after
24 hours

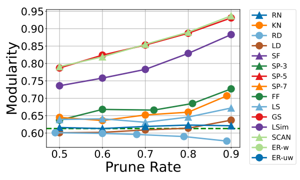
(o) Eigenvector Centrality Precision



(p) PageRank Precision



(q) Number of Communities



(r) Modularity

Timed out
after
24 hours

(s) Quadratic Form Similarity

Out
of
Memory

(t) Max Flow Stretch Factor

Out
of
Memory

(u) Max Flow Stretch Factor with Unreachable Ratio Constraint

Figure B.17: Metric Evaluation on com-friendster

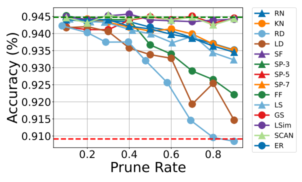


Figure B.18: Clustering GCN Accuracy on Reddit

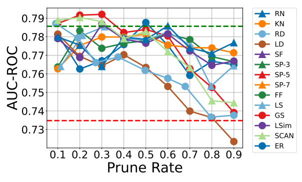


Figure B.19: GraphSAGE Accuracy on ogbn-proteins

BIBLIOGRAPHY

[1] About ffmpeg. https://www.ffmpeg.org/about.html.

[2] Amazon elastic transcoder pricing. https://aws.amazon.com/elastictranscoder/pricing/?nc1=h_ls.

[3] Big buck bunny about page. https://peach.blender.org/about/.

[4] Cisco visual networking index (vni) complete forecast update, 2017–2022. https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1213-business-services-ckn.pdf.

[5] Cpu or gpu: Which processing power you should boost to improve transcoding speed.

[6] Encoding presets for x264. https://dev.beandog.org/x264_preset_reference.html.

[7] Hours of video uploaded to youtube every minute as of may 2019. https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute/.

[8] Intel® vtune™ profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[9] Linux encoding. https://sites.google.com/site/linuxencoding/x264-ffmpeg-mapping.

[10] Perf wiki. https://perf.wiki.kernel.org/index.php.

[11] Understanding how general exploration works in intel® vtune™ amplifier.

[12] Understanding rate control modes (x264, x265, vpx). https://slhck.info/video/2017/03/01/rate-control.html.

[13] Video compression picture types. https://en.wikipedia.org/wiki/Video_compression_picture_types.

[14] Video space calculator. https://www.digitalrebellion.com/webapps/videocalc?format=uncompressed_8_1080&frame_rate=f30&length=1&length_type=seconds.

[15] [x264-devel] making sense out of x264 rate control methods. https://mailman.videolan.org/pipermail/x264-devel/2010-February/006934.html.

[16] x264 homepage. http://www.videolan.org/developers/x264.html.

[17] Spanning tree. https://en.wikipedia.org/wiki/Spanning_tree, Nov 2022.

[18] Tree (graph theory). https://en.wikipedia.org/wiki/Tree_(graph_theory), Nov 2022.

[19] Clustering coefficient. https://en.wikipedia.org/wiki/Clustering_coefficient, Feb 2023.

[20] Connected graph. https://mathworld.wolfram.com/ConnectedGraph.html, 2023.

[21] Cut (graph theory). https://en.wikipedia.org/wiki/Cut_(graph_theory), Feb 2023.

[22] Eigenvector centrality. https://en.wikipedia.org/wiki/Eigenvector_centrality, Jan 2023.

[23] Sergi Abadal et al. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Comput. Surv.*, 54(9), oct 2021.

[24] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., USA, 1993.

[25] Ayaz Akram and Lina Sawalha. A survey of computer architecture simulation techniques and tools. *IEEE Access*, PP:1–1, 05 2019.

[26] Antonino Albanese, Paolo Secondo Crosta, Claudio Meani, and Pietro Paglierani. Gpu-accelerated video transcoding unit for multi-access edge computing scenarios. In *Proceeding of ICN*, 2017.

[27] Ingo Althöfer, Gautam Das, David Dobkin, and Deborah Joseph. Generating sparse spanners for weighted graphs. In John R. Gilbert and Rolf Karlsson, editors, *SWAT 90*, pages 26–37, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[28] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016.

[29] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a predictive model of quality of experience for internet video. *ACM SIGCOMM Computer Communication Review*, 43(4):339–350, 2013.

[30] Gerassimos Barlas. Cluster-based optimized parallel video transcoding. In *Parallel Computing*, pages 226–244. Elsevier, 2012.

[31] Peter W. Battaglia et al. Interaction networks for learning about objects, relations and physics. *CoRR*, abs/1612.00222, 2016.

[32] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In T. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems*, volume 14. MIT Press, 2001.

[33] Elisabetta Bergamini, Michele Borassi, Pierluigi Crescenzi, Andrea Marino, and Henning Meyerhenke. Computing top-k closeness centrality faster in unweighted graphs, 2017.

[34] Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries, 2023.

[35] A. Bhattacharyya. On a measure of divergence between two multinomial populations. *Sankhyā: The Indian Journal of Statistics (1933-1960)*, 7(4):401–406, 1946.

[36] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[37] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, oct 2008.

[38] Geoff Boeing. Measuring the complexity of urban form and design. *Urban Design International*, 23:281–292, 11 2018.

[39] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier, New York, 1976.

[40] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30:107–117, 1998.

[41] RONALD S. BURT. *Structural Holes: The Social Structure of Competition*. 1992.

[42] 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.

[43] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[44] Rudrasis Chakraborty et al. Manifoldnet: A deep neural network for manifold-valued data with applications. *IEEE TPAMI*, pages 1–1, 2020.

[45] Cen Chen et al. Regnn: A redundancy-eliminated graph neural networks accelerator. In *2022 HPCA*, pages 429–443, 2022.

[46] Dehao Chen, Tipp Moseley, and David Xinliang Li. Autofdo: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 12–23. IEEE, 2016.

[47] Jie Chen et al. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *ICLR*, 2018.

[48] Yuhan Chen, Alireza Khadem, Xin He, Nishil Talati, Tanvir Ahmed Khan, and Trevor Mudge. Pedal: A power efficient gcn accelerator with multiple dataflows. In *Proceedings of the 26th Design, Automation, and Test in Europe (DATE) conference*, DATE 2023, April 2023.

[49] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 732–737, 2002.

[50] Andrew R. Curtis, Tommy Carpenter, and S. Keshav. Rewire: An optimization-based framework for data center network design. 2011.

[51] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011.

[52] Jan De Cock, Aditya Mavlankar, Anush Moorthy, and Anne Aaron. A large-scale video codec comparison of x264, x265 and libvpx for practical vod applications. In *Applications of Digital Image Processing XXXIX*, volume 9971, page 997116. International Society for Optics and Photonics, 2016.

[53] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. 2016.

[54] Andac Demir, Toshiaki Koike-Akino, Ye Wang, Masaki Haruna, and Deniz Erdogmus. Eeg-gnn: Graph neural networks for classification of electroencephalogram (eeg) signals. In *2021 43rd Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, pages 1061–1067, 2021.

[55] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, dec 1959.

[56] Ramesh K. Sitaraman Dilip Kumar Krishnappa, Michael Zink. Optimizing the video transcoding workflow in content delivery networks. In *Proceedings of the 6th ACM Multimedia Systems Conference*, pages 37–48. ACM, 2015.

[57] Richard Draves, Jitendra Padhye, and Brian Zill. Comparison of routing metrics for static multi-hop wireless networks. *SIGCOMM Comput. Commun. Rev.*, 34(4):133–144, aug 2004.

[58] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[59] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861–874, 2006. ROC Analysis in Pattern Recognition.

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

[61] Miroslav Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(2):298–305, 1973.

[62] Jack McKay Fletcher and Thomas Wennekers. From structure to activity: Using centrality measures to predict neuronal activity. *International Journal of Neural Systems*, 28(02):1750013, 2018.

[63] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[64] Linton Freeman. The development of social network analysis. 01 2004.

[65] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. C3: Internet-scale control plane for video quality optimization. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 131–144, 2015.

[66] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, page 90–100, USA, 2008. Society for Industrial and Applied Mathematics.

[67] Tong Geng et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *MICRO*, pages 922–936, 2020.

[68] Tong Geng et al. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO*, 2021.

[69] Palash Goyal and Emilio Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[70] Alex Graves et al. Multi-dimensional recurrent neural networks. In *Artificial Neural Networks - ICANN*, 2007.

[71] Matthew W. Hahn and Andrew D. Kern. Comparative Genomics of Centrality and Essentiality in Three Eukaryotic Protein-Interaction Networks. *Molecular Biology and Evolution*, 22(4):803–806, 12 2004.

[72] Michael Hamann, Gerd Lindner, Henning Meyerhenke, Christian L. Staudt, and Dorothea Wagner. Structure-preserving sparsification methods for social networks, 2016.

[73] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

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

[75] Weihua Hu et al. Open graph benchmark: Datasets for machine learning on graphs, 2020.

[76] 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.

[77] Yicheng Huang, An Vu Tran, and Ye Wang. A workload prediction model for decoding mpeg video and its application to workload-scalable transcoding. In *Proceedings of the 15th ACM international conference on Multimedia*, pages 952–961, 2007.

[78] Yu-Wen Huang, Bing-Yu Hsieh, Tung-Chien Chen, and Liang-Gee Chen. Analysis, fast algorithm, and vlsi architecture design for h. 264/avc intra frame coder. *IEEE Transactions on Circuits and systems for Video Technology*, 15(3):378–401, 2005.

[79] Donald J. Jacobs, A.J. Rader, Leslie A. Kuhn, and M.F. Thorpe. Protein flexibility predictions using graph theory. *Proteins: Structure, Function, and Bioinformatics*, 44(2):150–165, 2001.

[80] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The large-scale organization of metabolic networks. *Nature*, 407(6804):651–654, oct 2000.

[81] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. {CFA}: A practical prediction system for video qoe optimization. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 137–150, 2016.

[82] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18:39–43, 1953.

[83] Myunghwan Kim and Jure Leskovec. Modeling social networks with node attributes using the multiplicative attribute graph model, 2011.

[84] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15:45–49, 2016.

[85] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016.

[86] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[87] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. 1956.

[88] Y. Lecun et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[89] Stefan Lederer, Christopher Müller, and Christian Timmerer. Dynamic adaptive streaming over http dataset. In *Proceedings of the 3rd multimedia systems conference*, pages 89–94, 2012.

[90] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1:2, 2006.

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

[92] Zekun Li, Zeyu Cui, Shu Wu, Xiaoyu Zhang, and Liang Wang. Fi-gnn: Modeling feature interactions via graph neural networks for ctr prediction. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 539–548, New York, NY, USA, 2019. Association for Computing Machinery.

[93] Zhenhua Li, Yan Huang, Gang Liu, Fuchen Wang, Zhi-Li Zhang, and Yafei Dai. Cloud transcoder: Bridging the format and resolution gap between internet videos and mobile devices. In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, pages 33–38, 2012.

[94] Shengwen Liang et al. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE TC*, 2021.

[95] Chia-Wen Lin and Yuh-Reuy Lee. Fast algorithms for dct-domain video transcoding. In *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*, volume 1, pages 421–424. IEEE, 2001.

[96] Song Lin, Xinfeng Zhang, Qin Yu, Honggang Qi, and Siwei Ma. Parallelizing video transcoding with load balancing on cloud computing. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pages 2864–2867. IEEE, 2013.

[97] Yaning Liu, Joost Geurts, Jean-Charles Point, Stefan Lederer, Benjamin Rainer, Christopher Müller, Christian Timmerer, and Hermann Hellwagner. Dynamic adaptive streaming over ccn: A caching and overhead analysis. In *2013 IEEE international conference on communications (ICC)*, pages 3629–3633. IEEE, 2013.

[98] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[99] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A. Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. Vbench: Benchmarking video transcoding in the cloud. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 797–809, New York, NY, USA, 2018. Association for Computing Machinery.

[100] R. Duncan Luce and Albert D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14:95–116, 1949.

[101] Hong-Wu Ma and An-Ping Zeng. The connectivity structure, giant strong component and centrality of metabolic networks. *Bioinformatics*, 19(11):1423–1430, 07 2003.

[102] Damien Magoni and Jean Jacques Pansiot. Analysis of the autonomous system network topology. *SIGCOMM Comput. Commun. Rev.*, 31(3):26–37, jul 2001.

[103] Vijini Mallawaarachchi. Evaluating clustering results, Oct 2020.

[104] Amrita Mazumdar, Brandon Haynes, Magda Balazinska, Luis Ceze, Alvin Cheung, and Mark Oskin. Perceptual compression for video storage and processing systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 179–192, 2019.

[105] Loren Merritt. X264: A high performance h.264/avc encoder. 2006.

[106] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[107] Naveen Muralimanohar and Rajeev Balasubramonian. Cacti 6.0: A tool to understand large caches.

[108] Allan H. Murphy. The finley affair: A signal event in the history of forecast verification. *Weather and Forecasting*, 11(1):3 – 20, 1996.

[109] Daniel Müllner. Modern hierarchical, agglomerative clustering algorithms, 2011.

[110] M. E. J. Newman. *Mathematics of Networks*, pages 1–8. Palgrave Macmillan UK, London, 2016.

[111] Mark EJ Newman. Coauthorship networks and patterns of scientific collaboration. *Proceedings of the national academy of sciences*, 101(suppl_1):5200–5205, 2004.

[112] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic*, NIPS'01, page 849–856, Cambridge, MA, USA, 2001. MIT Press.

[113] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking : Bringing order to the web. In *The Web Conference*, 1999.

[114] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. Using graph theory to analyze biological networks. *BioData mining*, 4:1–27, 2011.

[115] F. Pedregosa et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[116] Tiago P. Peixoto. The graph-tool python library. *figshare*, 2014.

[117] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006. Citeseer, 2006.

[118] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[119] Varun Ramesh, Shivanee Nagarajan, Jason J. Jung, and Saswati Mukherjee. Max-flow min-cut algorithm with application to road networks. *Concurrency and Computation: Practice and Experience*, 29(11):e4099, 2017. e4099 cpe.4099.

[120] Theodoros Rapanos. What makes an opinion leader: Expertise vs popularity. *Games and Economic Behavior*, 138:355–372, 2023.

[121] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[122] Amedeo R. Odoni Richard C. Larsona. Urban operations research. 1981.

[123] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification, 2019.

[124] Veeranjaneyulu Sadhanala, Yu-Xiang Wang, and Ryan J. Tibshirani. Graph sparsification approaches for laplacian smoothing. In *International Conference on Artificial Intelligence and Statistics*, 2016.

[125] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 721–732, New York, NY, USA, 2011. Association for Computing Machinery.

[126] Franco Scarselli et al. The graph neural network model. *IEEE Transactions on Neural Networks*, 2009.

[127] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[128] Jan Scheurer and Sergio Porta. Centrality and connectivity in public transport networks and their significance for transport sustainability in cities. 07 2006.

[129] Jonathan Shlomi, Peter Battaglia, and Jean-Roch Vlimant. Graph neural networks in particle physics. *Machine Learning: Science and Technology*, 2(2):021001, dec 2020.

[130] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543, Los Alamitos, CA, USA, feb 2018. IEEE Computer Society.

[131] Daniel Spielman. Laplacians.jl. `https://github.com/danspielman/Laplacians.jl`, 2023.

[132] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. *SIAM Journal on Computing*, 40(6):1913–1926, 2011.

[133] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis, 2014.

[134] Yi Sun, Xiaoqi Yin, Junchen Jiang, Vyas Sekar, Fuyuan Lin, Nanshu Wang, Tao Liu, and Bruno Sinopoli. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 272–285, 2016.

[135] Minyong Sung, Minwoo Kim, Minsik Kim, and Won Woo Ro. Accelerating hevc transcoder by exploiting decoded quadtree. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*, pages 1–2. IEEE, 2014.

[136] Damian Szklarczyk, Annika Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda Doncheva, John Morris, Peer Bork, Lars Jensen, and Christian von Mering. String v11: protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47, 11 2018.

[137] Frank W. Takes and Walter A. Kosters. Computing the eccentricity distribution of large graphs. *Algorithms*, 6(1):100–118, 2013.

[138] Nishil Talati et al. A deep dive into understanding the random walk-based temporal graph learning. In *IISWC*, 2021.

[139] Nishil Talati, Haojie Ye, Sanketh Vedula, Kuan-Yu Chen, Yuhan Chen, Daniel Liu, Yichao Yuan, David Blaauw, Alex Bronstein, Trevor Mudge, and Ronald Dreslinski. Mint: An accelerator for mining temporal motifs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1270–1287, 2022.

[140] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[141] Elli Voudigari, Nikos Salamanos, Theodore Papageorgiou, and Emmanuel J. Yannakoudakis. Rank degree: An efficient algorithm for graph sampling. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 120–129, 2016.

[142] Xiao Wang, Deyu Bo, Chuan Shi, Shaohua Fan, Yanfang Ye, and Philip S. Yu. A survey on heterogeneous graph embedding: Methods, techniques, applications and sources. *IEEE Transactions on Big Data*, 9(2):415–436, 2023.

[143] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[144] L. Wei, J. Cai, C. H. Foh, and B. He. Qos-aware resource allocation for video transcoding in clouds. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(1):49–61, 2017.

[145] Jiangtao Wen, Max Luttrell, and John Villasenor. Trellis-based rd optimal quantization in h. 263+. *IEEE Transactions on Image Processing*, 9(8):1431–1434, 2000.

[146] R. Wickman, X. Zhang, and W. Li. A generic graph sparsification framework using deep reinforcement learning. In *2022 IEEE International Conference on Data Mining (ICDM)*, pages 1221–1226, Los Alamitos, CA, USA, dec 2022. IEEE Computer Society.

[147] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[148] Jun Xin, Chia-Wen Lin, and Ming-Ting Sun. Digital video transcoding. *Proceedings of the IEEE*, 93(1):84–97, 2005.

[149] Mengjia Xu. Understanding graph embedding methods and their applications. *SIAM Review*, 63(4):825–853, 2021.

[150] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. Scan: A structural clustering algorithm for networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, page 824–833, New York, NY, USA, 2007. Association for Computing Machinery.

[151] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 495–511, 2020.

[152] M. Yan et al. Hygcn: A gcn accelerator with hybrid architecture. In *HPCA*, 2020.

[153] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.

[154] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[155] Haoran You et al. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *HPCA*, 2022.

[156] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 236–252, 2018.

[157] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 793–803, New York, NY, USA, 2019. Association for Computing Machinery.

[158] X. Zhang, T. Huang, Y. Tian, M. Geng, S. Ma, and W. Gao. Fast and efficient transcoding based on low-complexity background modeling and adaptive block classification. *IEEE Transactions on Multimedia*, 15(8):1769–1785, 2013.

[159] Cheng Zheng, Bo Zong, Wei Cheng, Dongjin Song, Jingchao Ni, Wenchao Yu, Haifeng Chen, and Wei Wang. Robust graph representation learning via neural sparsification. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 11458–11468. PMLR, 13–18 Jul 2020.

[160] Fan Zhou, Chengtai Cao, Kunpeng Zhang, Goce Trajcevski, Ting Zhong, and Ji Geng. Meta-gnn: On few-shot node classification in graph meta-learning. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 2357–2360, New York, NY, USA, 2019. Association for Computing Machinery.

[161] Jie Zhou et al. Graph neural networks: A review of methods and applications. *AI Open*, 2020.

[162] Chenyi Zhuang and Qiang Ma. Dual graph convolutional networks for graph-based semi-supervised classification. In *WWW*, 2018.

[163] Zhenyun Zhuang and Chun Guo. Building cloud-ready video transcoding system for content delivery networks (cdns). In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 2048–2053. IEEE, 2012.