Decoupling Scheduling and Storage Formats for Balanced Graph Processing on a GPU

Shinnung Jeong

The Graduate School
Yonsei University
Department of Electrical and Electronic Engineering

Decoupling Scheduling and Storage Formats for Balanced Graph Processing on a GPU

A Dissertation Submitted
to the Department of Electrical and Electronic Engineering
and the Graduate School of Yonsei University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering

Shinnung Jeong

December 2024

This certifies that the Dissertation of Shinnung Jeong is approved.

Thesis Supervisor	Prof. Hanjun Kim
Thesis Committee Member	Prof. Won Woo Ro
Thesis Committee Member	Prof. William Jinho Song
Thesis Committee Member	Prof. Hyesoon Kim
Thesis Committee Member	Prof. Youngsok Kim

The Graduate School
Yonsei University
December 2024

ACKNOWLEDGEMENTS

무엇보다도 학부생 시절부터 박사 과정 마무리까지 연구자의 길에 대해 좋은 가르침을 주신 김한준 교수님께 진심으로 감사의 말씀을 드리고 싶습니다. 항상 저에게 새로운 길을 보여주시고 기회를 주신 교수님의 헌신에 깊이 감사드립니다. 10년이라는 긴 시간 동안 교수님께서 주신 소중한 가르침은 제 연구의 기반이 되어, 제 연구 방식의 단단한 뿌리가 되었다고 생각합니다. 앞으로도 교수님의 가르침을 마음 깊이 새겨, 좋은 나무와 같은 연구자가 될 수 있도록 노력하겠습니다.

또한, 저에게 새로운 연구 방식과 삶에 대해 가르쳐주신 김혜순 교수님께도 깊이 감사드립니다. 교수님께서 주신 소중한 기회 덕분에, 연구적으로도 인간적으로도 한층 성장할 수 있었으며, 넓은 세상에 대한 경험을 바탕으로 대학원 여정을 마친 이후의 미래를 꿈꿀 수 있게 되었습니다. 아직 많이 부족한 사람이지만, 교수님께서 보여주신 모습을 본받아 좋은 연구자이자 좋은 사람이 되도록 노력하겠습니다.

아직 어리숙하던 저의 첫 연구 과정에서 큰 도움을 주신 김영석 교수님과 이진호 교수님께도 감사의 말씀을 드리고 싶습니다. 처음 그래프 연구를 시작하던 때를 돌이켜보면 참 부족했던 점이 많았습니다. 그럼에도 불구하고, 교수님들께서 다양한 관점에서 주신 소중한 피드백과 조언 덕분에 그래프 연구를 발전시켜 나갈 수 있었다고 생각합니다.

마지막으로, 바쁘신 와중에도 귀중한 시간을 흔쾌히 내어 학위 논문 심사를 맡아주신 노원우 교수님, 송진호 교수님, 김혜순 교수님, 김영석 교수님께 깊이 감사드립니다. 심사위원 교수님들께서 제 논문에 대해 소중한 피드백과 조언을 주신 덕분에 더 좋은 졸업 논문을 완성할 수 있었습니다. 교수님들께서 주신 날카로운 지적과 조언을 잊지 않고, 항상 발전하는 연구자가 되도록 하겠습니다.

대학원 여정 동안 함께한 컴파일러 최적화 연구실 사람들에게도 감사의 마음을 전하고자 합니다. 항상 조언과 위로를 아끼지 않으신 선배님들, 허선영 교수님, 이경민 박사님, 김봉준 박사님, 김창수 박사님, 송승빈 박사님, 조성준 선배님 덕분에 어려운 서울살이를

이겨내고 좋은 연구를 할 수 있었습니다. 학부생 때부터 함께한 이용우 박사님, 그리고 제 연구에 큰 도움을 주신 최희림, 이주민에게도 특별히 감사의 말씀을 전하고 싶습니다. 연구실에서 함께했던 이재호, 김근우, 최희림, 윤성우, 박현준, 권현호, 이찬, 정건모, 정해은, 그리고 남주현 선생님께도 감사드립니다.

그리고 Georgia Tech에서 함께한 분들에게도 감사의 인사를 전하고 싶습니다. High Performance Architecture Lab에서 함께했던 Prof. Blaise Tine, Dr. Seonjin Na, Dr. Yonghae Kim, Dr. Jaewon Lee, Euna Kim, Jiashen Cao, Nicholas Parnenzini, Andrei Bersatti, Saurabh Singh, Ruobing Han, Xueyang Liu, Sam Jijina, Anurag Kar, Chihyo Ahn, Eric Lorimer, Liam Cooper, Euijun Chung, Michael Goldstein에게 감사의 말씀을 전합니다. 함께 시간을 보낸 정건화 박사님, 박미선, 이연주에게도 감사의 말을 전하고 싶습니다.

제 20대를 이야기할 때 빼놓을 수 없는 소중한 대학 친구들에게도 이 기회를 빌려 감사의 마음을 전하고 싶습니다. 대학교 때부터 함께한 친구들과의 시간이 없었다면, 저는 이 여정을 결코 끝맺지 못했을 것입니다. 저와 수많은 시간을 함께해준 유리, 진우, 선우, 지호, 현명, 예찬, 그리고 다른 친구들에게 감사의 말씀을 드립니다. 특히 가장 많은 시간을 함께 보내고, 미국까지 와서 휴가를 함께해준 유리에게 특별히 감사의 인사를 전합니다. 각자 자신의 길을 향해 떠나는 갈림길에 선 지금, 지금까지처럼 동고동락하지는 못할지라도 친구들의 앞길에 좋은 일만 가득하길 진심으로 바랍니다.

마지막으로, 사랑하는 부모님께 무한한 감사의 말씀을 드리고 싶습니다. 부모님의 사랑과 지지가 있었기에, 제가 박사라는 교육 과정의 끝이자 연구자로서의 시작점에 도달할 수 있었습니다. 이제 또 다른 여정을 준비하는 저에게 여전히 큰 지지를 보내주시는 부모님께, 말주변이 없어 마음을 잘 표현하지 못하지만 항상 사랑한다고 전하고 싶습니다.

이렇게 글을 쓰고 보니, 참으로 감사한 분들이 많다는 생각에 제가 복받은 사람이라는 것을 다시금 느낍니다. 좋은 분들과 함께했기에 이 여정을 즐겁고 행복하게 마무리할 수 있었습니다. 다시 한번 진심으로 감사드립니다.

TABLE OF CONTENTS

LIST OF FIGURES v

LIST OF TABLES viii

LIST OF ALGORITHMS ix

ABSTRACT x

1. INTRODUCTION 1

    1.1 Graph Processing on a GPU 1

    1.2 Contribution 4

    1.3 Organization 6

2. GRAPH PROCESSING MODEL AND ABSTRACTION FOR GRAPH PROCESSING ON A GPU 7

    2.1 Component of Graph Processing on GPU 8

        2.1.1 Graph Algorithm for GPU 9

        2.1.2 Graph Schedule for GPU 9

        2.1.3 Storage Format for GPU 12

    2.2 Related Work and Motivation of Decoupling 14

    2.3 Graph Processing Abstraction 19

        2.3.1 Schedule Abstraction 21

        2.3.2 Storage Format Abstraction 23

        2.3.3 Algorithm Abstraction 27

        2.3.4 Assembling Abstract Interfaces 28

2.4 GRAssembler Framework 30
2.4.1 Compiler Optimization 33
2.4.2 Tuning Space of GRAssembler 34
2.5 Evaluation 38
2.5.1 Overall Performance 39
2.5.2 Tuning Performance and Cost 43
2.5.3 Line of Code Analysis 44
2.5.4 Abstraction Overhead 44
2.5.5 Case Study 1: Extendability 45
2.5.6 Case Study 2: Impact of Tuning 47
2.5.7 End-to-End Comparison with Existing CPU and GPU Framework 48
2.6 Summary 51
3. CR^2: COMMUNITY-AWARE COMPRESSED REGULAR STORAGE FORMAT 52
3.1 Necessity of A New Storage Format 52
3.2 In-depth Analysis of Related Storage Formats 54
3.3 Design of CR^2 59
3.3.1 Design Goals of CR^2 60
3.3.2 Community-aware Subgraph 60
3.3.3 Degree-Ordered Subgraph (Degree-*n*) 65
3.3.4 Expand List for Split Source Access 67
3.4 Processing CR^2 Graph 68

3.4.1 Building CR^2 . . . 69
3.4.2 Launching Kernels with CR^2 . . . 71
3.4.3 Adding and Deleting an Edge . . . 73
3.4.4 Finding Neighbors of a Vertex . . . 75
3.5 Evaluation . . . 75
3.5.1 Execution Time . . . 77
3.5.2 Memory Efficiency . . . 79
3.5.3 Performance using Expand List . . . 80
3.5.4 Storage Format Build Time . . . 82
3.5.5 Performance Comparison of Push and Pull . . . 82
3.6 Summary . . . 84
4. SPARSEWEAVER: MICROARCHITECTURE FOR ACCELERATING SCHEDULE . . . 85
4.1 Necessity of Hardware Acceleration for Schedule . . . 85
4.2 Revisit Graph Processing on GPU in terms of Schedule . . . 87
4.3 Deep Dive into Related Schedules . . . 88
4.4 Hardware/Software Co-Design . . . 95
4.4.1 Software-based Schedule Abstraction . . . 95
4.4.2 Weaver Logic Design . . . 97
4.4.3 SparseWeaver Design . . . 100
4.4.4 Assembling Design Decisions . . . 104
4.5 SparseWeaver Framework . . . 106
4.5.1 SparseWeaver Instruction . . . 107

4.5.2 SparseWeaver Compiler and Runtime 109
4.5.3 Weaver Implementation 111
4.6 Evaluation of SparseWeaver 112
4.6.1 Comparison with Software-based Schemes 113
4.6.2 Skewness Sensitivity 115
4.6.3 Effect of Memory Configuration 116
4.6.4 Effect of Work Table Access 116
4.6.5 Cache and Memory Analysis 117
4.6.6 Hardware Overhead 118
4.6.7 Push and Pull Breakdown 119
4.6.8 Case Study 1: Existing Hardware-based Scheme 121
4.6.9 Case Study 2: Performance with GCN 123
4.6.10 Case Study 3: Comparison with GRAssembler 124
4.7 Discussion 125
4.7.1 Integrating into GRAssembler 125
4.7.2 General Usage of SparseWeaver 126
4.8 Summary 126
5. CONCLUSION 128
REFERENCES 130
ABSTRACT IN KOREAN 143

LIST OF FIGURES

2.1 Existing schedules 10

2.2 Existing storage formats 12

2.3 Comparison of processing times for various schedules and storage formats 16

2.4 The development model and example codes used in existing frameworks 18

2.5 Graph processing model in GRAssembler 19

2.6 Development model in the GRAssembler framework 29

2.7 Overview of GRAssembler 37

2.8 Performance evaluation of GRAssembler 40

2.9 Extendability of GRAssembler 45

2.10 Performance impact of tuning options 48

3.1 Existing storage formats designed for GPU 55

3.2 Inefficency of CSR-based graph processing 57

3.3 Vertex ID representation in the intra-cluster graph 63

3.4 Graph visualization of Randome-, Community-based reordered graph and concept in CR^2 63

3.5 Design process of CR^2 64

3.6 Source pointer representation in the expand list 68

3.7 Addition and deletion of the edge in CR^2 74

3.8 Memory efficiency of CR^2 79

3.9 Performance effect of expand list 80

3.10 Memory usage of expand list 81

3.11 Build time of CR^2 83

4.1 Overview of SparseWeaver 87

4.2 Expected warp iteration and performance of software-based schedule 90

4.3 Performance comparison with existing software schedule in two Nvidia GPUs 91

4.4 Stall breakdown and warp per instruction result of schedule 92

4.5 Disassemble graph processing by abstracting software-based schedule 95

4.6 SparseWeaver hardware logic called Weaver 98

4.7 SparseWeaver execution flow 105

4.8 SparseWeaver system overview 108

4.9 SparseWeaver gather kernel 110

4.10 Performance evaluation of SparseWeaver 114

4.11 Skewness sensitivity of SparseWeaver 115

4.12 Memory configuration effect 117

4.13 Performance effect of cache existence 117

4.14 Cache size effect 118

4.15 Work table access overhead 119

4.16 Block utilization of Vortex GPU and SparseWeaver 120

4.17 Execution cycle breakdown by direction 121

4.18 Comparison with existing hardware-based schedules 122

4.19 Performance of GCN operators 123

LIST OF TABLES

2.1 Comparsion of existing GPU-based graph processing framework . . 15

2.2 GRAssembler schedule interface specifications. 23

2.3 GRAssembler storage format interface specifications. 24

2.4 GRAssembler algorithm interface specifications. 24

2.5 The optimization space available in modern GPU-based graph processing frameworks 35

2.6 The optimal options of Figure 2.8 41

2.7 End-to-end performance comparison 49

3.1 Theoretically required memory sizes of different storage formats . 58

3.2 Terminology about a target graph and CR^2 61

3.3 The worst time complexity of basic graph operations of CR^2 73

3.4 Dataset specification and clustering 76

3.5 Performance evaluation of CR^2 77

3.6 Build time of CR^2 82

4.1 Comparison of schedule implementation details 89

4.2 SparseWeaver instructions 109

4.3 Graph dataset information 112

4.4 The area overhead of SparseWeaver 120

4.5 Speedup over VM Comparison using GRAssembler 124

LIST OF ALGORITHMS

2.1 Processing model of GRAssembler 31

3.1 CR^2 builder 69

3.2 Degree counting function of CR^2 70

3.3 Edge rearrange function of CR^2 71

3.4 Example of CR^2 processing 72

4.1 The GPU hash lookup 126

ABSTRACT

Decoupling Scheduling and Storage Formats for Balanced Graph Processing on a GPU

Only with a proper schedule and a proper storage format, a graph algorithm can be efficiently processed on GPUs. Existing GPU graph processing frameworks try to find an optimal schedule and storage format for an algorithm via iterative search, but they fail to find the optimal configuration because their schedules and storage formats are tightly coupled in their processing models. Moreover, their tightly coupled schedules and storage formats make it difficult for developers to extend the tuning space. However, implementing all possible combinations of the schedule and storage format would impose a substantial implementation burden. Therefore, clear decoupling is essential to efficiently explore optimization combinations efficiently.

This dissertation aims to enlarge the tuning space by decoupling the three key components in graph processing on a GPU, such as algorithm, schedule, and storage format. This dissertation begins by analyzing the characteristics of the existing optimizations, presenting a fundamental abstraction interface for each key component, and suggesting a new processing model. Furthermore, this research proposes GRAssembler, a new GPU graph processing framework that efficiently integrates the decoupled schedule, storage format, and algorithm without abstraction overhead. Finally, the research enhances the coverage, composability, extendability, and modularity of GPU graph pro-

cessing.

Furthermore, this research demonstrates that decoupled abstractions not only integrate existing studies but also help to uncover new opportunities for performance improvements focusing on workload imbalance in GPU graph processing. First, this research shows that new optimization can be proposed by considering the operation of the abstraction interface. By focusing on the role of storage format and deciding memory access patterns, this research proposes a new storage format called CR^2, aligned to the characteristics of both graphs and GPUs. Second, this research shows that new hardware acceleration opportunities can be revealed by analyzing the abstract methods of existing studies. Based on observations of runtime overhead in existing schedules because of the lack of hardware support, this research proposes a new lightweight GPU functional unit microarchitecture called SparseWeaver that converts sparse operations into dense operations to accelerate schedule. CR^2 and SparseWeaver are designed based on the behavior of the abstraction interface and enhance the tuning space.

Leveraging efficient decoupling and integration, GRAssembler significantly expands the tuning space from 336 to 4,480, resulting in a 1.3 times speedup over the state-of-the-art GPU graph processing framework. CR^2 achieves a 1.53 times performance boost while reducing memory usage by 32.1% on average. SparseWeaver delivers a 2.49 times reduction in execution time compared to the existing approach, with a minimal area overhead of just 0.045%.

Keywords: Graph Processing, GPU, Storage Format, Schedule, Auto-tuning, Workload Balancing

CHAPTER 1

INTRODUCTION

1.1 Graph Processing on a GPU

Graphs are one of the most important and fundamental data structures for reflecting sparse relationships in the real world. Various applications, including social network analysis, web search algorithms, and biological data processing [1, 2, 3, 4, 5], leverage graphs to structure data and perform diverse analytical operations. As real-world graphs grow in size, analyzing graphs demands substantial computational resources and time. Given that graphs comprise numerous vertices and edges, and graph processing typically involves applying the same operation across all vertices and edges, these tasks exhibit a highly parallel nature, making them well-suited for execution on graphics processing units (GPUs). Therefore, many studies have tried to perform graphs on GPUs, for example, and a survey paper was conducted by examining hundreds of papers [1]. Thus, optimizing graph processing on GPUs is critical to minimizing computational time.

To perform graph on GPU, GPU-based graph processing frameworks [6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20] have to consider three core components: algorithm, storage format, and schedule. The algorithm determines how to process the graph, the storage format determines how to store the graph topology in GPU memory, and the schedule determines how to execute

the algorithm. Specifically, storage format influences the memory access patterns, while schedule dictates which threads handle which part of algorithm and in what order.

Therefore, selecting an appropriate storage format and schedule for a given algorithm is crucial, as this selection significantly impacts the efficiency of graph processing on GPUs. Since real-world graphs with their diverse attributes such as size, irregularity, sparsity, skewness, and degree distribution, it is impractical for a single storage format or schedule to ensure optimal performance across all graph types and algorithms. However, existing graph processing frameworks primarily emphasize schedules, often neglecting the critical role of storage formats and thereby restricting the tuning space for graph processing. Their narrow focus on specific topology layouts hinders comprehensive optimization opportunities.

Furthermore, the limited compatibility between schedules and storage formats in existing frameworks [6, 7, 8, 9, 10, 11, 12] restricts tuning options and degrades graph processing efficiency. Current frameworks evaluate only a limited set of predefined schedules, and their storage formats are tightly coupled with the schedules. Consequently, when a schedule is selected, the corresponding storage format is predetermined, even though a different storage format might be more suitable for certain datasets. This rigid coupling between schedules and storage formats limits optimization flexibility and performance potential.

To make things worse, the tightly coupled schedules and storage formats severely limit the extendability of graph processing space. Ideally, developers should be able to explore all $M \times N$ combinations by independently im-

plementing M schedules and N storage formats (requiring only $M + N$ implementations). However, the coupling necessitates $M \times N$ implementations, imposing a substantial development burden. For example, introducing a new storage format requires implementing it separately for each of the M existing schedules. This development overhead discourages exploration and limits the expansion of GPU graph processing optimization spaces.

In addition, the tightly coupled schedule and storage format also limit the modularity of each component, making it difficult to explore new optimization opportunities for both schedule and storage format. The precise definition and role of each component make it easier to understand individual components and would help to find missing optimization opportunities. The decoupled storage format and schedule can conduct performance improvement research by focusing solely on specific operations without being constrained by other components. Alternatively, other components can be easily reused, allowing us to obtain new optimization opportunities.

In particular, increasing modularity can open up new opportunities to address workload imbalance, which is one of the most critical problems of graph processing on GPUs with a SIMT architecture. When traversing the neighboring edges of certain vertices in a graph, traversal is inherently based on a single vertex as the reference point, leading to an intuitive approach of mapping one edge to a single thread. However, one of the important graph types, such as web graphs or heavy-tailed graphs, exhibits high irregularity and skewness, where a small number of vertices are connected to a disproportionately large number of edges. For highly skewed graphs, the intuitive mapping often causes workload imbalance among inter- or intra-warp, causing idle threads

in warps. The key challenge lies in effectively distributing neighboring edges across threads. This issue can be addressed either by proposing a new storage format, which is a static and predefined approach, or through schedule, a runtime-based approach. Therefore, decoupling the two not only clearly defines their respective roles but also facilitates combinations with other optimizations, thereby unlocking opportunities for improved performance.

Therefore, it is essential to decouple schedule and storage format, enabling greater tuning coverage, composability, extendability, and modularity.

1.2 Contribution

This research has focused on decoupling storage format and schedule in graph processing on GPU and enlarging the tuning space. This decoupling enhances the coverage, composability, extendability, and modularity of Graph processing on GPU. In addition, this research also explores new optimization and acceleration opportunities based on deeply exploring storage format and schedule.

First, the main contribution of this research is proposing a new graph processing abstraction scheme that fully decouples schedules and storage formats. This work begins by characterizing schedule and storage format based on their data access order and the explicit representation of edge data. Leveraging this characterization, the proposed abstraction interfaces effectively separate schedules, storage formats, and algorithms, facilitated by an abstract processing model that integrates these interfaces seamlessly. In addition, this research develops a prototype, GRAssembler, which implements the proposed

graph processing abstraction interfaces with various optimizations and enhances the graph processing tuning space through the introduction of a new processing model. Lastly, a case study on extending the storage format library is presented, illustrating the high adaptability and extensibility enabled by this abstraction.

Second, based on analysis of existing storage format, this research suggests a new graph storage format called CR^2 focusing on memory access patterns considering GPU and real-world graph characteristics. CR^2 provides vertex ID compression with community-aware subgraphs and vertex degree regularization with degree-ordered subgraphs, enabling high-performance and memory-efficient processing. CR^2 reduces memory usage while exploiting the skewed locality of a graph and degree-ordered subgraphs that allow fine-grained workload balancing and the removal of offset arrays. Additionally, the work presents an in-depth evaluation of CR^2-based graph processing.

Third, this research proposes a new lightweight GPU functional unit microarchitecture called SparseWeaver based on an analysis of common functionality in existing software schedule schemes. SparseWeaver converts sparse operations in graph processing into dense operations using Weaver and balances the workloads across GPU threads. Weaver tightly integrated into the GPU with minimal hardware modifications. This research provides an in-depth evaluation of SparseWeaver, comparing it with both software and hardware schemes, using real-world graph datasets and benchmarks, demonstrating its effectiveness with minimal hardware modifications.

1.3 Organization

This dissertation is organized as follows:

Chapter 2 introduces a novel graph processing abstraction scheme for GPU that fully decouples schedules, storage formats, and algorithms using abstraction interfaces. Additionally, this chapter presents a prototype framework, GRAssembler, based on a new processing model which efficiently integrates the decoupled schedule, storage format, and algorithm components while minimizing abstraction-related overhead.

Chapter 3 proposes a new storage format optimization, referred to as CR^2. This scheme introduces vertex ID compression through community-aware subgraphs and vertex degree regularization using degree-ordered subgraphs, enabling high-performance and memory-efficient graph processing on GPUs.

Chapter 4 introduces SparseWeaver, a new schedule that utilizes a novel microarchitecture extension. This microarchitecture accelerates the schedule process by transforming sparse operations into dense ones through the use of storage format abstractions, ensuring balanced workloads across GPU threads.

Chapter 5 concludes the dissertation, summarizing the contributions and discussing future research endeavors that relate to this work.

CHAPTER 2

GRAPH PROCESSING MODEL AND ABSTRACTION FOR GRAPH PROCESSING ON A GPU

To enlarge the tuning space of graph processing, this research introduces **a novel graph processing abstraction scheme that completely decouples schedules and storage formats**. Through the development of extensible and non-interfering interfaces, we identify that schedules and storage formats can be effectively decoupled. A detailed analysis of the three primary components of graph processing reveals the following insights. First, schedules can be categorized into two types based on their graph data access. Second, storage formats can be abstracted into two forms depending on whether they explicitly define both source and destination vertices. Third, components of the same type share a small set of commonly used operations, enabling the creation of abstract interfaces capable of implementing all existing schedules and storage formats. These interfaces fully decouple schedules and storage formats, facilitating significant expansion of the tuning space.

This chapter introduces GRAssembler, a new GPU graph processing framework designed to support fully decoupled abstraction interfaces, thereby enhancing the enlargement of tuning spaces. GRAssembler comprises a tuner, a graph builder, and a runtime, which respectively enable the exploration of tuning spaces, the construction of various storage formats, and the execution

of graph programs using abstraction interfaces. To mitigate potential performance overheads introduced by the abstraction, GRAssembler minimizes argument passing and employs function templates instead of function pointers. By exploring the expanded tuning space, GRAssembler identifies the optimal combinations of storage format and schedule, surpassing the capabilities of existing frameworks with limited tuning spaces. Furthermore, GRAssembler broadens the tuning space with a new GPU-friendly optimization. Leveraging its abstract interfaces and optimizations, GRAssembler discovers the optimal graph processing model from an expanded tuning space that fully encompasses the entire tuning spaces of existing frameworks.

2.1 Component of Graph Processing on GPU

GPU graph processing frameworks [7, 8, 9, 10, 11, 12, 15, 16, 17] are designed to process various graphs using diverse algorithms with some tuning options. Therefore, these frameworks receive input graphs and simple algorithm functions from the user and process them on GPUs. The input graph normally consists of vertices and edges and sometimes includes weights for each edge. algorithm refers to the processing method applied to the graph, so the framework performs the algorithm function during pre-defined kernels. The tuning options modify the pre-defined kernels. This work categorizes the tuning options into **schedule** and **storage format**. Schedule describes the execution order of graph components, for example, how to distribute vertices to threads to process algorithm. Storage format describes how to store the graph topology in memory.

2.1.1 Graph Algorithm for GPU

An algorithm defines the approach for processing a graph $G = (V, E)$. Typically, graph processing involves the iterative execution of four core operations: $gather$, sum, $apply$, and $scatter$. $Gather$ retrieves data from the neighboring edges of an active vertex. Sum performs a reduction (e.g., fetch-and-add or atomic min/max) on the data gathered during the $gather$ step. Following this, $Apply$ updates the vertex value using the results from the $gather$ and sum operations. Specifically, $apply$ acts as a self-update function to generate the final value for the iteration. $Scatter$ identifies updated vertices and incorporates them into the new active vertex set [21]. Typically, single-GPU graph frameworks can scatter through memory without requiring a separate $scatter$ process. The four core operations iterate until an end condition, such as convergence or the absence of further updates, is met.

2.1.2 Graph Schedule for GPU

The schedule determines how GPU threads are assigned to specific parts of a graph and the sequence in which they process the data. Two fundamental scheduling schemes exist: the Vertex Mapping (VM) scheme assigns each vertex to a separate thread, while the Edge Mapping (EM) scheme assigns each edge to a thread.

The inherent skewed properties of real-world graphs often lead to significant workload imbalance on SIMT architecture [21, 22]. For instance, because vertices can have varying numbers of edges, VM suffers from workload imbalance, as illustrated in Figure 2.1. On the other hand, EM does not have work-

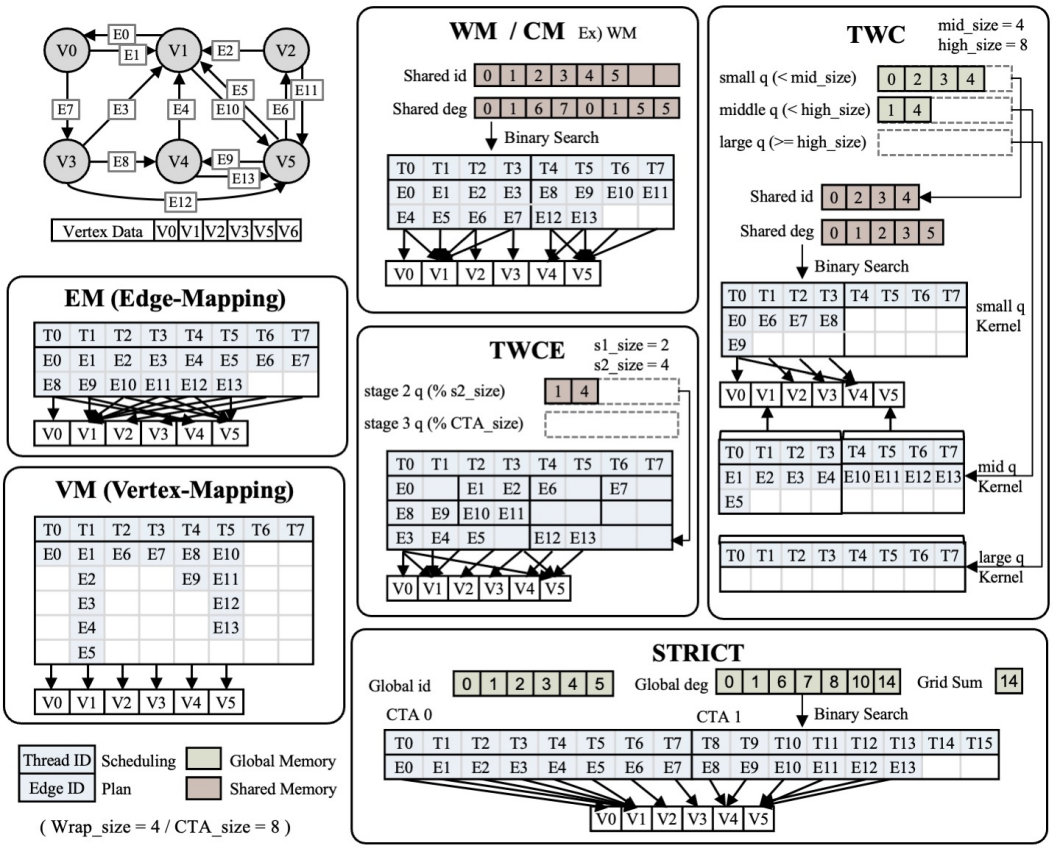


Figure 2.1: Existing schedules with the simple example graph. VM and EM are basic schedules based on graph characteristics, while five other schedules enhance workload balancing through additional computation or memory usage.

load imbalance, but it needs more memory access in $gather$ stage, as depicted in Figure 2.1.

To address these issues, a variety of schedules illustrated in Figure 2.1 employ different strategies to map workloads onto GPU resources, such as shared or global memory [7, 8, 13, 14]. While these schedules introduce additional communication and memory usage, they often achieve performance gains with certain graphs.

Cooperative Thread Array Mapping (CM) and Warp Mapping (WM) distribute vertices across threads within a warp or a Cooperative Thread Array (CTA) using shared memory. WM and CM load the degree of each vertex and store basic information in shared memory. Subsequently, WM and CM assign the neighboring edges of those vertices to individual threads. By sharing workloads at the warp or CTA level, WM and CM balance workloads effectively while reducing synchronization overhead.

The Thread, Warp, and CTA method (TWC) [13] adopts a different strategy by classifying vertices based on their degrees (i.e., the number of edges). Vertices are divided into low, middle, and high-degree buffers stored in global memory. TWC then launches three separate kernels, each optimized for thread, warp, or CTA granularities, to process these queues. Although TWC achieves a better workload balance than WM and CM due to its global workload distribution, it incurs greater overhead than CM and WM from additional kernel launches and frequent global memory accesses.

A variation of TWC, TWC based on Edge (TWCE)[8], constructs and processes the queues entirely within shared memory in a single kernel, reducing the overhead associated with multiple kernel launches. Additionally, this

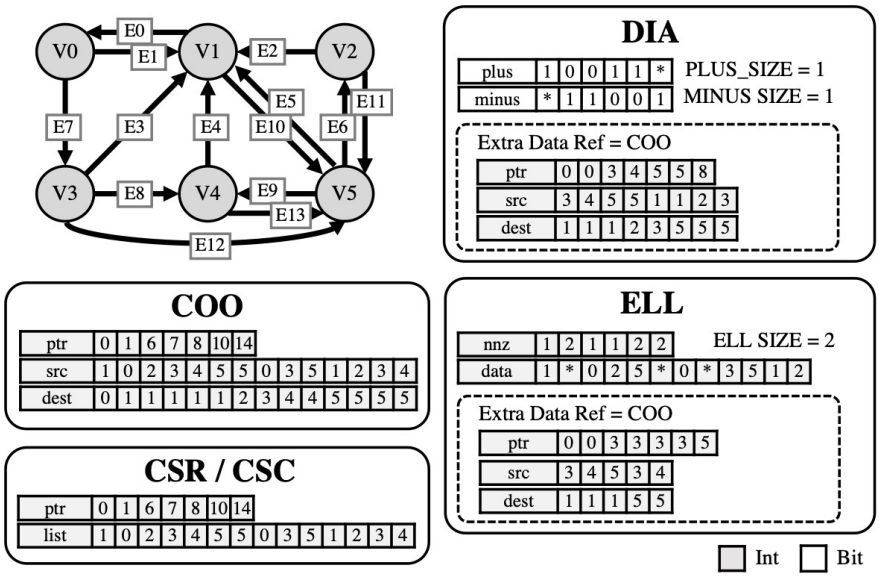


Figure 2.2: Existing storage formats with the simple example graph. COO is a basic form that includes both source and destination information in memory, while other layouts aim to reduce memory usage or balance the workload by modifying the stored information.

process attempts to replace shared memory loads with atomic updates. Another approach, STRICT[14], reorganizes active edges in global memory for the current iteration and distributes them efficiently across CTAs for parallel processing.

2.1.3 Storage Format for GPU

The storage format defines how a graph's topology is organized in GPU memory. Because the edges connected to a single vertex represent only a small portion of the total edges in a graph, graph topology is inherently sparse data. Consequently, a key challenge in storage format design is to compress this sparse topology while minimizing irregular memory accesses [12, 16, 17, 19,

23, 24, 25, 26]. Numerous storage formats have been developed to tackle this issue, as shown in Figure 2.2.

The Coordinate Storage (COO) format uses two arrays to represent edges, storing source vertex IDs (src) and destination vertex IDs ($dest$), along with an offset array (ptr) that indicates the starting indices for each vertex in the src and $dest$ arrays. For example, in Figure 2.2, $ptr[1] = 1$ and $ptr[2] = 6$ signify that the edges connected to vertex $v1$ are located from ($src[1]$, $dst[1]$) to ($src[5]$, $dst[5]$).

Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats eliminate one of the edge arrays (e.g., $dest$ in Figure 2.2) found in the COO format by leveraging the offset array (ptr) to deduce the missing information. For instance, in Figure 2.2, with $ptr[2] = 6$, the source and destination IDs of the edge at $list[6]$ are 5 (the value at $list[6]$) and 2 (the index of ptr), respectively. By omitting one edge array, CSR and CSC formats reduce memory usage compared to COO; however, reconstructing the omitted data from an edge ID involves an expensive binary search.

The Diagonal (DIA) format [24] stores subdiagonal edges in structured arrays such as $plus$ and $minus$, while non-subdiagonal edges are maintained in a separate format like COO. For example, in Figure 2.2, for vertex $V4$, $plus[4] = 1$ represents an upper subdiagonal edge pointing to 5, and $minus[2] = 1$ corresponds to a lower subdiagonal edge pointing to 1.

The Ellpack (ELL) format [24] reserves a fixed number (ELL_SIZE) of edges per vertex in a structured array ($data$), with any extra edges stored in a secondary format such as COO. For instance, in Figure 2.2, the edges adjacent to vertex $V4$ are stored from $data[ELL\_SIZE \times 4]$ to $data[ELL\_SIZE \times 5 - 1]$,

assuming a fixed number of edges per vertex. The exact edge count can be derived from $nnz[4] = 2$.

2.2 Related Work and Motivation of Decoupling

Existing graph processing frameworks [7, 8, 9, 10, 11, 12, 15, 16, 17] attempt to determine the best tuning options for a given algorithm by exploring their available configurations, including schedule and storage format. However, these frameworks would fail to find the optimal tuning options because of their constrained coverage of tuning options, limited composability, and lack of extendability.

Table 2.1 shows the detail of existing configurable graph processing frameworks, such as Gunrock [6], Gswitch [7], and GraphIt [8]. Gunrock [6] extends this by introducing active set deduplication tuning and additional scheduling plans, such as VM, EM, and TWC, to enhance load balancing. Gswitch [7] supports more schedules than Gunrock and defines five configurations such as direction, active-set data structure, load balancing (scheduling plans like GRAssembler), stepping, and kernel fusion—while emphasizing the generality and significance of graph programs. Similarly, GraphIt [8] supports the same schedules and introduces a vertex blocking tuning knob and the TWCE scheduling plan to refine optimization options further. These frameworks also support auto-tuning for their respective configuration options, enabling more efficient graph processing.

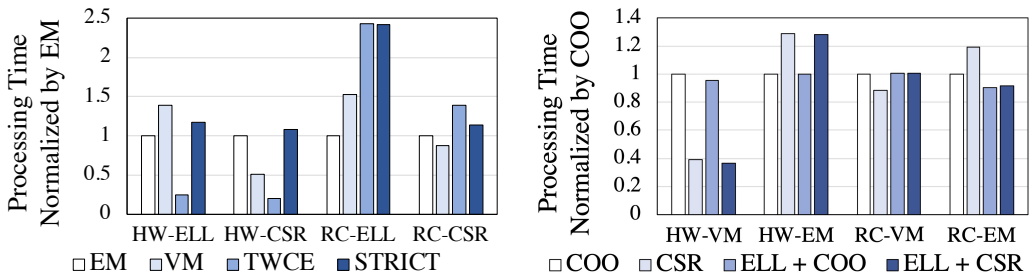
Despite these advances, existing frameworks predominantly focus on scheduling plans (schedules) while overlooking the critical role of topology layouts

Table 2.1: Comparsion of existing GPU-based graph processing framework reveals their varying levels of coverage across schedules, storage formats, and interfaces

Framework	Schedule	Storage Format	Interface		
			Algorithm	Schedule	Storage Format
Gunrock [6]	VM, EM, TWC	COO, CSR	O	X	X
Gswitch [7]	VM, EM, WM, CM, TWC, TWCE, STRICT	COO, CSR	O	X	X
Graphit [8]	VM, EM, WM, CM, TWC, TWCE, STRICT	CSR	O	X	X
GRAssembler	VM, EM, WM, CM, TWC, TWCE, STRICT	COO, CSR, ELL, DIA, Gshard	O	O	O

(storage formats). For instance, while frameworks like Gswitch and Gunrock support two common layouts, COO and CSR, GraphIt [8] supports up to seven schedule optimizations but only a single layout, CSR. This limited coverage of storage formats significantly restricts their tuning spaces and prevents the frameworks from fully optimizing graph processing efficiency. In contrast, GRAssembler extends the tuning space by incorporating storage formats into its configuration options, thus addressing this limitation and enabling more comprehensive optimization for graph algorithms on GPUs.

Problem 1: Limited exploration coverage. Previous graph processing frameworks fail to include a diverse range of schedules and storage formats in their tuning space. As illustrated in Table 2.1, each framework considers only a subset of schedules and storage formats within its exploration scope. While these frameworks extend their support for various schedules, they often overlook certain storage format options, such as CSR and COO, or fail to explore more advanced storage formats proposed in recent studies [17, 19].



(a) schedule (b) storage format

Figure 2.3: Comparison of processing times for various schedules and storage formats using the PageRank algorithm on the hollywood and road-central datasets [27].

Since storage format plays a crucial role in determining graph processing performance, as shown in Figure 2.3b, this limited exploration — particularly regarding storage format — prevents frameworks from capitalizing on further optimization opportunities.

Problem 2: Limited composability. As shown in Figure 2.3, the processing time varies significantly based on the combinations of schedules and storage formats across two datasets [27]. The optimal schedule depends on both the storage format and the dataset, while the best storage format similarly varies based on the schedule and dataset.

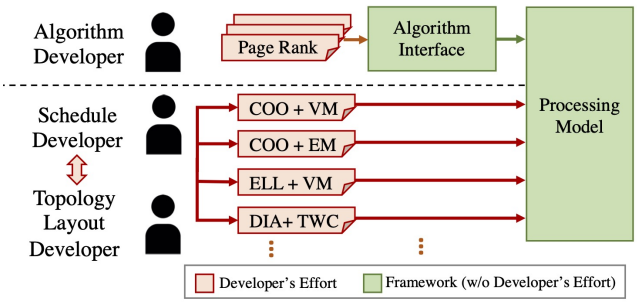
However, existing frameworks decouple only the algorithm from the processing model, while schedule and storage format remain tightly coupled within the model, as illustrated in Figure 2.4a. This tight coupling between schedule and storage format allows exploration only of pre-implemented combinations, potentially missing the optimal combination if it is not supported.

Problem 3: Limited exploration extendability. The tight coupling of schedule and storage format in existing frameworks also hampers their extendabil-

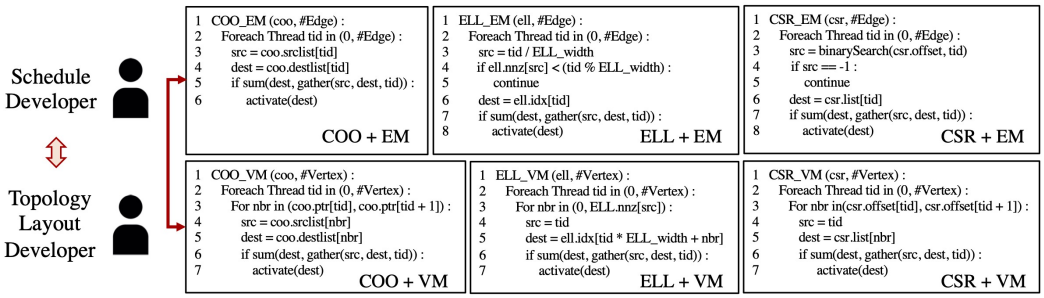
ity. When schedule and storage format are interdependent, a framework with M schedules and N storage formats requires the development of $M \times N$ combinations to support all possible configurations. For example, to fully implement both VM and EM schedules with COO, CSR, and ELL storage formats, the framework must provide six combinations (2×3), as shown in Figure 2.4a. Furthermore, adding a new schedule or storage format requires the developer to implement N or M additional combinations, respectively, to maintain full compatibility. Consequently, introducing new options in these frameworks involves significant development effort and cost.

Solution: A new graph processing abstraction model. To address these challenges, a new abstraction model and a new GPU graph processing model are needed, one that decouples not just the algorithm but also the schedule and storage format, as shown in Figure 2.6. This work proposes a new abstraction model designed with the objectives of high coverage, composability, extendability, and efficiency and processing model using that abstraction model.

First, the abstraction model should be versatile enough to encompass a broad range of schedules and storage formats (**high coverage**). Second, the scheduling of graph vertices and edges using a schedule, accessing storage format through a storage format, and processing algorithm functions should be fully independent, allowing for seamless combination within the abstraction model. This independence ensures that all possible combinations can be realized (**high composability**) and that new options can be added without requiring changes to existing implementations (**high extendability**). For instance, if there are M schedules and N storage formats, developers only need



(a) Development model of the existing frameworks



(b) Example codes for different schedules and storage formats

Figure 2.4: The development model and example codes used in existing frameworks [6, 7, 8] illustrate their limitations. As shown in (a), the development model of these frameworks decouples only the algorithms from the processing model. In (b), their example codes demonstrate how schedules (e.g., VM, EM) and storage formats (e.g., COO, CSR, ELL) are implemented. Due to the tight coupling between schedule and storage format within these frameworks, developers are required to implement $M \times N$ combinations to support M schedules and N storage formats.

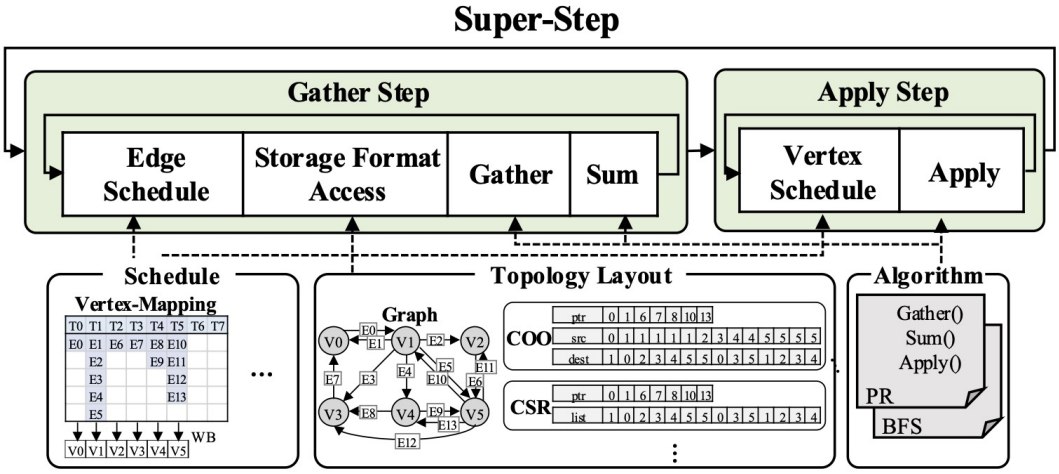


Figure 2.5: Graph processing model suggested in GRAssembler. The processing model includes edge schedule, storage format access, and vertex schedule, separately.

to implement $M + N$ options to enable all $M \times N$ combinations. As illustrated in Figure 2.6, developers would need to create only 5 $(2 + 3)$ options to fully support VM and EM schedules along with COO, CSR, and ELL storage formats.

Lastly, the decoupled schedule, storage format, and algorithms should be integrated efficiently, minimizing abstraction overhead (**high efficiency**). This design allows users to achieve optimal performance by selecting the best combination of schedule and storage format.

2.3 Graph Processing Abstraction

We introduce a new graph processing model. As shown in Figure 2.5, our processing model consists of a super-step that executes algorithm iteratively. The super-step consists primarily of two stages: the *gather step* and the *apply step*. During the gather step, the framework collects data from the incoming neigh-

bors of each vertex. In the apply step, it updates vertex data by incorporating the gathered data and the apply method of algorithm. After completing each super-step, the framework identifies a set of vertices called the *frontier*, which will be processed in the subsequent super-step.

The gather step is further divided into four sub-steps: *edge schedule*, *storage format access*, *gather*, and *sum*. In the edge schedule step, the framework determines the edge traverse direction, such as the incoming edge order or outgoing edge order of the frontier vertices. schedule method investigates neighbor information by decided direction and assigns the scheduled edges to threads. During the storage format access sub-step, it retrieves edge details such as source and destination vertex IDs and their weights from the storage format. The gather and sum sub-steps involve collecting and aggregating the incoming data for each vertex based on the requirements of algorithm. Since some large graphs have more edges, the total number of software-supported threads (e.g., maximum block size x maximum thread size) and the processing model iterate gather step till traversing all active edges are traversed.

After finishing the gather step, the apply step that consists of *vertex schedule* and apply is executed. To schedule vertex, the framework assigns vertices from the frontier to threads. The apply step then updates the data for each vertex by reflecting the accumulated incoming data. Same as the gather step, the apply step iterates till traversing all vertex in the graph.

Unlike existing graph processing models [7, 8, 10, 11, 12, 15, 16, 17], the newly proposed abstract graph processing model treats edge scheduling, vertex scheduling, and storage format access as distinct sub-steps. This separation allows a graph processing framework to independently perform edge

and vertex scheduling, access the storage format, and execute the algorithm. Consequently, with well-designed abstraction interfaces for schedule, storage format, and algorithm, this model enables a complete decoupling of schedule, storage format, and algorithm from one another.

2.3.1 Schedule Abstraction

The schedule determines which edge and vertex each GPU thread processes during the edge and vertex scheduling sub-steps. Since the gather step handles all the imbalanced tasks by processing incoming edges of each vertex, the apply sub-step becomes well-balanced, as it operates independently of other vertices or edges. Consequently, vertex scheduling involves a straightforward mapping of vertex IDs to GPU threads. In contrast, graph processing schedules [7, 8, 13, 14] focus on balancing the scheduling of imbalanced edges. This schedule abstraction also emphasizes defining a unified interface for all schedules discussed in Chapter 2.1.2, enabling the next edge to be scheduled after finishing the current edge scheduling sub-step.

To develop a generic schedule interface, this work analyzes the schedules described in Chapter 2.1.2 and designs the schedule interface as presented in Table 2.2. Since these schedules aim to balance workloads across GPU threads during edge scheduling, the abstraction is divided into two core interfaces: *edge scheduling* and *load balancing*. To further enhance coverage, extendability, and efficiency, the proposed interfaces include additional arguments for handling the *frontier* and *direction*.

Abstraction for edge scheduling: All schedules require scheduling the next edge for each GPU thread at the start of the gather step. Thus, the schedule in-

terface should include a method such as `getNextEdgeID` that takes the thread ID as an argument and returns the next edge ID for the gather step. Additionally, `getNextEdgeID` includes a variable ($state$) that indicates the scheduling state, enabling the framework to determine whether scheduling is complete or if an iteration has been skipped.

Abstraction for load balancing: To mitigate the synchronization overhead of EM and the imbalance issues in VM, complicate schedules such as WM or TWCE shared graph topology using global or shared memory. When starting each iteration, these schedules collect neighboring edges of vertices at the warp, CTA, or kernel level, depending on the schedule, and map these edges evenly. Since the neighboring edges are stored in memory, the abstraction provides distribution interfaces like `initGlobal` and `initShared`, along with arguments ($shared\_buffer$ and $edge\_list$) that pass distribution results to the `getNextEdgeID` method. For instance, TWC can use `initGlobal` to generate global queues for vertices with low, medium, and high degrees in global memory, while STRICT uses it to gather edges of frontier vertices and distribute them across CTAs. Similarly, `initShared` allows WM, CM, and TWCE to create shared IDs and degrees (as shown in Figure 2.1) and TWCE to form queues for small, medium, and high-degree vertices in shared memory. VM and EM can also configure edge ID ranges using `initShared` and store the results in $edge\_list$.

Frontier argument: To perform graph processing efficiently, frameworks analyze the *frontier*, a subset of vertices to process during each super-step, instead of processing all vertices. To support this, the schedule abstraction introduces an $frontier$ argument to the `initGlobal` and `initShared` methods. Fur-

Table 2.2: GRAssembler schedule interface specifications.

Method	Method Description
Schedule Interface	
void **`initGlobal`** (Frontier& frontier, bool isIn)	Initialize frontier in global memory for schedule ∗ frontier: frontier of the current iteration, isIn: direction of edge traverse (PUSH/PULL)
void **`initShared`** (int tid, Frontier& frontier, bool isIn, int* shared_buffer, int* edge_list)	Initialize frontier in shared memory for sche ∗ shared_buffer: pointer of predefined shared memory for schedule, edge_list: set of edges to execute in the current block
state **`getNextEdgeID`** (int tid, bool isIn, int* shared_buffer, int* edge_list, int& vid, int& eid)	Schedule neighboring edges for current thread, returning edge id

thermore, various frontier structures—such as queues, bitmaps, bytemaps, and counters—are provided to help the framework choose the most suitable representation. The $frontier$ argument interface allows seamless manipulation of frontiers regardless of their underlying data structure.

Direction argument: When performing VM, we can determine the traversal direction based on what is considered a neighbor: incoming edges (PULL) or outgoing edges (PUSH). Even though EM is not affected by the traversal direction, we include direction information as an argument for the method to accommodate VM and other more complex scheduling.

2.3.2 Storage Format Abstraction

To suggest storage format interface, this work begins by analyzing the storage formats detailed in Chapter 2.1.3 and proposes the storage format interface

Table 2.3: GRAssembler storage format interface specifications.

Method	Method Description
Storage format Interface	
void **getNeighbor** (int vid, bool isIn, int& begin, int& end)	Return the start and end positions of edge_list of target vertex vid. Note that storage format interface also receives direction information.
void **getEdge** (int eid, int vid, bool isIn, int& src, int& dest, WT& weight)	Return the source and destination vertex IDs along with the edge weight for a given edge ID (eid)
bool **searchVID** (int eid, bool isIn, int& vid)	Obtain a vertex ID (vid) corresponding to a specific edge ID (eid). This method returns the destination vertex ID for the Pull direction and the source vertex ID for the Push direction.
void **initTopology** (bool isIn, Frontier &old, Frontier &new)	Initialize virtual storage format for each super-step
TT **topologyGather** (int src_id, int dest_id, WT weight)	Return pre-gathered data stored in the virtual storage format.

Table 2.4: GRAssembler algorithm interface specifications.

Method	Method Description
Algorithm Interface	
void **initVertexValue** (int vid)	Set the initial vertex value
TT **gather** (int src_id, int dest_id, WT weight)	Gather information from an edge (src_id, dest_id) for the destination vertex
bool **sum** (int dest_id, TT data)	Integrate gathered data for a destination vertex (dest_id)
bool **apply** (int vid)	Update destination vertex value using the collected data from the gather method
bool **filter** (int vid)	Exclude a vertex (vid) from the frontier filter method. The filtered result returns TRUE.
bool **checkDirection** (int numOfVertices, int numOfFrontier)	Change the direction of the next iteration with base frontier information
WT **getNewGlobalThreshold** (WT oldThreshold)	Update the next weight threshold

described in Table 2.3. In this context, WT and TT represent types for edge weight, and temporary data exchanged between the gather and sum steps. FrontierType is a tunable data structure in GRAssembler for frontiers, which can take forms such as a queue, bitmap, bytemap, or counter. Given the sparsity of graph topology, various compression schemes are commonly applied to storage formats. To effectively support these schemes while enabling both fundamental storage format access and advanced storage format access, this work introduces storage format interfaces based on three abstractions: **storage format data access**, **fast data access**, and **virtual topology access**.

Abstraction for storage format data access: Since the purpose of a storage format is to represent graph topology, the primary operation of the storage format interface involves returning the incoming and outgoing edges for a vertex and identifying the source and destination vertices for a given edge. To achieve this, the interface defines two core methods: `getNeighbor` and `getEdge`. The `getNeighbor` method retrieves the edge list for a vertex ID (vid) as a range of indices ($begin$ and end) in the edge array. To specify whether incoming or outgoing edges are required, `getNeighbor` takes a ($isIn$) to indicate direction. The `getEdge` method provides details about an edge($srcvertexid$, $destvertexid$, $weight$), for a given edge ID (eid).

Abstraction for fast data access: Compression schemes such as CSR or Tigr [19] store only one vertex ID for each edge (source or destination) to conserve memory. To reconstruct the complete edge information, frameworks typically perform a computationally expensive binary search on the ptr array. However, since most schedules (excluding EM in Chapter 2.1.2) generate edge lists through `getNeighbor` for a base vertex, the framework already has

access to the missing information during the super-step. As a result, the base vertex ID can be used directly, eliminating the need for a binary search.

To prevent unnecessary additional computations, this work separates the `getEdge` functionality into two methods: `searchVID` and `getEdge`. The `searchVID` method performs a search operation on the ptr array for neighboring edges to determine the base vertex ID (vid) for a given edge ID (eid). The `getEdge` method then uses the base vertex ID (vid) and direction ($isIn$) as inputs. If the base vertex ID is known within the super-step, the `searchVID` step can be skipped with the help of compiler optimizations.

Abstraction for virtual topology access: To enable diverse access patterns, such as non-contiguous edge list access, this abstraction introduces virtual topology access through `initTopology` and `topologyGather`. The `initTopology` method initializes the virtual topology for each super-step and creates a new frontier, while `topologyGather` wraps the `gather` method in the algorithm interface to operate on the virtual topology.

For example, in a blocked layout containing multiple sub-graphs, a single real vertex may be represented by multiple virtual vertices distributed across the sub-graphs. Using `initTopology`, a storage format developer can generate these virtual vertices and their associated edge lists, enabling sequential access to edges across sub-graphs, even if the edges are not sequentially stored in the real topology.

`initTopology` and `topologyGather` allow the framework to process continuous edge lists with temporary data, as seen in Cusha [17]. After generating a virtual storage format for the temporary data via `initTopology`, the framework can gather neighboring edge data by continuously accessing the virtual

edge list using `topologyGather`.

2.3.3 Algorithm Abstraction

A graph algorithm can consists of four different operations: $gather$, sum, $apply$, and $scatter$, as described in Chapter 2.1.1. This work introduces algorithm interfaces, including `gather`, `sum`, and `apply`, excepting $scatter$. The `gather` function gathers information for the destination vertex ($dest_id$) generated using an edge (src_id, $dest\_id$, $weight$). The `sum` function aggregates the gathered results ($data$). Template types are used for edge weight (WT) and gathered information (TT) to allow flexibility in defining the types for edge weights and gathered data. The `apply` function updates the value of a vertex identified by its ID (vid) after finishing gathering all the information from neighboring edges. To inform the graph processing framework of updated vertices, both `sum` and `apply` return a boolean value. When a returning value is true, the framework invokes the `filter` method on the vertex and adds the vertex to the frontier based on filtered results. Additionally, this work provides `initVertexValue` to initialize vertex values at the start of the graph processing.

Removing redundant operations in $scatter$: The $scatter$ operation involves activation, filtering, and fan-out edge processing. By leveraging the return values of the $gather$ and $apply$ functions, the activation operation is embedded into the graph processing framework, eliminating redundant activation tasks in $gather$, $apply$, and $scatter$. Furthermore, since the fan-out edge processing in the current iteration has the same semantics as the gathering operation in the next iteration, this work modifies the $gather$ function to handle fan-out for updated values in the next super-step. As a result, only the filter function is

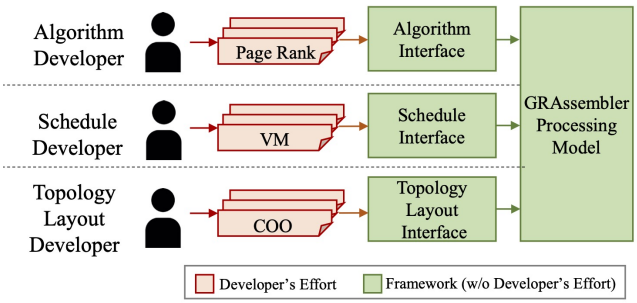
needed.

Abstraction for flexible execution: Certain algorithms, such as BFS and Delta-SSSP, dynamically adjust their processing direction between pull and push or update vertex values only if they exceed a dynamically changing threshold. To support such flexibility, this work introduces two methods: `checkDirection` and `getNewGlobalThreshold`. The `checkDirection` method allows an algorithm to decide the processing direction dynamically based on the frontier size during each super-step. The `getNewGlobalThre-shold` method enables an algorithm to update the threshold value from its previous value.

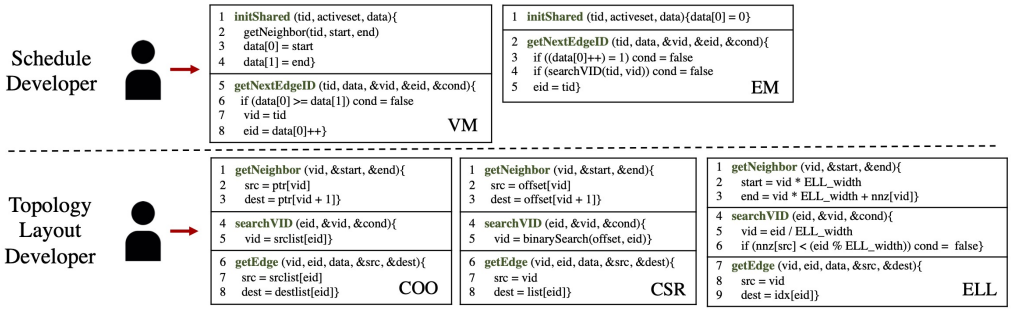
2.3.4 Assembling Abstract Interfaces

The newly proposed abstract graph processing model integrates the schedule, storage format, and algorithm interfaces to process a graph. As shown in Figure 2.6a, this approach enables developers to independently implement their schedule, storage format, and algorithm using the respective interfaces outlined in Table 2.2, Table 2.3, and Table 2.4. Using these interfaces, the abstract graph processing model described in Algorithm 2.1 performs a super-step iteration on a given graph with frontier. The model comprises two main phases: the gather step (Line 2 to 25) and the apply step (Line 26 to 30).

To handle storage formats like DIA and ELL that require additional layout-specific processing, the model iterates across multiple storage formats (Line 2). The edge processing begins by initializing the storage format and global memory (Line 3 to 4), followed by the launch of GPU kernels. During kernel execution, GPU threads collaboratively initialize shared memory once per kernel invocation (Line 7).



(a) Development model of this work



(b) Example codes for different schedules and storage formats

Figure 2.6: Development model and its example codes used in the GRAssembler framework proposed in this work. (a) illustrates the newly proposed development model, which fully decouples the schedule, storage format, and algorithm from the processing model. (b) provides example codes for schedules (VM, EM) and storage formats (COO, CSR, ELL) used in Figure 2.4. With the complete decoupling of schedule and storage format, developers only need to implement M + N options to support all M x N combinations of M schedules and N storage formats.

Once global and shared memory is initialized, the gather step iterates over a while loop to execute sub-steps: edge schedule, storage format access, gather, and apply. Through interaction with schedule, the model retrieves an edge ID (Line 10) and determines whether to terminate or skip the current iteration. It then interacts with storage format to fetch the source and destination vertex IDs and the weight of the edge (Line 16). If the source vertex is inactive, the model skips the iteration (Line 17). The algorithm interface is then used to perform `gather` and `sum` operations (Line 20). If `sum` updates the destination vertex value, the thread adds the destination vertex to the output frontier (Line 21).

After completing the gather step, the model proceeds to the apply step. A GPU kernel is launched to execute the `apply` operation, and if a vertex is updated, it is added to the output frontier (Line 26 to 30).

2.4 GRAssembler Framework

Figure 2.7 presents our framework, GRAssembler, which is composed of three primary components: a graph builder, a tuner, and a runtime. To identify the optimal combination, the tuner iteratively evaluates various tuning options using the runtime. The graph builder transforms raw input graph data into the selected storage format format. The runtime executes graph processing tasks by interacting with the GRAssembler library, where schedule, storage format, and algorithm implementations are developed.

GRAssembler tuner systematically searches for the best tuning options, including schedules and storage formats, for a specific algorithm and graph

Algorithm 2.1: Processing model of GRAssembler

```
Input: Sche : Schedule
       Layouts : Storage formats
       Alg : Algorithm
       ASet_in : Input frontier
Output: ASet_out : Output frontier
Function Super-Step (Sche, Alg, Layouts, ASet_in, ASet_out) :
    foreach Layout ∈ Layouts do
        Layout.initTopology (...)
        Sche.initGlobal (...)
        foreach kernel ∈ Sche.KernelSet do
            foreach thread ∈ kernel do
                Sche.initShared (...)
                while true do
                    // Edge Schedule Step
                    (cond, eid, vid) ← Sche.getNextEdgeID (thread, ...)
                    if cond = terminate then
                        break
                    else if cond = skip then
                        continue
                    // Storage format Access Step
                    (src, dest, weight) ← Layout.getEdge (eid, vid ...)
                    if Aset_in.check(src) == false then
                        continue
                    data ← Alg.gather (src, dest, weight)
                    if Alg.sum (dest, data) then
                        ASet_out.add(dest)
                end
            end
        end
    end
    foreach thread ∈ vertexKernel do
        vid ← thread
        if Alg.apply (vid) then
            ASet_out.add(vid)
    end
end
```

dataset. The tuner consists of a *manager*, *storage format selector*, *scheduling selector*, and *compiler*. The *manager* identifies the tuning options that can be adjusted for a given algorithm. For instance, the BFS algorithm dynamically modifies its processing direction (push or pull) based on the number of active vertices by using `checkDirection`. The manager recognizes the use of `checkDirection` in the algorithm, creates two separate sets of tuning options by excluding the direction option from the tunable set, and then explores the tuning options for each direction independently. Within the tunable options, the *storage format selector* focuses on choosing storage format-related configurations, while the *scheduling selector* selects options related to schedules, including the frontier type and the graph processing direction. Finally, the *compiler* generates and optimizes the graph processing program using the chosen tuning options and the given algorithm.

GRAssembler graph builder creates a storage format from raw graph input based on the specified storage format option. During this process, the *GRAssembler graph builder* divides the graph into subgraphs and merges them into a blocked graph to enhance data locality. As this process can often take longer than kernel execution, the resulting layout is saved as separate files to be reused across iterations.

GRAssembler runtime runs the assembled graph processing program following the super-step procedure described in Algorithm 2.1. The runtime initializes vertex values and the frontier data structure, executes the super-step algorithm defined in Algorithm 2.1, and invokes `checkDirection` and `getNewGlobalThreshold` to control subsequent super-step iterations. The runtime terminates execution if the output frontier is empty.

GRAssembler library consists of separate implementations of schedules, storage formats, and algorithms.

2.4.1 Compiler Optimization

Useless interface elimination: While the proposed abstract interfaces comprehensively cover all schedules, storage formats, and algorithms, certain combinations of schedule, storage format, and algorithm may render some interfaces unnecessary. If an implementer leaves the function body in Table 2 empty, GRAssembler identifies the emptiness by checking if the function body consists only of a terminator instruction and eliminates the associated call sites. For instance, the VM schedule does not require `initGlobal`, the COO storage format does not utilize `searchVID`, and the BFS algorithm does not invoke `apply`. To minimize graph processing latency, the GRAssembler compiler analyzes the library functions and removes redundant function calls.

Atomic operation elimination on vertex value: In graph processing, multiple edges are often loaded and vertex values updated concurrently, necessitating the use of atomic operations in `sum` to ensure correctness. However, if a vertex value is modified by only one thread, atomic operations are unnecessary. GRAssembler conservatively removes synchronization operations when all three of the following conditions are met: First, the `initTopology` method does not access its frontier argument, ensuring no aliasing of vertices or edges. Second, the `getNextEdgeID` method maps thread IDs directly to vertex IDs, guaranteeing that only one thread can access a given vertex. Third, the super-step direction is PULL, permitting the mapped thread to perform the vertex update exclusively. For example, when the COO storage format op-

erates in the PULL direction with the VM schedule, no multiple GPU threads attempt to update the same vertex, making it safe to avoid atomic operations. The GRAssembler compiler analyzes the library to identify synchronization points that can be safely removed and transform atomic operations into non-atomic ones only when the constraints are satisfied.

2.4.2 Tuning Space of GRAssembler

Table 2.5 compares the tuning options available in GRAssembler with those in existing frameworks [6, 7, 8, 11]. Unlike previous frameworks, GRAssembler offers the broadest range of schedules and storage formats and introduces new features such as storage format auto-tuning and CTA size optimization. For instance, while GraphIt [8] provides 336 tuning combinations—the largest prior to this work—**GRAssembler supports 4480 combinations for tuning options**. Notably, because some options, such as CTA size, are numeric and counted here with only two values, the true number of possible combinations is far greater than 4480. Below are additional tuning options available in GRAssembler, complementing the schedules and storage formats:

CTA size influences the ratio of active warps during GPU execution. Adjusting the CTA size helps memory-intensive applications hide memory latency effectively [28]. This tuning option is newly introduced in this work.

Blocking enhances graph locality by employing tiled graph partitioning. Gluon [15] introduces various partitioning strategies based on traversal direction (axis and dimension) and partitioning standard (edge or vertex), and these strategies have been integrated into GRAssembler. Blocking is especially beneficial for topology-driven applications where traversing all edges in a sin-

Table 2.5: The optimization space available in modern GPU-based graph processing frameworks

	GRAssembler	**GraphIt**[8]	**Gswitch**[7]	**Gunrock**[6]	**SEP-Graph**[11]
Storage format Auto-tuning	O	X	X	X	X
Storage format	COO, CSR, ELL, DIA, Gshard	CSR	COO, CSR	COO, CSR	COO, CSR
CTA Size Optimization	O	X	X	X	X
Blocking	O	O	X	X	X
Schedule	VM, EM, WM, CM, TWC, TWCE, STRICT	VM, EM, WM, CM, TWC, TWCE, STRICT	WM, CM, TWC, STRICT	VM, EM, TWC,	CM
frontier Data Structure	Queue, BitMap, ByteMap, Counter	Queue, Bitmap, ByteMap	Queue, Bitmap	Queue, Bitmap	Queue
Direction Optimization	O	O	O	O	O
Frontier Deduplication	O	O	X	O	O
Frontier Ordering	O	O	O	O	O
Number of Available Options	**4480**	**336**	**68**	**96**	**64**

gle iteration is crucial, such as in PageRank.

Frontier data structure defines how active vertex sets are represented. This work supports multiple options, including queue, bitmap, bytemap, and counter. A queue stores active vertex IDs, while maps represent activation states as boolean values. These diverse data structures are particularly suited for data-driven or model-driven applications, such as BFS or Delta-SP. However, some frameworks optimized for topology-driven applications that traverse all edges continue to use simpler structures. For applications that do not require detailed active edge information, this work introduces the counter, a novel data structure that only tracks the size of the frontier by incrementing its value for each ASet_{out}.add(vid) call. The counter is particularly efficient for applications where the frontier is used solely to determine emptiness.

Direction specifies whether the neighbor edge list is accessed based on the source vertex (push) or the destination vertex (pull) [10].

Frontier deduplication eliminates redundant computations [8]. In graph processing, the same vertex may be activated multiple times, leading to unnecessary work. By deduplicating these activations, redundant computations are removed, although this incurs a deduplication cost. Deduplication is mandatory when duplication impacts correctness.

Frontier ordering determines whether an active vertex is processed in the next iteration or deferred, based on a global threshold. This option is particularly advantageous for applications like Delta-SSSP [29, 30].

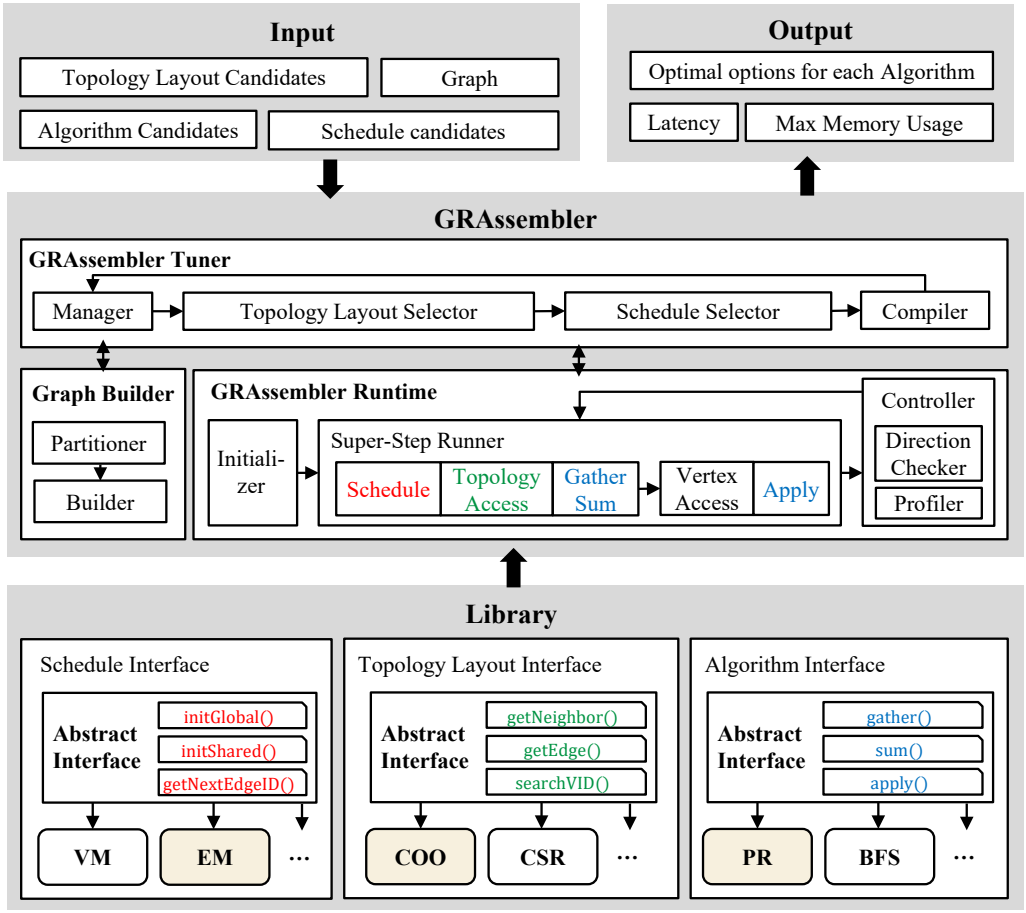


Figure 2.7: Overview of GRAssembler. GRAssembler consists of a tuner, builder, and runtime. The GRAssembler runtime uses libraries structured as schedule, storage format, and algorithm interfaces.

2.5 Evaluation

We evaluates the performance of GRAssembler by comparing it against existing graph frameworks [7, 8] using four algorithms across nine graphs. The evaluation also showcases the extendability of GRAssembler through two case studies.

The experiments were conducted on an NVIDIA GeForce RTX 3090 GPU, which features 10,496 CUDA cores, 82 streaming multiprocessors, 6MB of L2 cache, and 24GB of memory. The host system uses an Intel(R) Core(TM) i7-8700 CPU. The datasets used for the evaluation include nine graphs commonly used in prior works, such as GraphIt [8] and Gunrock [6]. These graphs are soc-orkut (OK)[31], uk-2005 (UK)[31], soc-twitter-2010 (TW)[31], soc-LiveJournal (LJ)[27], indochina-2004 (IC)[27], hollywood-2009 (HW)[27], roadNetCA (RN)[27], road usa (RU)[32], and road central (RC) [27].

The evaluation focuses on four algorithms: PageRank (PR)[33], Connected Components (CC)[34], Breadth-First-Search (BFS)[35], and Delta-SSSP (DS)[30]. Each algorithm showcases unique operational features. PR and CC update vertex values in every super-step, representing consistent update workloads. BFS dynamically adjusts its processing direction (push or pull) based on the ratio of the frontier, making it an effective test for `checkDirection`. Delta-SSSP introduces dynamic thresholding by modifying the delta value, which is implemented via the global threshold in the algorithm interface.

For comparison, this study evaluates GRAssembler against GraphIt [8] and GSwitch [7]. Both frameworks are known for their exploration of diverse graph processing options, including direction control, schedule selection, frontier

data structures, and active vertex ordering.

2.5.1 Overall Performance

Figure 2.8 demonstrates that GRAssembler achieves significant performance improvements for most applications. Specifically, GRAssembler attains 2.21x and 1.30x speedups compared to GSwitch and GraphIt, respectively. In detail, it achieves speedups of 2.33x, 1.84x, 1.66x, and 3.38x over GSwitch, and 1.52x, 1.77x, 0.99x, and 1.07x over GraphIt for PR, CC, BFS, and DS, respectively.

By leveraging function templates and minimizing function arguments, Grassembler reduces the overhead associated with assembling abstract interfaces. First, it reduces the number of function arguments by utilizing global symbols. Second, it replaces function pointers with class templates and function templates. Passing device function pointers to GPU kernels complicates compiler analysis, which limits optimization opportunities such as inlining. Using function templates resolves polymorphic calls at compile time, enhancing the effectiveness of compiler optimizations.

For PR and CC, GRAssembler significantly outperforms other frameworks, primarily due to its CTA size optimization. GPUs perform better with specific thread and block sizes for coalesced memory accesses. For example, using 1024 threads achieves better performance than 512 threads for the IC and RC datasets in PR. The CTA size optimization is particularly beneficial for the EM schedule due to its efficient coalesced memory access to edge data. For instance, in the PR algorithm on the UK dataset, CTA size optimization enhances the performance of EM with COO, CSR, ELL with COO, and Block-COO by 71

GRAssembler also demonstrates superior performance for the CC algo-

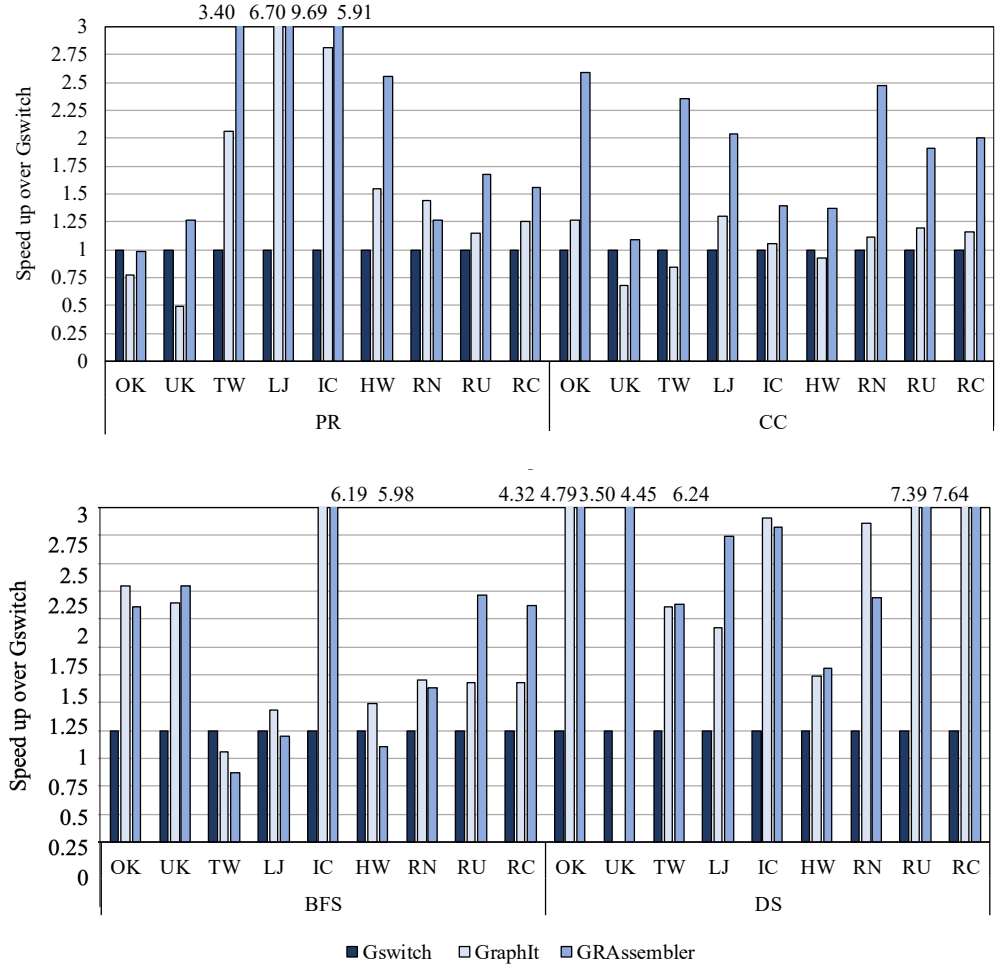


Figure 2.8: A comparison of performance between GRAssembler and state-of-the-art graph processing frameworks, including GraphIt and GSwitch. Each graph illustrates the speedups achieved over GSwitch. The evaluation utilizes four algorithms applied to nine graphs on an NVIDIA GeForce RTX 3090 GPU.

Table 2.6: The optimal options of Figure 2.8

Alg	Dataset	Topology	Schedule	Direction	Frontier	Thread #
PR	LJ, HW	Block-COO	EM	Push	Counter	512
	OK, TW	Block-COO	EM	Push	Counter	512
	RC	Block-COO	EM	Push	Counter	1024
	RN, RU	CSR	VM	Pull	Counter	512
	IC	CSR	TWC	Push	Counter	1024
	UK	CSR	TWCE	Push	Counter	512
BFS	RC, RU	CSR	TWCE	Push	Queue	512
	RN	CSR	VM	Push	Queue	512
	LJ, OK	ELL + CSR	VM	Pull	RQnB	512
		ELL + CSR	TWCE	Push	Queue	512
	IC	CSR	VM	Pull	QnB	512
		CSR	TWCE	Push	Queue	512
	TW	CSR	CM	Pull	RQnB	512
		CSR	TWCE	Push	Queue	512
	UK	CSR	TWCE	Pull	RQnB	512
		CSR	TWCE	Push	Queue	512
	HW	CSR	CM	Pull	RQnB	512
		CSR	STRICT	Push	Queue	512
CC	HW, RC, LJ,	Block-COO	EM	Push	Counter	512
	TW, RU, OK	Block-COO	EM	Push	Counter	512
	RN	CSR	VM	Pull	Counter	512
	UK	CSR	TWCE	Pull	Counter	512
	IC	ELL + COO	EM	Pull	Counter	512
DS	RN	COO	TWCE	Push	Queue	512
	HW, RC	CSR	TWCE	Push	Queue	512
	LJ, OK	CSR	TWCE	Push	Queue	512
	TW, RU, UK	CSR	TWCE	Push	Queue	512
	IC	CSR	CM	Push	QnB	1024

rithm by identifying optimal storage format configurations such as CSR and Block-COO, depending on the dataset. Additionally, the CC algorithm iterates the super-step until no updates occur. Unlike GraphIt, which only supports two frontier options (bitmap and bytemap) and must verify the activity status of all vertices and the presence of active vertices in the set, GRAssembler supports counters as frontiers. This eliminates the need to check all vertices, thereby reducing termination overhead.

For BFS, GRAssembler adapts its tuning options based on the dataset, as shown in Table 2.6. Unlike other algorithms, BFS dynamically switches between pull and push directions during execution. Consequently, the optimal tuning configuration involves two sets of directions, schedules, and frontier data structures for datasets such as LJ, OK, IC, TW, UK, and HW. However, GRAssembler incurs higher overheads for BFS compared to other frameworks due to the algorithm's simplicity and the complexity of its tuning options. Since the apply function in BFS is empty, the abstraction overhead disproportionately affects execution time. As a result, GRAssembler, with its more extensive abstractions, experiences greater overhead and lower performance for BFS. To mitigate this, GRAssembler dynamically adjusts its schedule and frontier structures based on the return value of the `checkDirection` function. Incorporating intensive online tuning techniques [7, 36] that support more than two dynamically changing options could further enhance BFS performance.

Finally, GRAssembler surpasses other frameworks on the DS algorithm. Table 2.6 presents the tuning options used for optimal performance. For example, QnB refers to using a queue for the input frontier structure and a map

for the output frontier structure, while RQnB indicates using a reverse queue for the input frontier structure.

2.5.2 Tuning Performance and Cost

GRAssembler identifies the optimal tuning options that result in the shortest execution time. Compared to the second-best solution, GRAssembler achieves speedups of up to 2.04$times$ and a geomean speedup of 1.16$times$. Notably, 52.78% of the second-best tuning results utilize storage formats that differ from the optimal ones, underscoring the necessity of exploring diverse tuning options to find the best configuration. The execution times for the optimally tuned applications evaluated range from 0.19 milliseconds to 1.17 seconds, with an average of 33.75 milliseconds. These represent speedups ranging from 1.36x to 450.33x (a geomean speedup of 21.48x) compared to the worst-case execution times, translating to an average reduction of 401.63 milliseconds and up to 2.7 seconds. Detailed performance results for different tuning options are provided as a case study in Chapter 2.5.5.

Tuning with GRAssembler takes up to 2 hours for each application and dataset, compared to 10 minutes for GraphIt. This longer duration is because GRAssembler evaluates 14 times more candidates than GraphIt. However, the tuning time gap is less than the factor of 14, as GRAssembler reduces overhead by terminating candidates early if their latency exceeds the best-known result. Despite the relatively high tuning cost compared to the execution time, tuning graph processing options remains critical since graph applications are often executed multiple times at runtime. Additionally, if the auto-tuner could leverage prior tuning results to eliminate inefficient candidates, the tuning

cost for GRAssembler could be significantly reduced.

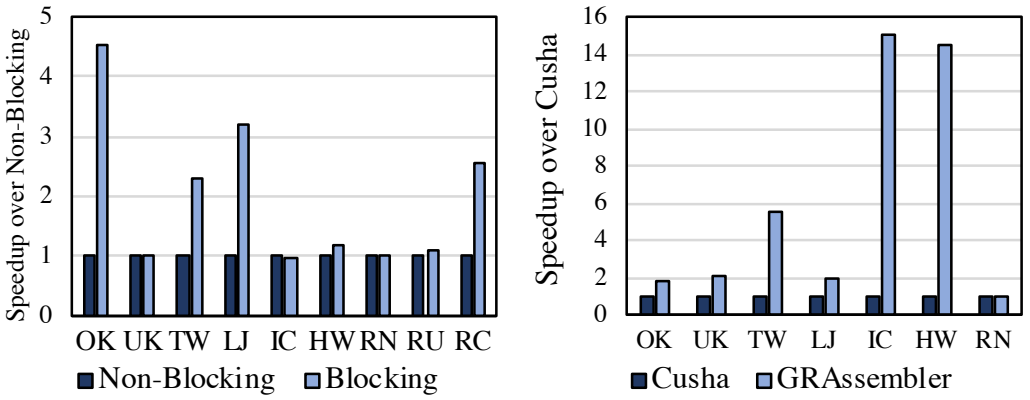
2.5.3 Line of Code Analysis

GRAssembler separates schedule and storage format into distinct interfaces, resulting in modularized implementation. The total lines of code for the schedule implementation amount to 1540 lines, distributed as follows: VM (82), EM (84), CM (185), WM (141), TWC (292), TWCE (238), STRICT (395), and Interface Utilities (123). Similarly, the storage format implementation comprises 644 lines, with contributions from COO (114), ELL (118), CSR (131), Gshard (166), and Interface Utilities (115).

In contrast, GraphIt implements its processing model with 1044 lines of code. Of this, 64% (669 lines) are devoted to device functions that access and manipulate graph data. Adding a new storage format in GraphIt requires an in-depth understanding of its processing model and modifications to a significant portion of the model's codebase, increasing the complexity of extending its functionality.

2.5.4 Abstraction Overhead

To evaluate the overhead introduced by the GRAssembler interface abstraction and integration, this work compares the execution times of GRAssembler and GraphIt using identical tuning options. The selected tuning options correspond to those used for the optimal GraphIt execution. The results indicate that GRAssembler incurs a 21.1% longer geomean processing time compared to GraphIt. Specifically, PR, CC, BFS, and DS exhibit 15.6%, 15.2%, 34.0%, and 20.0% longer processing times on geomean, respectively.



(a) Block-COO (b) Cusha

Figure 2.9: Extending the storage format library. (a) illustrates the speedup of Block-COO compared to COO using the GRAssembler interface. (b) presents the performance improvement of the GShard storage format over the original GShard implementation in Cusha [17].

The results highlight that GRAssembler experiences higher overhead for BFS than other applications. This is attributed to BFS's relatively simple computation algorithm combined with its more complex optimal tuning options. The abstraction overhead is more pronounced for BFS, where the lightweight computations amplify the impact of additional abstraction layers.

2.5.5 Case Study 1: Extendability

This case study highlights the extendability of GRAssembler by detailing the implementation of two example storage formats. The results demonstrate that GRAssembler achieves performance equal to or exceeding that of the original implementations.

Block-COO: Blocking is a widely used optimization technique to enhance data locality when accessing vertex values. As a case study, this evaluation describes how a blocked version of COO was implemented using the abstractions

provided by GRAssembler. In a blocked graph, the neighbor edges of a vertex are grouped into multiple edge lists, and virtual vertices are introduced to represent these blocks. The GRAssembler interfaces, including `getNeighbor`, `getEdge`, and `searchVID`, were adapted to handle this blocked representation. The `getNeighbor` method was designed to return a sublist of the blocked edge list, where the edges in the sublist correspond exclusively to the virtual vertex assigned to that block. Figure 2.9a shows that the Block-COO implementation achieves up to a 4.53x speedup over COO when using Push and EM without CTA optimization. This demonstrates that the proposed interface supports seamless integration of blocked graph implementations. In contrast, while GraphIt [8] supports blocking optimization, it lacks the ability to generate a block-wise connected neighbor list for a vertex, restricting the application of blocking optimization to EM scheduling alone.

Cusha: Cusha [17] introduces GShard, a storage format that reorders and blocks edge data to enhance performance. The implementation of GShard on GRAssembler follows a process similar to that of Block-COO. In the Cusha processing model, the separated `gather` and `sum` operations are performed during the *apply step*.

GRAssembler provides support for `topologyGather`, which enables the integration of GShard within its framework. The GRAssembler compiler detects the presence of `topologyGather`, replaces the `gather` operation in the processing model with `topologyGather`, and incorporates the original `gather` operation into the `initTopology` implementation. Figure 2.9b demonstrates that the GRAssembler implementation delivers performance comparable to or exceeding that of the original Cusha implementation. One contributing factor is

that the GRAssembler implementation avoids asynchronous updates, which can lead to unnecessary synchronization during atomic updates. This reduction in synchronization overhead improves processing times, particularly for the TW, IC, and HW datasets.

2.5.6 Case Study 2: Impact of Tuning

Figure 2.10 compares the performance of various tuning options. The evaluation includes four storage formats (COO, CSR, ELL+COO, Block-COO), two schedules (EM, TWCE), and two directions (Push, Pull) as part of the tuning configuration. Figure 2.10 presents the sorted performance results across these different tuning combinations. The comparison reveals three key characteristics of graph processing performance.

First, expanding the tuning space is essential for achieving optimal performance. The Block-COO storage format demonstrates its superiority as it is part of the best-performing tuning configuration, delivering nearly twice the performance of the second-best configuration (CSR, TWCE, and Pull). Since existing frameworks lack support for Block-COO, they are unable to discover this optimal tuning option.

Second, it is critical to account for the synergistic impact of combining tuning options. While the Block-COO storage format contributes to the best-performing configuration, it is also part of the worst-performing configuration, with a performance gap of 4.7x. The results highlight that the way storage format, schedule, and other options are combined can significantly influence processing time. Neglecting even one tuning option could lead to missed opportunities for substantial performance improvements.

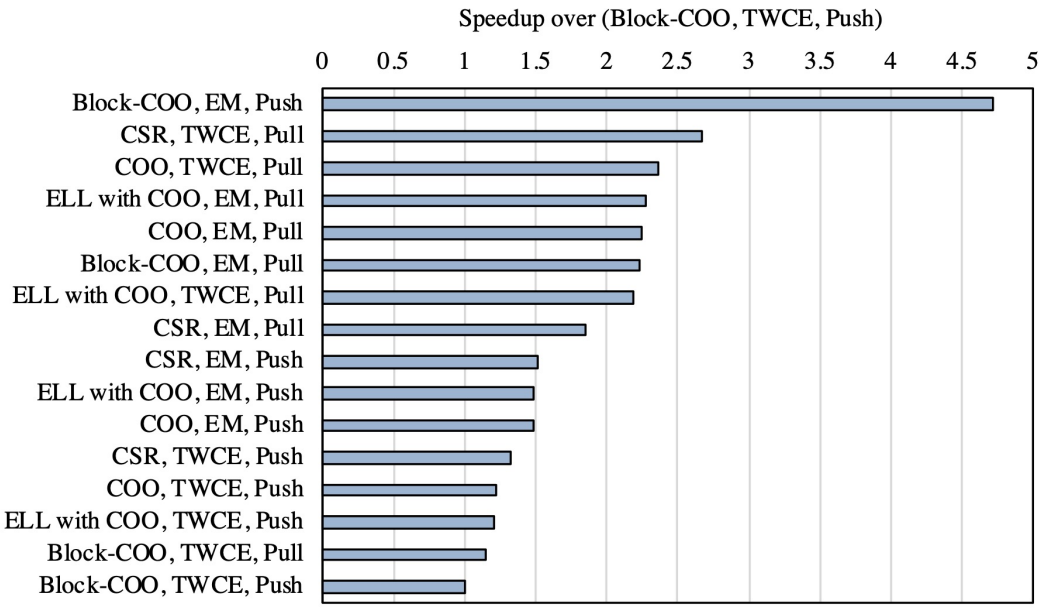


Figure 2.10: Comparison of performance for various tuning options for PR on LJ. The evaluation includes four storage formats (COO, CSR, ELL+COO, Block-COO), two schedules (EM, TWCE), and two directions (Push, Pull). The baseline configuration is set to (Block-COO, TWCE, Push).

Third, incorporating diverse tuning options beyond just schedule and storage format is equally important. For instance, the direction has a significant effect on performance. By integrating a variety of tuning options, including direction, GRAssembler is able to achieve superior performance outcomes.

2.5.7 End-to-End Comparison with Existing CPU and GPU Framework

The table shows the end-to-end performance comparison between GRAssembler and three different graph processing frameworks: DGL [37, 38], PriorityGraph [39], and cuGraph [40]. We use Deep Graph Library (DGL) and PriorityGraph as CPU baseline. DGL is widely used for graph analysis and GNNs,

Table 2.7: End-to-End Performance Comparison with DGL CPU [37, 38], PriorityGraph [39], and Cugraph [40](Second).

	PageRank (20 iterations)				BFS			
	DGL	PGraph	cuGraph	GRAssembler	DGL	PGraph	cuGraph	GRAssembler
UK	0.27	0.03	0.05	**0.01**	0.11	**0.01**	0.14	0.01
LJ	3.11	1.38	0.11	**0.06**	2.63	**0.03**	0.31	0.05
HW	1.82	0.87	0.12	**0.08**	1.39	0.12	1.64	**0.05**
IC	6.10	1.97	0.32	**0.23**	6.80	0.23	OOM	**0.14**
RC	0.23	0.08	0.01	**0.01**	6.10	0.08	0.12	**0.02**
OK	5.17	2.31	OOM	**0.15**	3.55	**0.03**	1.55	0.09
TW	30.00	15.37	OOM	**0.53**	28.48	**0.22**	OOM	0.26

and PriorityGraph supports OpenMP parallelization using the Ligra [41] library. The GPU baseline is cuGraph 24.12 with CUDA version 12.0. The CPU evaluation is conducted on a server with two Intel(R) Xeon(R) Silver 4210 CPUs @ 2.20GHz and 128GB of memory. Since GRAssembler targets optimization within a GPU, we evaluate its performance on a single CPU server, not a multi-cluster CPU server. The GPU evaluation is performed on an Nvidia RTX 4090 GPU paired with an AMD Ryzen 9 3950X 16-core processor and 128GB of memory. Note that the GPU evaluation with cuGraph and GRAssembler shows the end-to-end performance comparison, factoring in host-to-device memory copy overhead and all initialization processes required to start the application.

Chapter 2.5.7 shows that GRAssembler shows better overall performance than other frameworks. Since PageRank (PR) accesses all edges 20 times, it involves relatively more computation than BFS. Consequently, GPU frameworks outperform CPU frameworks, DGL and PriorityGraph. With the BFS algorithm, GRAssembler also exhibits good performance for the HW, IC, and RC benchmarks, whereas PriorityGraph performs well on the UK, LJ, TW, and

OK benchmarks compared to GPU-based graph processing frameworks. Since BFS is a traversal algorithm, if the diameter of the root node (i.e., the number of depths from the root vertex) is small, the process cannot have enough workload to conceal memory transfer overhead. For instance, UK, LJ, TW, and OK have diameters of only 7, 7, 17, and 12 with root vertex 1, respectively, so CPU frameworks demonstrate better performance.

GRAssembler shows 1.86x and 12.26x speedups compared to cuGraph with PR and BFS, respectively. We believe this performance gain comes from autotuning with diverse options. First, not all combinations of GRAssembler outperform cuGraph. For example, LJ shows 1.76x better performance than cuGraph, but the combination of CSR and VM takes longer than cuGraph. This is because LJ has a less skewed graph, where the combination of CSR and WM creates better synergy. Second, the performance gain of BFS comes from direction optimization [35], which cuGraph does not seem to perform. The absence of this optimization has a significant impact on social graph datasets such as HW, LJ, and OK. Since social graphs exhibit heavy-tailed distributions, they contain more super-nodes (nodes with many edges, such as users with many friends or followers) and require direction changes (Push -> Pull) due to the presence of many active nodes in the middle of iterations. Third, GRAssembler can support larger graphs compared to cuGraph. CuGraph encounters Out-of-Memory errors while processing OK and TW with PR, as well as TW and IC with BFS. However, GRAssembler can execute these inputs. Although GRAssembler faces Out-of-Memory errors with the OK and Gshared or COO format, alternative storage formats, such as CSR, can be used to resolve this issue.

2.6 Summary

Our insight into decoupling schedule and storage format from graph processing simplifies the expansion of the graph tuning space. Based on this insight, this work introduces a graph processing abstraction comprising schedule, storage format, and algorithm, grounded in the characterization of graph programs. Subsequently, we propose GRAssembler, which implements our processing model and abstract interfaces, incorporating various graph processing optimizations.

CHAPTER 3

CR^2: COMMUNITY-AWARE COMPRESSED REGULAR STORAGE FORMAT

This chapter introduces a novel graph storage format called CR^2, community-aware, and degree-ordered subgraphs representation. To achieve high performance, CR^2 leverages the locality of skewed graph structures by clustering densely connected vertices into community-aware subgraphs. By decomposing vertex IDs into cluster IDs and local IDs, CR^2 represents each vertex using only its local ID, significantly reducing memory usage. To address the SIMT structure of GPUs, CR^2 further partitions the graph into degree-ordered subgraphs, where all vertices within a subgraph have a uniform, regularized number of edges. This regularization eliminates the need for offset arrays in sparse data compression, enabling fine-grained workload balancing across GPU warps while maintaining lower memory usage. The combination of these features makes CR^2 as an effective solution for compressing graphs for GPU processing.

3.1 Necessity of A New Storage Format

As demonstrated in Chapter 2, GRAssembler effectively identifies better graph processing combinations through autotuning, outperforming existing approaches. While exploring various schedule, storage format, and optimizations

for Chapter 2, we observed that workload imbalance is one of the most critical challenges in GPU-based graph processing, especially when handling real-world graphs exhibiting skewed characteristics. These graphs often follow a heavy-tailed or power-law distribution, where a small subset of vertices is connected to a disproportionately large number of edges [21]. This skewed vertex degree distribution leads to workload imbalances on GPUs, reducing resource utilization efficiency. To address this problem, both schedule and storage format can be used. schedule redistributes imbalanced edges across parallel threads, while storage format regulates edge distribution using statically predefined structures. GRAssembler addresses this issue by exploring various optimization combinations.

While exploring storage formats, we observed that storage format can be further optimized by reconsidering `getNeighbor`. `getNeighbor` does not need to return all neighbor edges at once; instead, it is sufficient to return the edges incrementally during the gather step. Regulating the neighbor edge set and processing multiple neighbor edges in smaller chunks can help balance the workload more effectively. This regulation particularly benefits applications that iterate over all edges, such as PageRank.

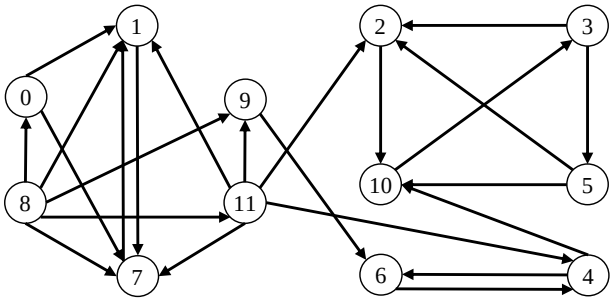
Additionally, we observed that storage format plays a critical role in memory usage. As shown in Chapter 2.5.7, memory usage presents challenges when processing large graphs on GPUs with limited memory capacity. To handle such graphs, efficient memory management is essential. The interesting thing is that `getEdge` only needs to return the necessary information, such as source ID, destination ID, and weight. Thus, storage format does not need to store all graph information in memory.

Motivated by these observations, we began exploring advanced optimizations for storage format. To address workload balancing or memory usage in graph algorithm processing, existing works [12, 16, 17, 19, 23, 24, 25, 26, 42, 43] have proposed various graph storage formats. However, these approaches do not fully account for the skewed topologies of graphs or the SIMT architecture characteristic of GPUs. Consequently, they suffer from intra-warp workload imbalances. To the best of our knowledge, no existing graph storage format simultaneously achieves fine-grained workload balancing, low memory overhead, and high performance.

3.2 In-depth Analysis of Related Storage Formats

A storage format specifies the source and destination of graph edge data within GPU memory. storage format plays a critical role in graph processing for three main reasons. First, a storage format determines the arrangement of graph data in memory, which directly impacts data locality. Second, it dictates the memory footprint required to store and process the graph. With limited GPU memory, a compact storage format enables the processing of larger graphs. Third, a storage format influences how efficiently a GPU handles graph data. By evenly distributing edges across threads, a storage format can mitigate workload imbalances caused by the skewed structure of graphs. To optimize data locality, minimize memory usage, and balance workloads, numerous storage formats [12, 16, 17, 19, 23, 24, 25, 26, 42, 43] have been proposed.

In this chapter, we explore three representative storage formats used on a GPU: Compressed Sparse Row (CSR), Cusha [17], and Tigr [19].



(a) Example graph

src id	0	1	2	3	4	5	6	7	8	9	10	11	end
offset	0	2	3	4	6	8	10	11	12	17	18	19	24

edge list	1	7	7	10	2	4	6	10	2	10	5	1	0	1	7	9	11	6	3	1	2	5	7	9

(b) Compressed Sparse Row (CSR)

Shard 0

dest id	1	2	2	1	0	1	3	1	2
src id	0	3	5	7	8	8	10	11	11
src value	v_0	v_3	v_5	v_7	v_8	v_8	v_{10}	v_{11}	v_{11}

Shard 1

dest id	7	7	5	6	4	7	6	4	7
src id	0	1	3	4	6	8	9	11	11
src value	v_0	v_1	v_3	v_4	v_6	v_8	v_9	v_{11}	v_{11}

Shard 2

dest id	10	10	10	9	11	9
src id	2	4	5	8	8	11
src value	v_2	v_4	v_5	v_8	v_8	v_{11}

(c) G-Shards [17]

virtual src id	0	1	2	3	4	5	6	7	8	8	9	10	11	11	end
offset	0	2	3	4	6	8	10	11	12	14	17	18	19	21	24

edge list	1	7	7	10	2	5	6	10	2	10	4	1	0	1	7	9	11	6	3	1	2	4	7	9

(d) Tigr [19]

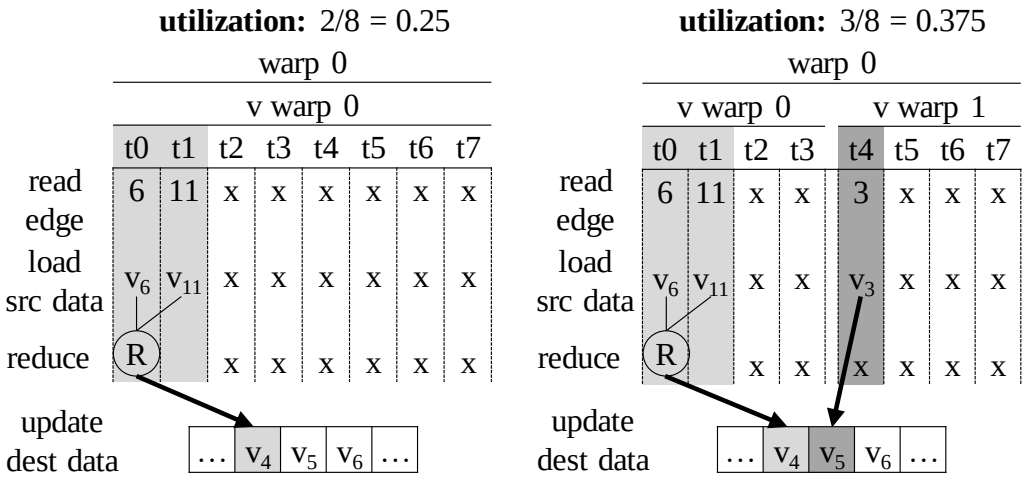
Figure 3.1: Example graph and storage formats.CSR is the most widely used storage format, while G-shards and Tigr are specifically designed for GPUs.

Compressed Sparse Row (CSR) is a space-efficient storage format designed for sparse graphs. CSR stores neighbor edge information (incoming or outgoing) of the base vertices(source or destination) in a single contiguous array and maintains offset indices for opposite vertices (destination or source) that point to its associated edges. CSR is designed to reduce the storage required for vertex IDs, so it uses less memory than other storage formats, such as adjacency matrices or COO. Generally, CSR uses three arrays to represent a graph:

- `edge list`: Contains each vertex's incoming or outgoing neighbor stored in a contiguous block.
- `offset`: Holds indices pointing to the ranges in the `edge list` corresponding to each vertex. A vertex's neighbors can be accessed by iterating through the `edge list` from `offset[vid]` to `offset[vid+1]`.
- `weight`: Stores the values or attributes associated with each edge.

Figure 3.1b shows how CSR represents the example graph using outgoing direction in Figure 3.1a.

The most straightforward approach to processing CSR on a GPU involves mapping each thread to a single vertex. However, this method often encounters challenges related to data locality and workload imbalance. Since CSR does not account for the graph's topology, it may fail to group densely connected vertices together, resulting in reduced cache efficiency. Additionally, because vertices have varying degrees, this strategy assigns threads workloads of unequal sizes. When warp consists of 32 threads, threads processing vertices with fewer edges must remain idle until threads handling vertices with more edges complete their tasks. This problem leads to the underutiliza-



(a) Virtual warp size: 8

(b) Virtual warp size: 4

Figure 3.2: Inefficiency of CSR-based graph processing [43] when the physical warp size is 8

tion of GPU lane.

The virtual warp-centric model (VWC) [43] assigns each vertex to a virtual warp and allows the lanes within the warp to process the edges of the assigned vertex. However, VWC struggles to maximize GPU lane utilization due to its statically fixed virtual warp size. Figure 3.2 illustrates scenarios where VWC results in lane underutilization. For instance, when the virtual warp size is set to 8, processing vertex 1 leaves six thread lanes idle (Figure 3.2a). Conversely, with a virtual warp size of 4, processing vertices 1 and 2 within separate virtual warps leaves five thread lanes unused (Figure 3.2b). While smaller virtual warp sizes can enhance lane utilization, they also increase the time required to process a single vertex. Additionally, smaller virtual warps make sequential reading of the edge list within the warp more challenging.

G-Shards, proposed by Khorasani et al.[17], divides a graph into several

Table 3.1: Theoretical memory requirements for various storage formats. Here, V and E represent the sets of vertices and edges, respectively. The parameter f indicates the ratio of additional virtual vertices introduced in the graph, while r represents the memory reduction achieved by the compressed vertex IDs in CR^2.

Representation	**Edge Representation**	**Value**	
		Vertex	Edge
CSR [43]	$\|E\| + \|V\| + 1$	$\|V\|$	$\|E\|$
G-Shards [17]	$2\|E\|$	$\|V\| + \|E\|$	$\|E\|$
Tigr [19]	$\|E\| + (1 + f_t)\|V\|$	$\|V\|$	$\|E\|$
CR^2	$(1 - r)\|E\| + (1 + f_c)\|V\|$	$\|V\|$	$\|E\|$

sub-graphs known as *shards*[44], which serve as units for parallel computation [45]. Each shard processes a distinct subset of destination vertices. If V_i represents the set of destination vertices in shard i, then $\cup_i V_i$ encompasses all vertices in the graph. Each shard stores a list of incoming edges in ascending order based on source vertex indices using four arrays:

- `dest id`: Contains the indices of destination vertices.
- `src id`: Stores the indices of source vertices.
- `src value`: Holds values corresponding to each vertex in `src id`.
- `weight`: Contains values associated with each edge.

Figure 3.1c show how G-Shards represent the example graph in Figure 3.1a.

Although G-Shards improve lane utilization and ensure regular memory access by allowing each thread to process edges alternately, they require significantly more memory space than CSR. Table 3.1 compares the memory requirements of different storage formats. Considering that the number of edges is generally much larger than the number of vertices (Table 3.4), G-Shards demands approximately three times more memory space than CSR. In Figure 3.1c, the gray boxes highlight the additional data required by G-Shards

compared to CSR. Given the limited memory capacity of GPUs, G-Shards may not be suitable for processing large-scale graphs.

Tigr [19] addresses workload imbalance by virtually splitting an original vertex into multiple virtual vertices and redistributing their edges. While Tigr reduces workload imbalance across warps, it still encounters inter-warp workload imbalance issues. Furthermore, similar to G-Shards, Tigr suffers from significant memory overhead. The graph boxes in Figure 3.1d illustrate the additional memory usage in Tigr compared to CSR. Although the extra memory overhead appears small, Tigr duplicates each vertex for every partition size. This duplication results in substantial memory usage for vertices with very high degrees. For instance, in the soc-twitter-2010 dataset, the maximum vertex degree reaches 698k. With a partition size set to 8, this vertex is duplicated over 82k times. Consequently, Tigr's memory requirements far exceed those of CSR.

In conclusion, to the best of our knowledge, none of the existing storage formats for GPUs simultaneously improve workload balance and data locality while minimizing memory usage.

3.3 Design of CR^2

This work introduces CR^2, a novel GPU-based storage format that is both community-aware and degree-ordered. This chapter outlines the design objectives of CR^2 and explains the methods used to create community-aware and degree-ordered subgraphs. Table 3.2 provides an overview of the terminology employed in this chapter, along with example values for a scenario where vertex

IDs are represented using 8 bits.

3.3.1 Design Goals of CR^2

Motivated by the limitations of existing storage formats, this work proposes a novel GPU-based storage format designed to meet the following objectives: First, the storage format should minimize GPU memory usage for storing graphs. Reducing the memory footprint allows GPUs to efficiently process larger graphs. Second, the storage format should leverage the inherent properties of input graphs. Specifically, it should exploit the high locality of skewed graphs with clustered edges to reduce memory usage and enhance performance. Third, the storage format must align with GPU architectural characteristics (e.g., warp size) to maximize performance in GPU-based graph processing frameworks. A strong correlation between the storage format design and GPU architecture will enable to achieve high performance without requiring extensive performance tuning or optimization.

This work accomplishes these objectives through the use of community-aware subgraphs and degree-ordered subgraphs, as detailed in Chapter 3.3.2 and Chapter 3.3.3.

3.3.2 Community-aware Subgraph

This work introduces community-aware subgraphs, leveraging the observation that many real-world graphs exhibit community structures, where most vertices are densely connected within specific groups [46, 47, 48]. CR^2 creates these subgraphs by applying graph vertex reordering. Vertex reordering algorithms, such as label propagation and hierarchical clustering [49, 50, 51],

Table 3.2: Terminology about a target graph and CR^2

Contents	Terms	Values in Figure 3.5
# of vertices	$\|V\| \sim 2^v$	12
# of edges	$\|E\|$	24
Vertex ID size	M ($M \geq v$)	8
Cluster size	$2^N \times 2^N$	$2^4 \times 2^4$
# of clusters	2^{v-N}	3
Warp size	2^w $(= 2^5)$	4 $(= 2^2)$
# of degree-*n* subgraphs	$w + 1$	3
# of virtual vertices	$\|V_v\|$ $(= (1 + f_c)\|V\|)$	16

reorder vertex IDs based on the graph's structure, such as its community organization, and extract highly dense clusters along the diagonal of the adjacency matrix.

For example, Figure 3.4b depicts a real-world graph after reordering, based on the graph shown in Figure 3.4a. The graph's axes represent source and destination vertices, and each dot indicates the presence of an edge. Given that some real-world graphs, such as power-law graphs, often exhibit strong community structures, vertex IDs can be reordered to reflect these community-aware arrangements.

As shown in Figure 3.4b, the sparse matrix of the reordered graph reveals that many edges are concentrated along the diagonal, with large communities appearing as square blocks. On the reordered graph (Figure 3.5a), CR^2 defines $2^N \times 2^N$ square clusters along the diagonal and divides the graph into two parts: intra-cluster and inter-cluster graphs.

Intra-cluster graph: The intra-cluster graph consists of multiple fixed-size clusters. Each cluster contains edges located within a $2^N \times 2^N$ square along the diagonal of the sparse matrix (Figure 3.4c). For instance, with a cluster size of $2^4 \times 2^4$, the reordered graph (Figure 3.5a) can be clustered as shown in Fig-

ure 3.5c. In CR^2, vertex IDs are represented as a combination of *cluster ID* and *local vertex ID*. The *global vertex ID* can be derived from these components using the following equation:

$$\text{global vertex ID} = (\text{cluster ID} \ll N) \mid \text{local vertex ID}. \tag{3.1}$$

This allows each cluster to represent its intra-cluster graph using only the *local vertex ID*, effectively reducing memory costs for data structure construction by half. For example, with a cluster size of $2^4 \times 2^4$ ($N = 4$), the upper 4 bits of the *global vertex ID* correspond to the *cluster ID*, while the lower 4 bits correspond to the *local vertex ID*. Specifically, vertex 34 in Figure 3.5a can be represented as *cluster ID 2* and *local vertex ID 2*, as illustrated in Figure 3.3.

Inter-cluster graph: The inter-cluster graph comprises edges that span across clusters (dashed area in Figure 3.4c). In Figure 3.5c, these edges lie outside the rectangular boxes representing the intra-cluster graph. Unlike the intra-cluster graph, the vertex IDs in the inter-cluster graph are not a direct combination of *cluster ID* and *local vertex ID*. Since the inter-cluster graph represents edges that do not fall within clusters, there is no straightforward rule to derive *global vertex ID* from *local vertex ID*. Consequently, the inter-cluster graph uses *global vertex IDs* to represent its structure. As a result, CR^2 maintains one global inter-cluster graph, in contrast to the multiple intra-cluster graphs.

By separating intra-cluster subgraphs from the original graph, the community-aware subgraphs in CR^2 leverage highly skewed graphs' locality while reducing overall memory usage. Compared to CSR, these subgraphs require additional memory for the source ID array because each vertex is divided

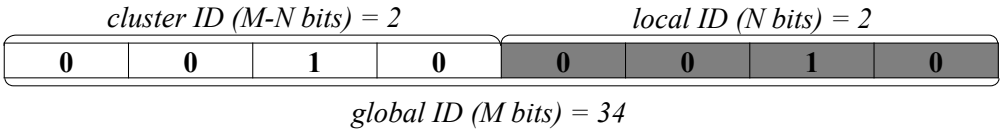
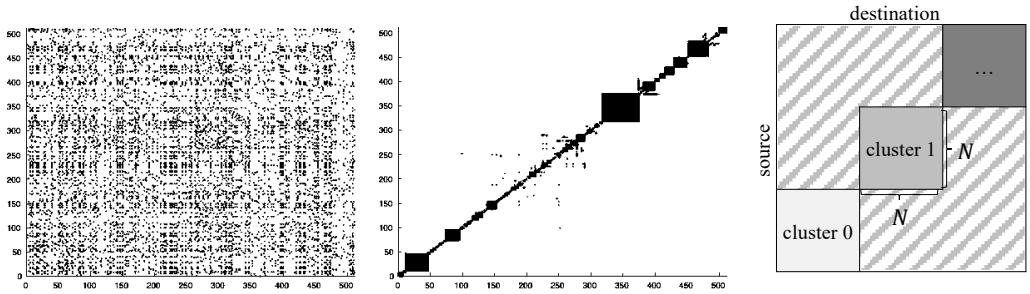


Figure 3.3: Vertex ID representation in the intra-cluster graph. Here, the vertex ID size M is larger than or equal to the order of the number of vertices, v.

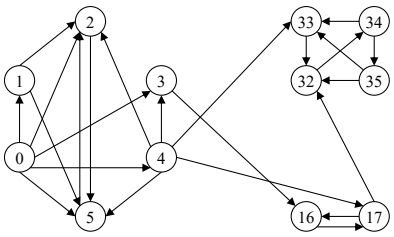


(a) Random order (b) Community-based (c) Clustering in CR^2

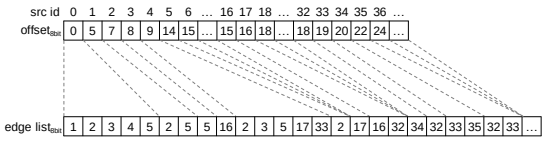
Figure 3.4: Random- and community-based reordered sparse matrices, along with the clustering concept in CR^2, where the cluster size is fixed at N $\times$ N.

across multiple subgraphs, with each subgraph containing only a subset of the vertices, as depicted in Figures 3.5c and 3.5d. However, this additional memory overhead is offset by the inter-cluster graph, which represents edge lists using only 4 bits, significantly reducing memory usage.

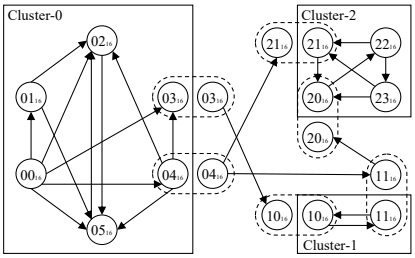
Furthermore, the vertex-degree regulation introduced in the following chapter eliminates the need for offset arrays in all community-aware degree-ordered subgraphs. As a result, for most skewed graphs with highly clustered edges, CR^2 not only reduces memory consumption compared to CSR but also achieves the first and second design goals.



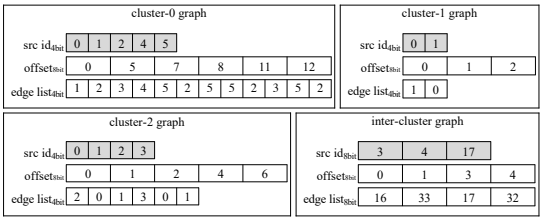
(a) Community-aware reordered graph



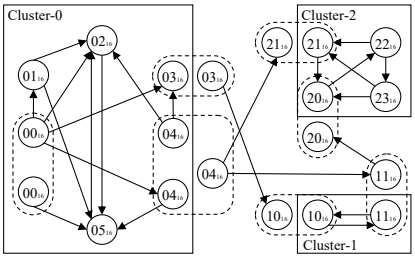
(b) Community-aware reordered CSR



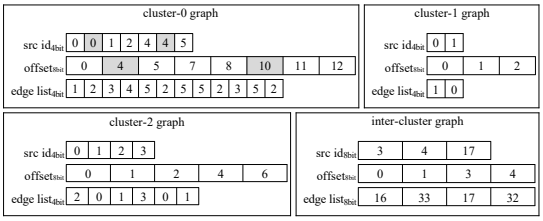
(c) Community-aware clustered graph



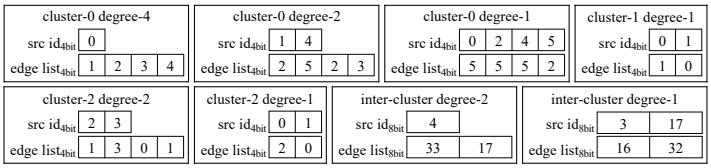
(d) Community-aware clustered representation



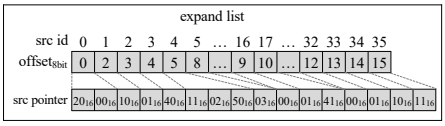
(e) Vertex split graph



(f) Vertex split representation



(g) representation



(h) expand list

Figure 3.5: Design process of CR^2 using two novel design solutions, community-aware subgraphs, and degree-aware vertex splitting.

3.3.3 Degree-Ordered Subgraph (Degree-*n*)

Addressing workload imbalance is a crucial challenge in GPU programming, as it significantly influences application execution times. To mitigate the imbalance caused by the power-law degree distribution found in real-world graphs, prior works [19, 52] have proposed splitting vertices into multiple virtual vertices, each with a limited number of edges. While vertex splitting can reduce workload imbalance across threads or warps, these approaches often fail to account for GPU architectural characteristics, such as warp size, thereby missing opportunities for performance improvements. For instance, Tigr uses a fixed constant α for vertex splitting but still faces workload imbalance issues for edges below the α threshold.

This work introduces degree-ordered subgraphs with vertex-degree regulation, leveraging GPU architecture properties to achieve improved workload balance. Given that the warp size (2^w) in modern GPUs is 2^5, this approach regulates vertex degrees to powers of two (up to 32). Vertices are initially split so that no vertex has more than 32 edges. Subsequently, their edge counts are divided by 2^n, where n ranges from 4 to 0. As a result, all vertices have degrees that are powers of two (up to 32). Vertex-degree regulation is applied separately to both intra-cluster and inter-cluster graphs. Figure 3.5e illustrates the graph after vertex splitting based on incoming edges from Figure 3.5c. For example, vertex 04_{16} in cluster 1 of Figure 3.5c is split into smaller vertices, each containing 1 and 2 edges.

The vertex splitting process increases the number of source vertices (e.g., vertex 00_{16} and vertex 04_{16} in Figure 3.5e) compared to the community-aware

clustered graph (Figure 3.5c). Consequently, the number of elements in the offset array also grows since each split vertex has a distinct number of edges (Figure 3.5f). While the increase in source ID elements does not introduce significant memory overhead in intra-cluster graphs due to *local ID* representation, the growth of the offset array can pose challenges, as it typically uses a 32-bit data type. For inter-cluster graphs, although vertex and offset increases occur, their impact is relatively smaller than in intra-cluster graphs, especially when clusters contain numerous edges.

The number of edges for each vertex in the intra- and inter-cluster graphs ranges from 2^0 to 2^5 ($= 2^w$) due to the vertex splitting technique. This enables the grouping of graphs based on vertex edge counts. For each cluster in the intra-cluster and inter-cluster graphs, six subgraphs are constructed, termed Degree-*n* subgraphs, where each vertex has exactly n edges ($n = 1, 2, 4, 8, 16, 32$). Managing Degree-*n* subgraphs separately provides two key advantages:

Elimination of Offset Arrays: Degree-*n* subgraphs eliminate the need for offset arrays, reducing the memory overhead introduced by vertex splitting. Since all vertices within a Degree-*n* subgraph have n edges, the edge list index can be directly calculated by multiplying the source ID index by n. This approach minimizes memory usage and reduces memory bandwidth requirements.

Improved Warp Lane Utilization: Degree-*n* subgraphs maximize warp lane utilization, addressing workload imbalance. While vertex splitting reduces thread or warp imbalance, GPUs still face underutilization of warp lanes or incur additional computational overhead, such as binary searches. For instance, in VWC processing, each virtual warp processes the edges of a

single source vertex. This can lead to warp underutilization when edge counts do not align with the warp size (Figure 3.2). Previous approaches [53] allow multiple vertices to share a warp but require binary searches to identify the source vertex for each thread. Degree-*n* subgraphs address both issues by pre-defining vertex degrees as powers of two, ensuring vertices fit precisely within warp sizes. For example, four vertices with eight edges each fully utilize all 32 warp lanes, achieving 100% lane utilization.

Figure 3.5g presents the final representation of CR^2 without offset arrays. Unlike Figure 3.5f, each cluster and the inter-cluster graph is further subdivided based on vertex edge counts. For instance, cluster 0 in Figure 3.5f now contains Degree-1, Degree-2, and Degree-4 subgraphs, all managed without requiring offset arrays (Figure 3.5g).

3.3.4 Expand List for Split Source Access

Certain algorithms, such as BFS or SSSP, update only portions of a graph during each iteration. These algorithms achieve better performance by limiting access to active vertices rather than traversing all vertices in the graph [10]. However, the CR^2 representation shown in Figure 3.5g does not inherently support selective access to active vertices. This limitation arises because vertices are divided into multiple virtual vertices distributed across various Degree-*n* subgraphs, and their exact locations are not directly identifiable. Accessing active vertices requires additional pointers to their corresponding virtual vertices.

To facilitate access to virtual vertices, CR^2 introduces an expand list, which maintains pointers mapping real vertices to their virtual counterparts, akin

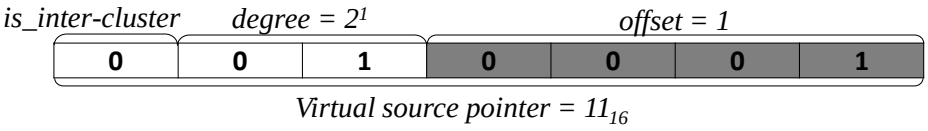


Figure 3.6: Virtual Source pointer representation in the expand list. The pointer uses the first three bits to represent cluster information and the last four bits to represent the offset.

to the CSR representation (Figure 3.5h). To minimize the memory overhead of the expand list, this work encodes each pointer using a combination of three components: a *is_inter-cluster* flag, a *degree* value, and an *offset* value, as detailed in Figure 3.6. The *is_inter-cluster* flag, represented by a single bit, indicates whether the virtual vertex resides in an inter-cluster subgraph. The *degree* value specifies the degree of its corresponding Degree-*n* subgraph, requiring $\lceil \log(w+1) \rceil$ bits, where w represents the warp size. The *offset* value denotes the location of the virtual vertex within its Degree-*n* subgraph, using $\log |V_v|$ bits. In the worst case, the offset size is $2v - w$ bits, as each vertex connects to all other vertices with 2^{v-w} virtual vertices, given a total of 2^v vertices.

Additionally, the encoded source pointer does not include the cluster ID because the cluster ID is inherently embedded in the vertex ID. This encoding strategy reduces the memory footprint of the expand list while enabling efficient access to virtual vertices in CR^2.

3.4 Processing CR^2 Graph

This chapter explains the fundamental processing steps of CR^2, including how to construct a CR^2 graph storage format and execute GPU kernels utilizing CR^2. It is important to note that various reordering algorithms [49, 50] can be ap-

plied to rearrange graph data to leverage the community structure inherent in many graphs. For simplicity, this work assumes the dataset is pre-ordered and describes the method for generating CR^2 based on the reordered graph.

3.4.1 Building CR^2

Algorithm 3.1: CR^2 builder

```
   Input: E : Edge set of input graph
   Output: G_cr2 : CR^2 graph
1  G_cr2 ← CR2Graph()
2  if needReordering then
3  |  E_r ← reorderGraph(E)
4  countDegree(E_r, G_cr2)
5  allocSubGraph(G_cr2.intraGraph)
6  allocSubGraph(G_cr2.interGraph)
7  insertEdges(E_r, G_cr2)
```

Algorithm 3.1 outlines the procedure for constructing CR^2 in both push and pull directions from an input graph. The process involves four primary steps. First, if the input graph is not preordered, the builder applies a reordering algorithm, such as the Rabbit-order algorithm [50], to enhance the graph's community structure (line 3). Second, the builder calculates the in-degree and out-degree of each vertex to gather necessary degree information (line 4). Third, leveraging the degree information, the builder allocates memory for the storage formats corresponding to push and pull directions and prepares intermediate data structures to ensure edges are placed correctly (line 5 to line 6). The intermediate data, *remainEdges* and *prefixEdgeLoc*, track the number of remaining edges and the current insertion locations for edges, respectively (line 7). Finally, the builder inserts the edges into the preallocated memory space for the CR^2 representation (line 7). In this implementation, the en-

Algorithm 3.2: countDegree(E_r, G_{cr2})

```
   Input: E_r : Edge set of reordered graph
          G_cr2 : CR^2 graph
1  foreach e ∈ E_r do
2      (src, dest) ← e.getEdgeInfo()
3      if (src / N) = (dest / N) then
4          fetch_and_add(G_cr2.intraGraph.inDegree[dest], 1)
5          fetch_and_add(G_cr2.intraGraph.outDegree[src], 1)
6      else
7          fetch_and_add(G_cr2.interGraph.inDegree[dest], 1)
8          fetch_and_add(G_cr2.interGraph.outDegree[src], 1)
9  end
```

tire process is executed on the CPU.

To construct CR^2, the builder examines the in-degree and out-degree of each vertex within the intra-cluster and inter-cluster graphs. Algorithm 3.2 presents the degree counting algorithm used for CR^2. For each edge, the builder determines whether the edge belongs to the intra-cluster graph or the inter-cluster graph based on the cluster size constant N (line 3). Next, the builder increments the corresponding degree arrays by 1 (line 4 to line 8) in the identified graph. To enhance efficiency, the degree counting process is executed in parallel (line 1).

The core task of the builder involves storing edges into the memory space based on the degree information obtained earlier. Algorithm 3.3 outlines the edge insertion process into the allocated memory. The builder determines its insertion location for each edge by iterating through the degrees from the maximum degree constant w to 0. As shown in line 6 to line 14, the insertion process for edges within the intra-cluster graph and push direction is carried out as follows: First, the builder verifies whether the *deg*-subgraph has sufficient remaining space to store the edge (line 8). Second, it retrieves the ap-

Algorithm 3.3: insertEdges(E_r, G_{cr2})

```
   Input: E_r : Edge set of reordered input graph
          G_cr2 : CR^2 graph
1  for e ∈ E_r do
2      (src, dest) ← e.getEdgeInfo()
3      if (src / N) = (dest / N) then
4          // Insert edge to intra-graph for push direction
5          // remainEdges and prefixEdgeLoc contains numbers of remaining degree and
             current locations to insert edge, respectively
6          for deg ← [w,0] do
7              v_loc ← src * (w + 1) + deg
8              if G_cr2.remainEdges.intra.out[v_loc] > 0 then
9                  e_loc ← G_cr2.prefixEdgeLoc.intra.out[v_loc]
10                 G_cr2.intraGraph.edgeList.intra.out[e_loc] ← dest
11                 ...
12                 G_cr2.prefixEdgeLoc.intra.out[v_loc] ++
13                 G_cr2.remainEdges.intra.out[v_loc] −−
14         end
15         // Insert edge to intra-graph for pull direction
16         ...
17     else
18         // Insert edge to inter-graph ...
19 end
```

propriate location for the edge (line 9) and adds the destination vertex of the edge to the edge list for the intra-cluster graph in the push direction (line 10). After completing the insertion, the builder increments the values of the intermediate data for the edge by 1(line 12 to line 13). This process completes the construction of the CR^2 storage format.

3.4.2 Launching Kernels with CR^2

To process graph applications, each kernel is responsible for handling the Degree-*n* subgraphs from both intra- and inter-cluster graphs. Since CR^2 groups all Degree-*n* intra-cluster subgraphs with the same *n* and processes them in a single kernel launch, the number of kernel launches is restricted to twice

Algorithm 3.4: Example of CR^2 processing

```
   Input: deg : Degree of this kernel
          baseIDList: List of base vertex id
          edgeList: List of remaining vertex id
          isPull: Indicator for direction
          workload: Number of virtual nodes
1  eid ← threadIdx.x + blockDim.x * blockIdx.x
2  vid = eid >> deg
3  if vid >= workload then
4  |   return;
5  src = baseIDList[vid]
6  if isDense then
7  |   globalID ← src & 0xffff0000
8  |   dest ← globalID | (DENSE_TYPE *)edgeList[eid]
9  else
10 |   dest ← (SPARSE_TYPE*)edgeList[eid]
11 if isPull then
12 |   swap(src, dest)
13 ...
14 // perform algorithm with source and destination ID.
```

the number of degrees. For example, to process the entire graph represented by CR^2 shown in Figure 3.5g, five kernels—corresponding to degree-1, 2, and 4 intra-cluster subgraphs, as well as degree-1 and degree-2 inter-cluster subgraphs—are launched.

Algorithm 3.4 provides an example implementation of a CR^2 processing kernel that binds edges to threads. The kernel accepts the storage format array, degree (*deg*), direction, and the number of vertices as its input arguments. The storage format array includes *edgeList* and *baseIDList*, which represent source ID lists or destination ID lists, depending on whether the direction is push or pull.

First, the kernel uses the thread ID as the edge ID (line 1), and calculates the corresponding vertex by shifting the edge ID by *deg* (line 2), and performs a validation check (line 3). Second, the kernel retrieves the source ID of the

Table 3.3: The worst time complexity of basic graph operations. $deg(v)$ means the number of edges of the vertex v.

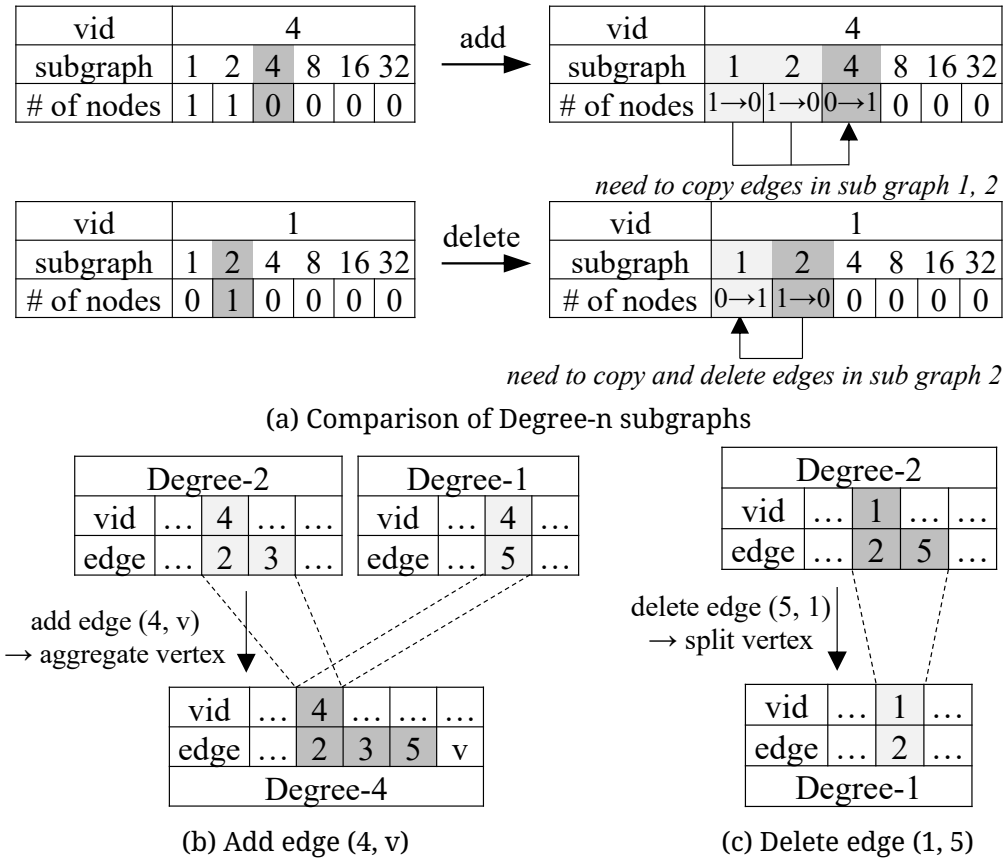
Operation	CSR	CR^2
add & delete	$O(\|V\| + \|E\|)$	$O(\|V\| + \|V_v\|)$
find neighbors	$O(deg(v))$	$O(deg(v))$

edge from the *vid*-th element of the *baseIDList*. Third, the kernel determines the destination ID.

If the kernel is launched for intra-cluster graphs, it calculates the global ID (line 7) and computes the destination ID using the global ID and Equation 3.1 (line 8). For inter-cluster graphs, the destination ID is directly retrieved from the *eid*-th element of the *edgeList*. In the case of a pull direction, the kernel swaps the source and destination IDs. Finally, the kernel executes the graph algorithm, such as PageRank or BFS, using the source and destination IDs.

3.4.3 Adding and Deleting an Edge

Since CR^2 organizes intra-cluster and inter-cluster graphs separately, the initial step for adding or deleting an edge, $e = (u, v)$, is to determine the cluster to which the target edge belongs. This can be checked using the condition $\lfloor u/$*cluster size*$\rfloor == \lfloor v/$*cluster size*$\rfloor$. If the condition evaluates to true, the edge is part of the cluster identified by $\lfloor u/$*cluster size*$\rfloor$. Otherwise, the edge resides in the inter-cluster graph. In CR^2, the edges of a vertex *v* are distributed across Degree-*n* subgraphs based on the vertex's in-degree. For example, as shown in Figure 3.7a, the edges of vertex 4 (in cluster-0) from Figure 3.7c are distributed across Degree-1 and Degree-2 subgraphs before the addition of a new edge. When the new edge is added, vertex 4's in-degree increases to 4



(a) Comparison of Degree-n subgraphs

(b) Add edge (4, v)

(c) Delete edge (1, 5)

Figure 3.7: Addition and deletion of the edge represented by CR^2 in Figure 3.5

(Figure 3.7a). Consequently, the edges originally in the Degree-1 and Degree-2 subgraphs must be migrated to the Degree-4 subgraph (Figure 3.7b). During this migration, CR^2 also updates the expand list to reflect the changes. Deleting an edge operates in the reverse manner (Figures 7a and 7c).

The worst-case time complexity for adding and deleting an edge is summarized in Table 3.3. When adding or deleting an intra-cluster edge, the time complexity for moving edges between Degree-*n* subgraphs is constant because the maximum number of vertices within a cluster is fixed. However, for

inter-cluster edges, the time complexity for relocating edges among Degree-*n* subgraphs becomes proportional to the number of vertices, $|V|$. Additionally, updating the expand list following the CSR format incurs a time complexity of $|V| + |V_v|$. Therefore, the worst-case time complexity for adding or deleting an edge is $O(|V| + |V_v|)$.

3.4.4 Finding Neighbors of a Vertex

To retrieve all neighbors of a vertex v, CR^2 utilizes the virtual source pointers stored in the expand list. Each virtual source pointer contains information about the associated Degree-*n* subgraph and the specific location of the vertex within that subgraph. Using these virtual source pointers, CR^2 can efficiently access all neighbors from the corresponding Degree-*n* subgraphs. After obtaining the virtual source pointers, CR^2 performs neighbor retrieval in constant time. Therefore, the worst-case time complexity is determined by the process of searching for virtual source pointers in the expand list. Since the expand list is represented in CSR format, this search has a time complexity proportional to the number of virtual vertices of v, which is less than $deg(v)$.

3.5 Evaluation

We evaluate the performance of CR^2 using four algorithms across seven datasets, compared to two state-of-the-art frameworks: VWC-CSR [43] and Tigr [19]. VWC-CSR employs CSR as its storage format, optimized as shown in Figure 3.2, while Tigr utilizes the latest storage format techniques. Both frameworks were sourced from their public repositories, and performance was measured for both push and pull directions, selecting the better result in each case.

Table 3.4: Dataset Specification [27]. $edges_d$ indicates the percentage of edges that reside within the intra-cluster graph, where the cluster size is set to $2^{16} \times 2^{16}$, compared to the total number of edges in the graph.

Dataset	# nodes	# edges	Max deg	% $edges_d$	sym
uk-2005(UK)	130K	23M	850	99 %	sym
indochina-2004(IC)	7,415K	302M	56,425	93 %	sym
hollywood-2009(HW)	1,140K	113M	11,467	70 %	sym
soc-twitter-2010(TW)	21,298K	530M	698,112	38%	sym
soc-LiveJournal(LJ)	4,848K	86M	20,333	19%	sym
soc-orkut(OK)	2,997K	213M	27,466	15%	sym
it-2004(IT)	41M	1,151M	1,326,745	92%	asym

The evaluation was conducted on an NVIDIA GeForce RTX 3090 GPU, equipped with 10,496 CUDA cores, 82 streaming multiprocessors, 6MB L2 cache, and 24GB memory, paired with an Intel(R) Core(TM) i7-8700 CPU. All source codes were compiled using the CUDA toolkit 12.0 with the `-O3` optimization flag.

Table 3.4 provides details on the seven datasets used in this study, sourced from the network repository [27]: uk-2005 (UK), Indochina-2004 (IC), hollywood-2009 (HW), soc-twitter-2010 (TW), soc-LiveJournal (LJ), soc-orkut (OK), and it-2004 (IT). The implemented graph algorithms include Breadth-First-Search (BFS) [35], Single-Source-Shortest-Path (SSSP) [30], Connected-Components (CC) [34], and PageRank (PR) [33].

The expand list in CR^2 facilitates optimizations for BFS and SSSP by enabling targeted access to active vertices starting from the root node. Execution times were measured excluding graph building and memory transfer operations. BFS, SSSP, and CC were executed until convergence, while PR was evaluated on a per-iteration basis.

Table 3.5: Execution times (in milliseconds) of four algorithms across eight input graphs are compared for three frameworks: VWC-CSR, Tigr, and CR^2, utilizing a Nvidia GeForce RTX 3090.

	PR (time per round)			BFS		
Graph	VWC	Tigr	CR^2	VWC	Tigr	CR^2
UK	0.33	0.27	**0.26**	1.89	0.48	**0.29**
IC	14.99	10.21	**5.44**	269.71	13.46	**9.38**
TW	57.27	39.26	**28.85**	461.59	35.47	**19.12**
HW	**2.12**	2.50	2.19	13.06	2.82	**1.26**
OK	15.82	16.06	**10.86**	29.63	8.00	**4.82**
LJ	6.91	5.80	**4.84**	25.02	5.17	**4.77**
IT	61.17	60.74	**25.96**	2545.86	116.78	**52.54**
	CC			SSSP		
Graph	VWC	Tigr	CR^2	VWC	Tigr	CR^2
UK	1.90	0.59	**0.38**	1.97	**0.60**	0.72
IC	259.54	17.5	**7.94**	804.98	**74.5**	115.41
TW	440.13	106.14	**72.88**	504.02	**42.81**	53.99
HW	12.13	3.08	**2.81**	42.33	**10.92**	12.42
OK	24.7	18.75	**9.89**	169.67	74.83	**38.28**
LJ	25.08	10.35	**7.57**	138.52	39.08	**27.54**
IT	2527.31	486.19	**35.28**	2509.75	seg fault	**48.96**

3.5.1 Execution Time

Table 3.5 presents the processing times for four graph algorithms evaluated on VWC-CSR, Tigr, and CR^2. CR^2 achieves geomean speedups of 6.47× and 1.55× compared to VWC-CSR and Tigr, respectively. Specifically, CR^2 delivers 1.62×, 15.63×, 8.99×, and 7.70× geomean speedups over VWC-CSR, and 1.42×, 1.74×, 2.11×, and 1.01× geomean speedups over Tigr for the PR, BFS, CC, and SSSP algorithms, respectively.

For the PR algorithm, CR^2 demonstrates 1.65× and 1.43× better performance than VWC-CSR and Tigr. Since PR requires accessing all vertices and edges at every iteration, the improvement highlights CR^2 's superior locality

and workload balancing capabilities. For instance, with the IC dataset, which has a high percentage of intra-cluster edges, CR^2 achieves a 2.76× speedup over VWC-CSR. This result suggests that CR^2 performs particularly well on highly clustered datasets. Additionally, CR^2 outperforms VWC-CSR by 1.98× for the TW dataset, which is well-clustered except for its super nodes (Max deg in Table 3.4). However, datasets like UK and HW, despite being well-clustered, exhibit smaller performance gains due to their relatively low vertex counts, which limit the benefits of locality. Even for datasets like OK and LJ, with fewer intra-cluster edges, CR^2 still shows improvements due to the degree-ordered subgraph technique, which effectively mitigates warp imbalance.

For BFS, CR^2 achieves 13.29× and 1.69× better performance than VWC-CSR and Tigr, respectively. This performance boost stems from the Hybrid-BFS algorithm [35], which maintains frontiers of vertices for each iteration. Unlike Tigr, which lacks a mapping from real to virtual vertices, CR^2 utilizes its expansion list to efficiently access active vertices and their neighbors, avoiding unnecessary processing of inactive vertices and edges. The detailed mechanism is discussed in Chapter 3.5.3.

For the CC algorithm, CR^2 outperforms VWC-CSR and Tigr by 8.22× and 2.12×, respectively. Similar to PR, CC requires accessing all vertices and edges during each iteration. Consequently, CR^2 benefits from its efficient handling of locality and workload balance, particularly on large, well-clustered datasets like IC and IT, where it effectively leverages shortcuts to accelerate iteration completion.

For SSSP, CR^2 achieves 6.83× and 1.01× speedups compared to VWC-CSR and Tigr, respectively. However, unlike the other algorithms, CR^2 shows com-

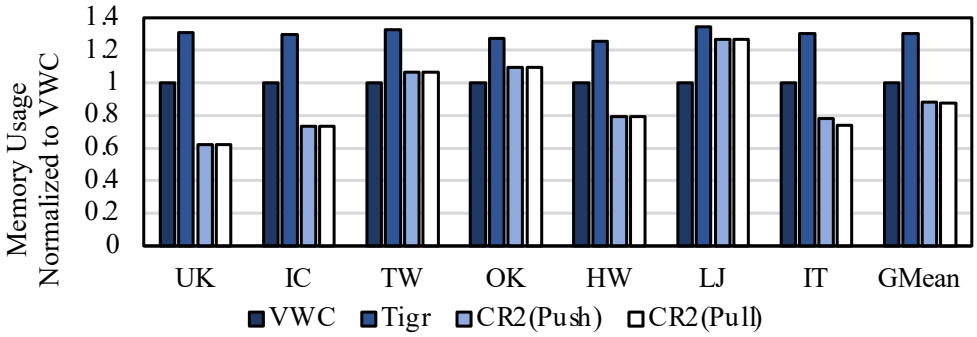


Figure 3.8: Memory usage of VWC-CSR, Tigr and CR^2. Each graph is normalized to the memory usage of VWC-CSR.

parable performance to Tigr. This is because the SSSP algorithm maintains a relatively large frontier for many iterations, as vertex updates frequently impact neighboring vertices. This persistent large frontier results in significant overhead when accessing the expand list, which mitigates CR^2's advantages for SSSP.

3.5.2 Memory Efficiency

This work evaluates the memory usage of VWC-CSR, Tigr, and CR^2 as depicted in Figure 3.8. The measured memory size accounts for the storage format of each framework, excluding the algorithm-specific memory usage. For Tigr, the configuration (K=8) from its original paper [19] and public repository is used. For CR^2, the measurement includes the memory required for the expand list.

On average, CR^2 uses 12.3% and 32.1% less memory compared to VWC-CSR and Tigr, respectively. This efficiency arises from CR^2 's compression of vertex IDs for intra-cluster edges. Datasets such as UK, IC, HW, and IT, which have over 70% intra-cluster edges, demonstrate significant memory reduc-

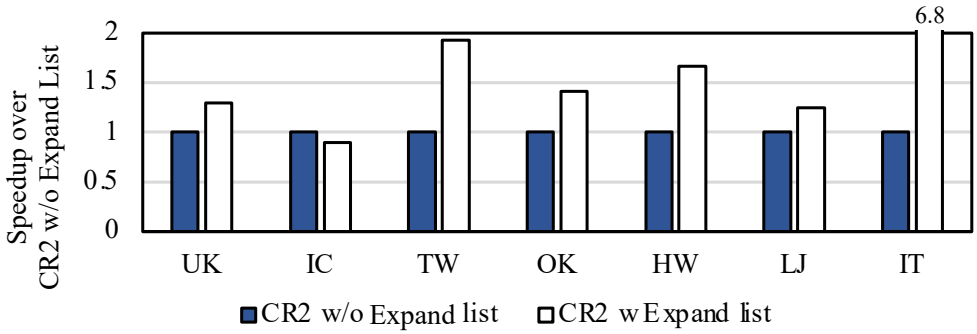


Figure 3.9: Performance comparison between CR^2 without the expand list and with the expand list for the BFS algorithm.

tions with CR^2. For other datasets, CR^2 incurs slightly higher memory usage than VWC-CSR due to the overhead of the expand list.

Similar to Tigr, CR^2 splits vertices into multiple virtual vertices and regulates the number of edges per virtual vertex. However, CR^2 employs finer-grained regulation than Tigr and introduces additional data structures like the expand list. Despite these additions, CR^2 consumes considerably less memory than Tigr. This is attributed to its elimination of the offset array through the use of degree-ordered subgraphs and its compression of vertex IDs with community-aware subgraphs.

3.5.3 Performance using Expand List

Some algorithms update only portions of a graph by reflecting changes in vertex values. By maintaining an active vertex set, such algorithms can restrict processing to vertices in the frontier during subsequent iterations, avoiding the need to traverse the entire graph. This selective processing significantly reduces execution time. However, the use of virtual vertices in Tigr and CR^2

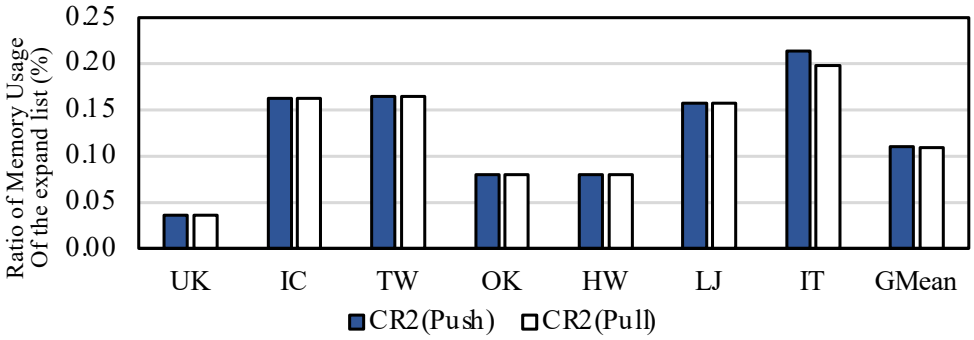


Figure 3.10: Ratio of memory usage of the expand list.

complicates frontier management, as a single vertex is split into multiple virtual vertices distributed across various locations without pointers linking them.

To address this limitation, CR^2 introduces the expand list, which efficiently maps each vertex to its corresponding virtual vertices. This mapping allows seamless support for frontiers, overcoming the challenges posed by virtual vertex distributions.

The benefits of the expand list are evident in Figure 3.9, which demonstrates its impact on the Hybrid-BFS algorithm [35]. Incorporating the expand list results in a 1.72× geomean speedup compared to CR^2 without the expand list. While the expand list introduces additional memory overhead, as shown in Figure 3.10, it accounts for only 11.8% of CR^2's total memory usage on average. This results in a memory footprint comparable to CSR but still lower than that of Tigr.

Consequently, if the system provides sufficient memory resources, the expand list enables CR^2 to trade a modest increase in memory usage for substantial performance gains. This trade-off underscores the expand list's utility in accelerating execution for graph algorithms.

Table 3.6: Storage format build time (*Miliseconds*). The cluster size of CR^2 is $2^{16} \times 2^{16}$.

Dataset	CSR	Tigr	CR^2
UK	40.82	192.22	118.515
IC	582.17	2541.17	2009.93
TW	1620.52	4617.53	4870.29
HW	234.45	903.06	705.615
OK	577.13	1727.45	1661.05
LJ	317.5	772.2	1002.52
IT	10803	10136.9	28726.2

3.5.4 Storage Format Build Time

Figure 3.11 presents the time required to construct storage formats from raw graph datasets. On average, CR^2 takes approximately 3.0× and 1.05× more time than VWC-CSR and Tigr, respectively. The faster build time for VWC-CSR is attributable to its use of CSR, which requires less computational effort compared to the more complex representation techniques employed by Tigr and CR^2. CR^2 takes more build time compared to Tigr is particularly evident with datasets like TW, LJ, and IT. These datasets contain a higher proportion of non-diagonal edges, resulting in more insertions during the construction process and thus increasing the overall build time. It is important to note that the evaluations utilize graph datasets without any additional reordering applied.

3.5.5 Performance Comparison of Push and Pull

The performance of graph algorithms like PR can vary significantly depending on the processing direction, so this work supports both push and pull directions. For PR, the execution time ratios of the push direction to the pull direction for symmetric graphs are observed as 32.2%, 14.6%, 22.4%, 12.4%,

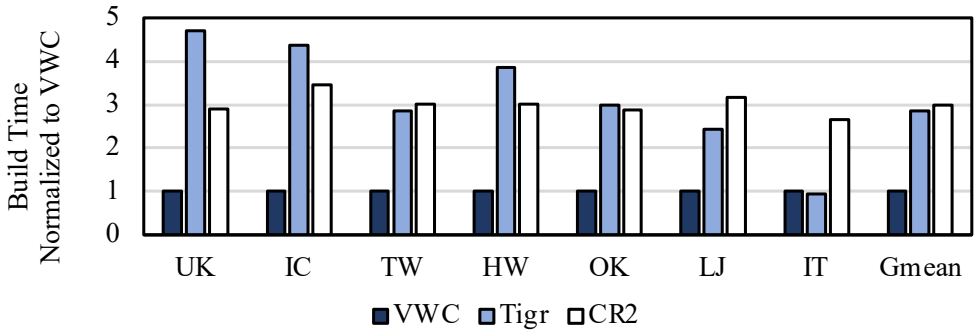


Figure 3.11: Build Time of VWC-CSR, Tigr and CR^2. Each graph is normalized to the build time of VWC-CSR and performs one Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.

151.5%, and 137% for UK, IC, HW, TW, LJ, and OK, respectively.

The differences in performance arise from the inherent characteristics of the graphs. In the push direction, CR^2 can efficiently reuse source data to update outgoing edges, leveraging temporal locality as vertex value accesses are coalesced within the same subgraph. Conversely, in the pull direction, each GPU thread updates one destination vertex, with all threads in the subgraph targeting similar memory locations, thereby exploiting spatial locality.

Graphs with higher $edges_d$ ratios benefit more from the push direction due to its efficient temporal locality utilization. In the evaluation, datasets such as UK, IC, and HW, which exhibit high $edges_d$ ratios, perform better with the push direction. Conversely, datasets like LJ and OK, characterized by lower $edges_d$ ratios, achieve better performance using the pull direction, where spatial locality is more effectively exploited.

3.6 Summary

In Chapter 3, we propose a new graph representation, CR^2, which combines vertex ID compression using community-aware subgraphs with vertex degree regularization through degree-ordered subgraphs. This approach enables high-performance, memory-efficient graph processing on GPUs. The evaluation demonstrates that CR^2 achieves a 1.53× performance speedup while reducing memory usage by 32.1% on average (geometric mean) compared to state-of-the-art techniques.

CHAPTER 4

SPARSEWEAVER: MICROARCHITECTURE FOR ACCELERATING SCHEDULE

This research proposes a new hardware-software collaborative graph processing framework, SparseWeaver, that converts sparse operations in graph processing into dense operations using a new microarchitecture and balances the workloads across GPU threads.

4.1 Necessity of Hardware Acceleration for Schedule

Schedule determines the distribution of edges across threads, making it one of the important optimizations for workload balancing. While storage format can provide a static strategy for achieving a balanced memory access pattern in the graph, it is not always effective. As discussed in Chapter 3.5.1, static approaches are ineffective for algorithms like BFS and SSSP, where active vertices change dynamically across iterations. On the other hand, schedule can be viewed as a method for deciding how to map computations at runtime. This dynamic nature necessitates various schedule techniques designed to optimize workload balancing.

However, software-only solutions come with inherent limitations. First, they introduce additional overhead for remapping edges across warps or cores, requiring extra memory operations to store and share indirect pointers and

additional computations for mapping. Second, the effectiveness of software-based schemes can vary depending on the sparsity and skewness of the graph structure, making it challenging to find an optimal solution for diverse real-world graphs [7, 8, 18].

Some works [54, 55, 56] propose hardware-based solutions, such as load-balancing units for generating edge lists. However, these approaches also face challenges. The additional memory access required to gather edge information introduces overhead. Furthermore, since specialized hardware handles memory reads and writes for edge data independently, these approaches often fail to fully utilize GPU bandwidth. This limitation is exacerbated for memory-intensive graph workloads, where the ability to hide memory stalls using warp-level parallelism is critical.

To balance workloads efficiently while minimizing overhead, we propose leveraging a lightweight, low-overhead hardware accelerator. The fundamental reason for the workload imbalance problem lies in the sparsity of graph workloads, driven by their irregular structure, as illustrated in Figure 4.1. GPUs, traditionally optimized for dense operations, struggle with these irregularities. By introducing compact hardware to transform sparse operations into dense, SIMD-compatible operations, we can address workload imbalance with minimal overhead. Moreover, integrating this hardware into the GPU execution pipeline allows for fine-grained, pipelined execution while preserving the advantages of GPU architecture, such as warp-level parallelism and high memory throughput.

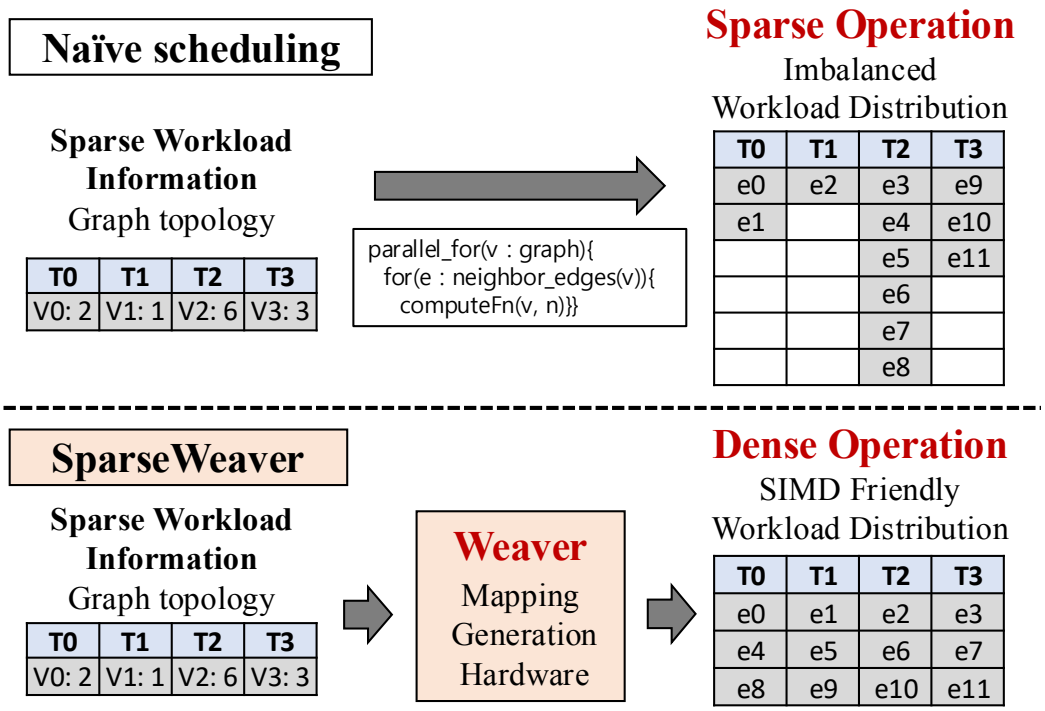


Figure 4.1: Workload imbalance among threads in warps during graph processing caused by the irregular graph structure. SparseWeaver can address workload imbalance problem by SIMD-friendly workload distribution by hardware logic called Weaver

4.2 Revisit Graph Processing on GPU in terms of Schedule

Accessing neighboring edges involves two steps during the gather operation, resulting in sparse operations. The first step retrieves the graph topology of a vertex to identify its neighboring edges. The second step accesses the information of each neighboring edge to perform gather and sum operations. While the first step is inherently dense and easily parallelized across threads, the second step requires threads to process sparse neighboring edges, leading to workload imbalance and poor warp utilization among threads within warps,

as illustrated in Figure 4.1.

This issue is exacerbated when performing gather and sum operations on real-world graphs due to several inherent characteristics of these graphs. First, real-world graphs are often sparse and highly skewed in structure [21, 22]. Second, the neighboring edges of a vertex are frequently unpredictable [44, 57, 58], resulting in irregular memory access patterns for both edge data and vertex properties. Third, real-world graphs can consist of millions to billions of edges, and some vertices may have an exceptionally high degree, with thousands of neighbors [59, 60, 61].

Given these challenges, the workload imbalance problem becomes particularly severe for real-world graphs. Therefore, optimizing gather and sum operations by effectively balancing the workload is essential to achieve high performance.

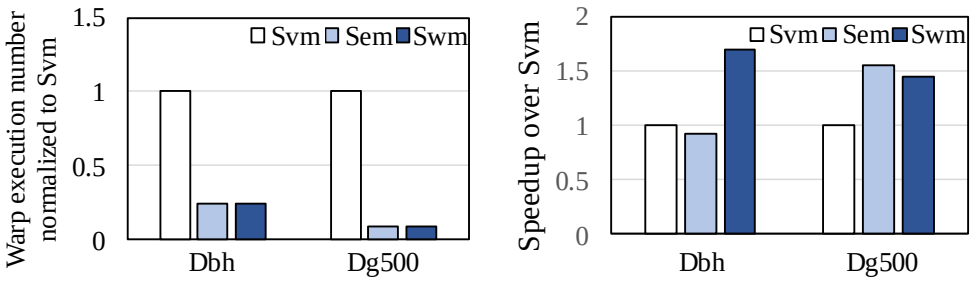
4.3 Deep Dive into Related Schedules

Existing schedules [7, 8, 13, 62] aim to achieve workload-balanced mappings but require computationally expensive operations for synchronization and data sharing among threads. Table 4.1 highlights the additional memory usage and computational overhead involved in these methods.

These schedules address workload imbalance caused by skewed distributions in various ways, but their approaches are not well-suited to the Single Instruction Multiple Threads (SIMT) model. This is because they rely on sharing profiled data among threads and require iterative accesses to that shared data to assign work IDs to threads. For instance, the WM mapping pro-

	VM	EM	WM [7]	CM [7]	TWC [13]	TWCE [8]	STRICT [14]	SparseWeaver
Sharing Granularity	Thread	Kernel	Warp	Block	T, W, B	T, W, B	Kernel	Block
Imbalance	high	low	mid	low	low	mid	low	low
Edge Mem. Access	$2\|V\| + \|E\|$	$2\|E\|$	$2\|V\| + \|E\|$	$2\|V\| + \|E\|$	$2\|V\|+\|E\|$	$2\|V\|+\|E\|$	$2\|V\|+\|E\|$	$2\|V\|+\|E\|$
Shared Mem.	X	X	$3\|B\|$	$3\|B\|$	$3\|B\|$	$6\|B\|$	$3\|B\|$	$4\|B\|$
Global Mem.	X	X	X	X	$3\|V\|$	X	$3\|V\|$	X
Complexity of computing in registration	low	low	mid	mid	high	high	high	low
(Sync, add Kernel, #Atom, #Warp sfhl)	0, 0, 0, 0	0, 0, 0, 0	1, 0, 0, 6	17, 0, 0, 15	1, 0, $3\|V\|$, 6	1, 3, $2\|V\|$, 0	17, 3, 0, 15	1, 0, 0, 0
Complexity of computing in distribution	low	low	high	high	high	high	mid	low
(#BinarySearch, #atomics, #sync)	0, 0, 0	0, 0, 0	$\|E\|$, 0, 0	$\|E\|$, 0, 0	$\|E\|$, 0, 0	0, $\alpha\|E\|$, $\alpha\|E\|$	$\|E\|$, 0, 0	0, 0, 0
Edge Access Locality	low	high	mid	high	mid	mid	high	high

Table 4.1: Comparison of implementation details among existing schedules [7, 8, 13, 14]. $|V|$, $|E|$, $|B|$ means the number of vertices, number of edges, and thread block size, respectively.



(a) Expected warp iteration (b) Speedup over VM

Figure 4.2: Expected warp iteration and performance of software-based schedules. (a) shows expected warp iteration using the PageRank (PR) algorithm with three different schedules (VM, EM, WM) on the bio-human graph (BH) and graph-500-scale19 (G500). (b) shows speedup over VM

cess, illustrated in Figure 4.2a, demonstrates this challenge in detail. The process involves accessing the offset list to profile vertex degrees, creating a degree sum array, broadcasting the total degree within the warp, and updating shared memory. Subsequently, each thread calculates its next work ID by performing a binary search on the shared degree array, which must be repeated for every iteration. This process introduces an $O(n \log(n))$ time complexity for shared memory scans. Similarly, the CM scheduling approach aggregates total degrees at the block level, with searches also occurring at this level. While this improves load balancing compared to WM, it introduces additional overhead due to the increased computational complexity. In general, these schedules offset the runtime costs of mapping generation by the performance benefits gained from workload balancing. However, the inherent inefficiencies in their mapping processes limit their suitability for graph workloads that require both high performance and low overhead.

To mitigate the workload imbalance issue, it is crucial to transform sparse

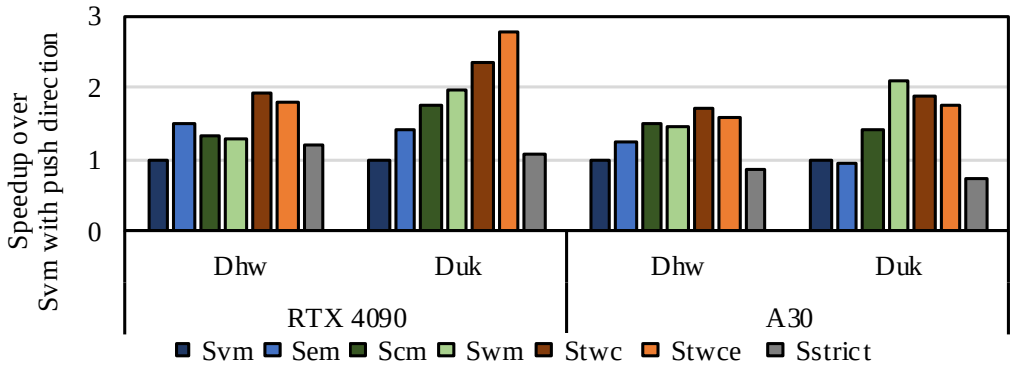


Figure 4.3: Performance comparison with existing software schedule in two Nvidia GPUs (Ampere (A30) and Ada (RTX4090)) using HW and UK dataset with PageRank and GRAssembler. Each graph shows speedups over VM.

operations into dense ones during the second step of edge access, which can be achieved by reorganizing the edge list across threads in a dense manner. Accordingly, software-based techniques [7, 8, 13, 14, 17, 19, 20, 25] propose schedules that redistribute active edges across threads within a warp or core to optimize edge access performance. Table 4.1 provides detailed comparisons of these schedules. Although simpler mappings, such as Vertex mapping (Naive schedule, VM, assigning each vertex and its neighbors to threads) and Edge mapping (EM, allocating each edge to a thread), are straightforward, they often face challenges like workload imbalance and increased memory accesses for edge data. More advanced schedules, including WM [7], CM [7], TWC [13], TWCE [7], and STRICT [14], aim to distribute workloads more evenly by sharing graph topology across blocks, warps, or threads. These methods, however, require additional computation, synchronization, and shared or global memory operations to offset workload imbalances, often achieving higher performance by balancing these trade-offs. Further details are discussed in

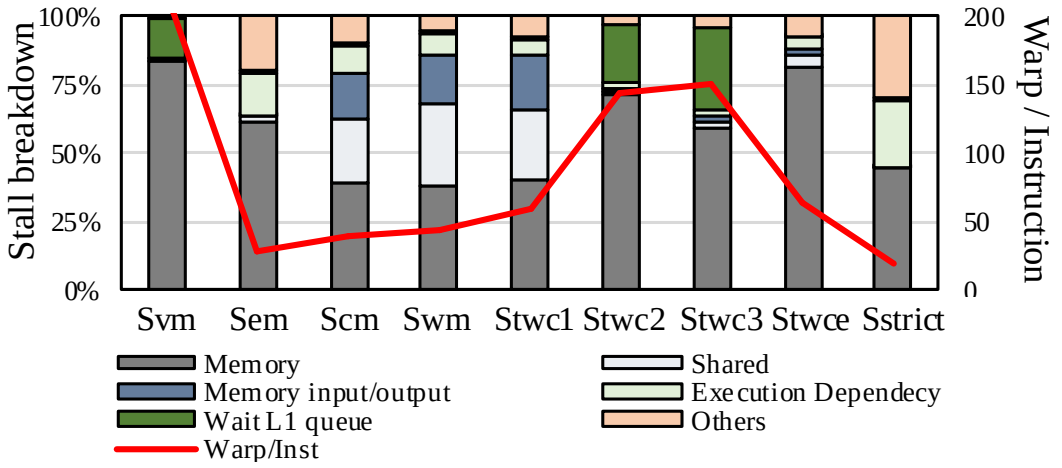


Figure 4.4: Stall breakdown and warp per instruction result of existing schedules with Nvidia GPU (A30) and PageRank using HW dataset

Chapter 4.4.1.

Despite these advances, achieving a balance between workload distribution and minimizing overhead remains a significant challenge, as the overhead is not always effectively concealed. Figure 4.2 illustrates the expected warp iterations and performance of the edge-gathering phase when running the PageRank algorithm on the BH and G500 graph datasets [31]. As depicted in Figure 4.2a, workload imbalance results in VM requiring 4x and 11x more warp iterations compared to WM and EM for the BH and G500 datasets, respectively. While WM and EM achieve similar expected warp iterations by addressing workload imbalance, their performance differs due to varying overheads. Specifically, EM incurs double the memory reads for edge data, whereas WM relies on additional computation and shared memory usage. As shown in Figure 4.2b, WM and EM deliver the best performance for the BH and G500 datasets. However, because the G500 graph has relatively more ver-

tices and fewer edges than the BH graph, it suffers from greater overhead due to the increased computational demand per edge.

Workload imbalance is a prevalent issue in Nvidia GPUs as well. Figure 4.3 illustrates that complex software schedules, such as CM, WM, TWC, and TWCE, frequently outperform VM on Nvidia GPUs, achieving up to a 2.80x speedup by adopting alternative schedules. However, these schedules also come with their own overheads on Nvidia GPUs, as highlighted in Figure 4.4 [1)]. PageRank, for instance, primarily involves one addition and a read/write operation for edges and vertices. Despite this simplicity, some schedules introduce additional stalls, including shared memory stalls (WM, CM, and TWC1), L1 queue waiting stalls (VM or TWC1), or increased warp/instruction latencies.

These findings emphasize the difficulty in identifying the optimal schedule for specific datasets and applications, as schedules often incur varying levels of overhead. While prior works [7, 8, 18] have proposed auto-tuners to optimize graph processing on GPUs, such approaches are inherently time-intensive and introduce abstraction-related costs.

This raises a crucial question: *Is software optimization alone a fundamental solution to this challenge?* The core difficulty in processing graph workloads on GPUs lies in effectively mapping irregular, data-dependent graph tasks onto the GPU's one-dimensional parallel execution units. Existing software-based solutions address this by distributing such irregular tasks across GPU warps at runtime (schedule) or compile-time (storage format) to achieve work-

[1)]The original stall metrics in Nvidia Nsight Compute are named as follows: Memory (long scoreboard), Shared (short scoreboard), Memory input/output (MIO Throttle), Execution Dependency (Wait), Wait for L1 queue (LG Throttle), and Warp/Instruction (average warp latency per instruction issued).

load balance. However, these approaches often necessitate additional computations and synchronizations, as each thread must independently generate and process the required mapping information.

Tools for exploring efficient schedules or storage formats, such as those based on DSE (Design Space Exploration) [18], can aid in selecting the most suitable schedule. Yet, the workload imbalance persists because GPU hardware lacks native support for data-dependent workload mapping within its one-dimensional structure.

A fundamental solution to this problem requires converting sparse operations into dense ones at the hardware level. Unlike software schemes, hardware can achieve this mapping with lower overhead by avoiding redundant computations and synchronized updates needed for thread-specific calculations. Furthermore, this conversion can be accelerated using a dedicated FSM to handle the remapping process. However, designing such hardware presents significant challenges. First, fully offloading the neighboring edge access process might overlook opportunities to leverage the GPU's fine-grained pipeline and microarchitecture. Second, the conversion hardware could become a bottleneck when handling memory-intensive graph workloads. Third, the offloading process could increase memory usage and access demands.

In this research, we analyze the limitations of existing software-based schemes and pinpoint the critical steps required to convert sparse operations into dense ones. We then propose a novel lightweight hardware extension seamlessly integrated into the GPU execution pipeline to capitalize on GPU architecture's advantages while effectively addressing workload imbalance.

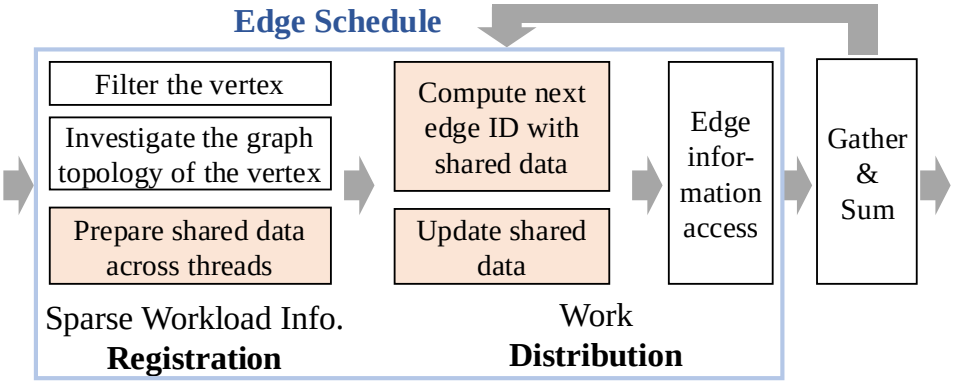


Figure 4.5: Disassemble graph processing by abstracting software-based schedule

4.4 Hardware/Software Co-Design

4.4.1 Software-based Schedule Abstraction

Existing software-based schedules [7, 8, 13, 14] share common patterns in their approach to mapping generation. These schemes typically gather graph topology and generate edge identifiers across warps, blocks, or entire kernels. As illustrated in Figure 4.5, software-based schedules can be broadly categorized into two stages: (1) the **registration stage** and (2) the **distribution stage**.

In the registration stage, threads collect essential data, such as sparse workload information, to facilitate the distribution stage. Each thread processes this data by filtering the base vertex ID (source or destination vertex ID), accessing the base vertex's graph topology, and calculating any required additional information. During the distribution stage, threads execute tasks based on the sparse workload data. Specifically, each thread generates an edge ID

for the current lockstep using shared data and, upon completion, updates the shared data for subsequent iterations. This involves accessing, gathering, and summing edge information through indirect pointers.

Details of these schedules are summarized in Table 4.1. Depending on the target schedule granularity, overheads such as memory usage and computations vary between the registration and distribution stages. For instance, Warp-sharing mapping (WM) focuses on balancing workloads within warps. In the registration stage, each thread loads the graph topology to determine the degree of neighboring edges and the edge indicator of the first neighboring edge. Threads then compute a prefix sum array of degrees and update it in shared memory. In the distribution stage, threads calculate their edge indicators by performing a binary search on the degree prefix sum, incurring a time complexity of $O(n \log(n))$ for shared memory scans.

Generating finer-grained mappings often enhances workload balance but also introduces additional overhead. Schemes targeting kernel-level balance [14] suffer from increased global memory access and kernel launch overheads, while warp- and block-level balancing schemes [7, 8, 13] aim for improved performance. Block-level sharing, however, incurs synchronization and searching overhead. Unlike these software approaches, incorporating lightweight hardware at the core level can eliminate such overhead, avoiding the costs associated with block-level sharing. Since GPUs execute one warp at a time, hardware can dynamically compute and return edge indicators for the executing warp, enabling efficient block-level workload balancing.

As depicted in Figure 4.5, the essence of existing schedules lies in leveraging sparse workload information to distribute edge IDs densely across threads

within warps. By offloading minimal but critical components of scheduling to lightweight hardware, it is possible to reduce overhead and accelerate the process. Well-designed hardware can minimize the need for additional schedule buffers and integrate the scheduling process seamlessly into the GPU pipeline. Consequently, this research proposes to offload the following tasks, highlighted as orange boxes in Figure 4.5: (1) preparing shared data across threads, (2) computing the next edge indicator using shared data, and (3) updating shared data.

4.4.2 Weaver Logic Design

SparseWeaver is designed to generate edge indicators while maintaining shared data through two key components: the Sparse Workload Information Table (ST) and the Dense Work ID Table (DT). The ST stores shared registration data, including vertex IDs (VIDs), degrees of neighboring vertices, and the starting positions of neighbor edges in the edge array. During the registration stage, SparseWeaver collects this information and populates the ST. Conversely, the DT holds Work IDs, such as base vertex IDs and generated edge IDs (EIDs), corresponding to positions in the neighbor edge array. In the distribution stage, SparseWeaver utilizes the DT to return Work IDs for operations such as edge information access, gathering, and summation.

At the core of SparseWeaver, Weaver handles the generation of Work IDs for the warp by scanning and decoding the registered ST entries. To facilitate this, Weaver loads and manages active ST entries in a buffer called Current Entry Data (CED). It then decodes the CED to produce intermediate Work IDs, storing these results in the Output Data (OD) buffer. The workflow and finite

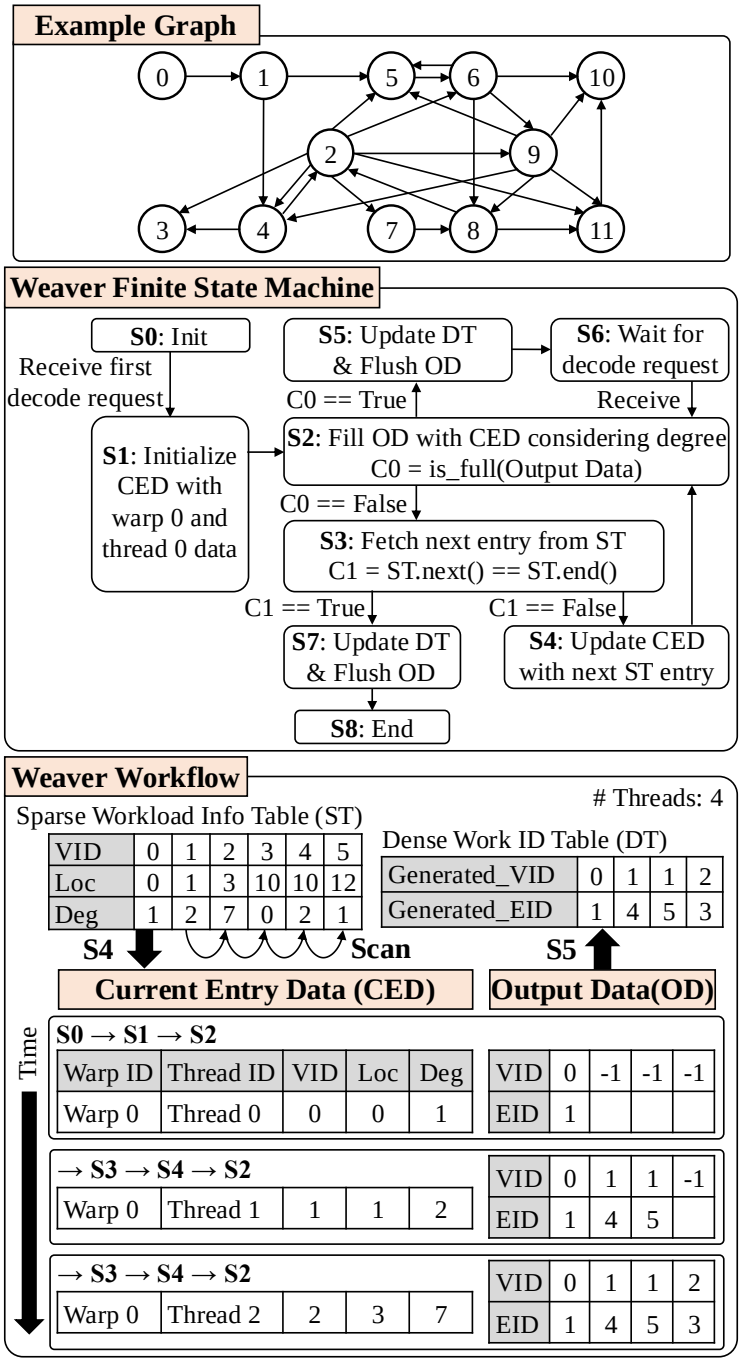


Figure 4.6: SparseWeaver hardware logic called Weaver. The FSM and workflow illustrate how the Weaver works, using CED and OD.

state machine (FSM) of Weaver are depicted in Figure 4.6. The FSM supports two operational modes: filling the OD buffer from multiple low-degree entries (S3 $\rightarrow$ S4 $\rightarrow$ S2) and filling multiple OD buffers from a single high-degree entry (S5 $\rightarrow$ S6 $\rightarrow$ S2).

When processing begins, Weaver initializes by loading the first entry in the CED buffer (S1). It then attempts to populate the OD buffer with Work IDs derived from the CED (S2). If the degree of the current CED entry is insufficient to fill the OD buffer (C0 == False), Weaver fetches the next ST entry (S3, S4) and continues filling the OD buffer (S2). Once the OD buffer is fully populated, the FSM updates the DT (S5) and awaits the next decode request (S6). If all ST entries have been processed, the FSM transitions to its end states (S7, S8). When no valid Work IDs remain, SparseWeaver outputs return empty Work IDs (e.g., -1). Upon receiving a new registration request in the registration stage, the FSM reinitializes to its starting state.

An example of SparseWeaver's operation, assuming a warp contains four threads, is provided in Figure 4.6. Initially, Weaver reads the first ST entry, populating the CED buffer with vertex ID 0, start location 2, and degree 1 during initialization (S0 $\rightarrow$ S1). In the decoding stage (S2), Weaver uses this CED entry to fill one OD buffer entry with vertex ID 0 and edge ID 2. Since the degree of the current CED entry is insufficient to fill the OD buffer, Weaver retrieves the next ST entry, updating the CED buffer (S3, S4). The OD buffer, now requiring three more entries, is further populated using the new ST entry (2, 10, 2), filling two OD entries with (2, 10) and (2, 11). Finally, Weaver processes an additional ST entry (4, 30, 5) to complete the OD buffer with Work ID (4, 30) before transitioning to state S5 in the FSM.

4.4.3 SparseWeaver Design

To seamlessly integrate Weaver into the GPU pipeline, we design the execution workflow of SparseWeaver with Weaver. This chapter discusses the design decisions to incorporate Weaver into the GPU execution flow.

SparseWeaver Input To achieve the goal of converting sparse operations into dense operations, Weaver must have access to shared data such as vertex IDs, locations, and degrees. Because the storage format inherently includes information about a vertex's incoming and outgoing edges, SparseWeaver gathers this data during the registration step for the specified workload. Whether examining incoming edges (pull direction) or outgoing edges (push direction), shared data — such as the base vertex ID, the start location of the neighbor edge list, and the degree — can be collected [18].

SparseWeaver registers this shared data into the ST during the registration stage for use in the distribution stage. Since GPU cores handle graph topology access and the data is stored in registers, SparseWeaver requires an ISA extension to define how to extract and gather data from the destination registers of specific code sequences, ensuring efficient integration with the GPU execution pipeline.

> †**Design Decision**: Gather workload information from the GPU core: Base vertex ID, Location, and Degree.

SparseWeaver Output SparseWeaver generates Work IDs for each thread within a warp simultaneously in response to every decode request. To transfer these Work IDs to the GPU core, SparseWeaver must return Edge IDs via a newly introduced ISA. Additionally, SparseWeaver returns the base vertex

ID, which identifies one vertex in the edge vertex pair, enabling fast edge data access [18]. This process also requires delivering Work IDs to the core to facilitate the execution of subsequent instructions, necessitating another ISA extension to transmit these IDs from SparseWeaver to the GPU core.

Furthermore, SparseWeaver provides a thread mask to guide thread activation. By default, SparseWeaver activates all threads within the core to execute the distribution stage. However, during mapping, SparseWeaver may create a scenario where some threads are left without assigned work. For example, if the total degree is not evenly divisible by the number of threads in a warp, some threads may remain idle. In such situations, thread divergence can occur, potentially leading to performance degradation and requiring divergence control mechanisms like split and join [63]. To address this, SparseWeaver returns a thread mask that serves as a hardware-based mechanism to control active threads, effectively replacing the need for additional divergence control logic.

†**Design Decision**: Return workload indicator to GPU core: EID and VID. Return a clue for thread activation.

Out of Order Registration and Ordered Scan SparseWeaver relies on the ST not only to store shared data but also to enable entry access in VID order. Organizing edge access by vertex ID during the gathering phase can significantly improve performance [7]. However, the out-of-order execution of warps presents a challenge, as SparseWeaver may encounter data in a disordered sequence. To address this, we implement a two-step approach to construct an ordered ST.

First, the kernel code produced by the SparseWeaver compiler integrates

investigation logic that utilizes software thread IDs (e.g., CUDA thread ID or OpenCL global ID), allowing threads to process vertices sequentially. Second, shared data is organized within the ST using warp IDs and thread IDs as keys, ensuring that entries are stored and accessed in an ordered manner.

> †**Design Decision**: Perform ordered decode. For ordered decode, use software thread ID to investigate the graph topology in the kernel code and use hardware thread ID and warp ID to index the workload information table.

Dynamic Work Distribution SparseWeaver dynamically assigns work IDs, enabling a distribution mechanism that aligns with out-of-order warp execution. Specifically, SparseWeaver maps and allocates work based on the sequence of incoming requests rather than relying on warp IDs. This dynamic approach enhances locality by keeping related edge IDs actively processed, which helps optimize warp execution efficiency.

> †**Design Decision**: Distribute the workload dynamically to maximize the benefit from graph locality.

Synchronization between Registration and Distribution SparseWeaver relies on synchronization for three key reasons. First, synchronization ensures that all warps have completed registering the data they are responsible for processing. This step is crucial because the ST can only be finalized with workload information from all warps within the core, enabling an ordered scan. Second, the need for registration is determined by the graph topology, causing the number of vertices processed by warps in each iteration to vary. As a result, Weaver cannot predict the number of warp requests, necessitat-

ing explicit synchronization points within the GPU code to manage these variations. Third, synchronization is essential to avoid premature exits. For instance, if one warp completes the registration and distribution stages before another warp has finished registering, it risks underutilizing warp resources, preventing the efficient distribution of the remaining workload. To address these challenges, we introduce synchronization between the registration and distribution steps. This process is also necessary for other complex schedule algorithms, requiring just one synchronization per iteration. It is worth noting that software-based schemes, except for native ones, also include at least one synchronization point.

†**Design Decision**: Insert synchronization between the registration and distribution stages in the GPU code.

Filtering workload Filtering plays a critical role in improving performance for certain graph applications [6]. As such, the work IDs distributed by SparseWeaver must be filtered whenever possible. Specifically, filtered work IDs should be returned if filtering based on the base vertex ID is feasible. However, since the primary objective of SparseWeaver is to distribute work IDs efficiently, it avoids introducing additional structures or logic for tasks that can be handled within the GPU pipeline.

To achieve this, the SparseWeaver compiler inserts filters into the registration stage when they relate to the base vertex ID. It also inserts code to set the degree to zero for any filtered vertices, ensuring that no corresponding work IDs are generated during the distribution stage.

Moreover, some algorithms, such as BFS, do not require processing all neighbors during gather operations once the necessary information has been

obtained. In these scenarios, an early exit from the distribution stage for a specific vertex ID may be required. This is particularly relevant for supernodes with a large number of neighbors, where decoding might need to terminate midway. To support this functionality, SparseWeaver requires an ISA extension that allows Weaver to skip decoding for specific vertex IDs and exclude them from further processing.

†**Design Decision**: Integrate a filter into the GPU code at the appropriate location and send a skip signal when no further distribution is needed for a specific vertex.

4.4.4 Assembling Design Decisions

Figure 4.7 presents the complete workflow of SparseWeaver, illustrating how warps execute over time and how tasks are offloaded to SparseWeaver. Within the GPU core, execution follows the sequence of registration, synchronization, and distribution.

During the registration stage, the GPU core gathers shared data such as vertex IDs, locations, and degrees to deliver this information to Weaver. Assuming out-of-order warp execution, Weaver stores shared data in the Sparse Work Information Table, indexed by warp ID and thread ID. For instance, as depicted in Figure 4.7, three warps execute in the order Warp 0, 2, 1, with the shared data from Warp 2 being stored in entries 8, 9, 10, and 11 of the ST. If a vertex is filtered, the corresponding thread sets its degree to 0 during the investigation process. Upon completing the registration stage, all active warps wait for synchronization to ensure all warps have registered their data.

In the distribution stage, the GPU core issues decode requests to retrieve

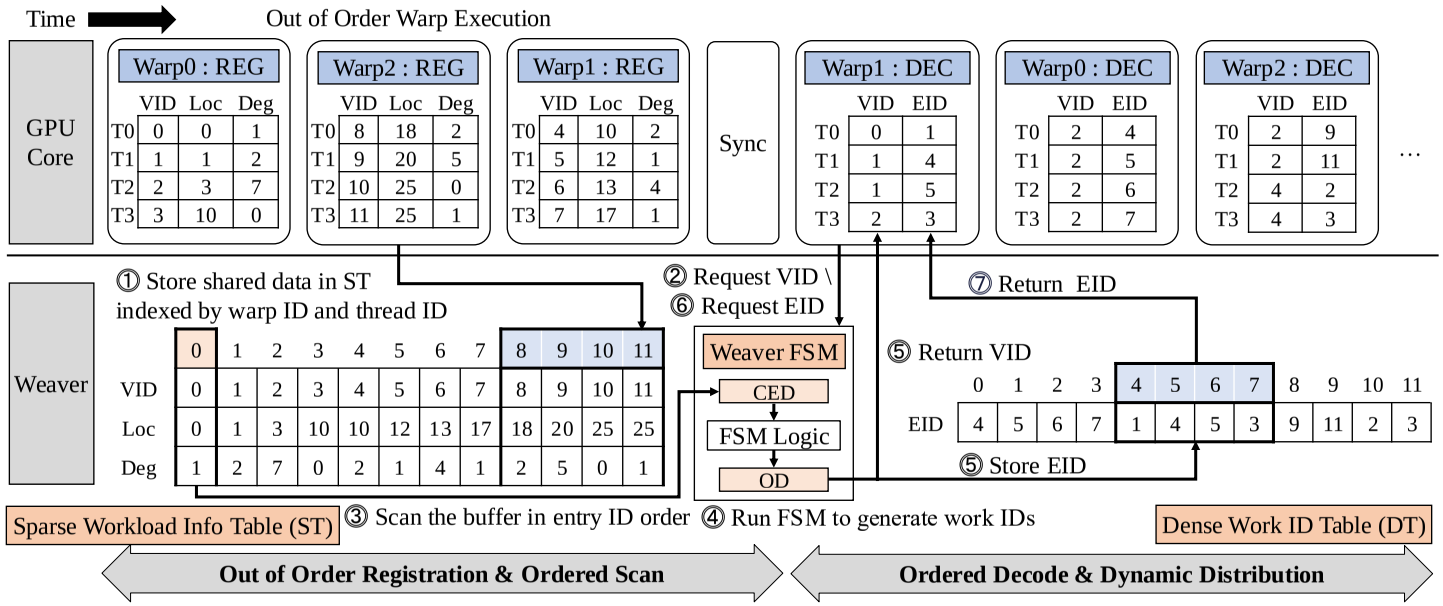


Figure 4.7: SparseWeaver execution flow. SparseWeaver performs out-of-order registration and ordered scan, then performs ordered decode and dynamic distribution when receiving requests from the GPU core.

work IDs generated by Weaver. As shown in Figure 4.7, Warp 1 sends the first decode request for VID to Weaver. Following the FSM described in Figure 4.6, Weaver processes ST entries such as (0, 2, 1), (2, 10, 2), and (4, 30, 5) to generate the OD buffer with (0, 2), (2, 10), (2, 11), and (4, 30). Weaver then returns VIDs (0, 2, 2, 4) to the GPU core and stores the corresponding EIDs (2, 10, 11, 30) in the Dense Work ID table. When Warp 1 sends a decode request for edge IDs, Weaver provides EIDs (2, 10, 11, 30). Using these edge IDs, warps can filter the opposite vertex IDs by accessing them.

By assembling the design decisions detailed in Chapter 4.4.3, SparseWeaver effectively integrates Weaver into the GPU execution pipeline, ensuring seamless operation within the overall workflow.

4.5 SparseWeaver Framework

Figure 4.8 provides a system overview of SparseWeaver. The SparseWeaver system accepts input User Defined Functions (UDFs) for algorithms and graphs using a storage format, a storage format interface, and a specified direction.

The UDFs comprise four distinct methods: *init*, *gather*, *apply*, and *filter*. Users decompose a graph algorithm into these methods, similar to other graph processing frameworks [18]. Since the input graph is stored in a specific storage format, such as Compressed Sparse Row (CSR), the user leverages the storage format interface to allow SparseWeaver to access the storage format. The storage format interface provides two methods, *getNeighbor* and *getEdge*, to retrieve graph topology and edge information [18]. The direction indicates whether the gathering process focuses on incoming or outgoing edges.

The SparseWeaver frontend compiler processes the graph algorithm and generates GPU kernel code, while the backend compiler performs target-specific optimizations and produces the GPU binary. The compiled binary is executed on the SparseWeaver GPU with Weaver support. To implement Weaver, we use the open-source RISC-V GPU, Vortex [63, 64, 65], in conjunction with PoCL [66] and LLVM [67] as the frontend and backend compilers, respectively.

4.5.1 SparseWeaver Instruction

Table 4.2 outlines the SparseWeaver instructions, which act as the interface for exchanging inputs and outputs with Weaver. The SparseWeaver system includes a WEAVER_REG instruction to register the base vertex ID, start edge ID, and degree. Additionally, it provides two output instructions to retrieve data from the SparseWeaver microarchitecture: WEAVER_DEC_ID, which returns the vertex ID, and WEAVER_DEC_LOC, which returns the edge ID. The system also incorporates the WEA-VER_SKIP instruction, allowing specific vertices to be skipped during decoding.

Our implementation is based on a RISC-V open-source GPU, so the instructions are defined in the RISC-V format. According to the RISC-V manual [68], WEAVER_DEC_ID and WEAVER_DEC_LOC are implemented as R-type instructions [68], structured with opcode, rd, funct3, rs1, rs2, and func7. Meanwhile, WEAVER_REG is implemented as a CUSTOM instruction, formatted with opcode, rd, funct3, rs1, rs2, funct2, and r3. We utilize funct3 and funct2 to differentiate between these instructions.

An essential aspect is that one of the output instructions must signal the termination of the work distribution loop. While the thread mask can deter-

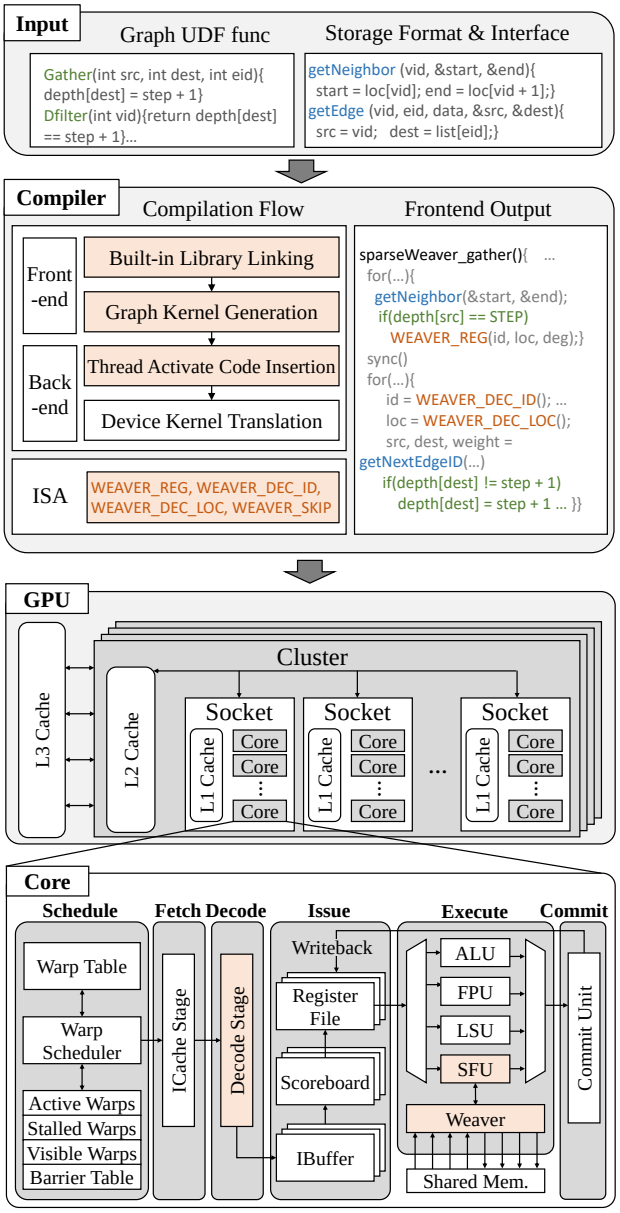


Figure 4.8: SparseWeaver system overview. SparseWeaver consists of the compiler and GPU. The compiler generates programs using the ISA extension. The GPU core decodes and executes instructions by extending the decoder and the SFU.

Table 4.2: SparseWeaver instructions

Instruction	IType	Opcode	funct	Description
WEAVER_REG, VID, loc, deg	C	CUSTOM1	1	Register VID, loc, deg
WEAVER_DEC_ID, VID	R	CUSTOM0	7	Return VID of next workload
WEAVER_DEC_LOC, EID	R	CUSTOM0	8	Return EID of next workload
WEAVER_SKIP, VID	C	CUSTOM1	2	Send skip signal using VID

mine whether tasks proceed, it cannot explicitly indicate the exit status, as deactivating all threads could result in an unintended termination state. To address this, the WEAVER_DEC_ID instruction returns -1 to signify the end of the loop. When all active threads return -1, SparseWeaver concludes the distribution stage.

4.5.2 SparseWeaver Compiler and Runtime

The compilation process is divided into distinct frontend and backend stages by leveraging the updated PoCL and LLVM to incorporate the Weaver ISA and device-specific runtime libraries. Each stage performs essential optimizations to adapt the kernel for the target architecture effectively.

The SparseWeaver frontend compiler takes user inputs and produces graph processing kernels, as illustrated in Figure 4.9. The frontend compiler applies two key optimizations. First, it performs **Built-in Library Linking**, integrating functions such as atomic operations or math utilities tailored to the specific hardware target. Second, it carries out **Graph Kernel Generation**, combining Weaver ISA intrinsics, user-defined functions, and the storage format interface to construct the kernel.

Figure 4.9 provides an example of a pull-direction kernel generated by the frontend compiler. This kernel traverses incoming edges based on the

```
1 void SparseWeaverGatherKernel(wset, graph) {
2     tid = get_thread_id() // Get the TID
3     // Registration step
4     for (id = wset.base_id + tid; id < wset.bound; id += wset.stride) {
5         vid = getFrontier(id)
6         if (dest_filter(vid))
7             continue
8         start, end = getNeighbor(graph, ...)
9         WEAVER_REG(vid, start, end - start)}
10    synchronization()
11    while (true) { // Distribution step
12        vid = WEAVER_DEC_ID()
13        if (vid == -1)
14            break
15        eid = WEAVER_DEC_LOC()
16        src, dest, weight = getEdge(eid...)
17        if (dest_filter(dest))
18            WEAVER_SKIP(dest)
19        if (src_filter(src))
20            continue
21        computeFn(loc, src, dest, weight)
22    }
23 }
```

Figure 4.9: Generated SparseWeaver gather kernel (Pull)

destination vertex ID for gather processing. It begins with the registration stage (lines 4 to 9) followed by the distribution stage (lines 11 to 22). In the registration stage, the kernel uses the WEAVER_REG intrinsic to pass shared data to Weaver (line 9). During the distribution stage, the WEAVER_DEC_ID and WEAVER_DEC_LOC intrinsics decode work IDs (line 12 and line 15). Depending on the specified direction, the compiler inserts filters at appropriate stages. For pull-direction processing, it adds a destination filter in the registration stage, incorporating the WEAVER_SKIP intrinsic (line 18).

The SparseWeaver backend compiler processes the intermediate representation (IR) generated by the frontend compiler and performs **Thread Activate Code Insertion** before translating it into GPU binary through device-

specific kernel translation. This stage introduces target-specific optimizations, such as thread mask control logic, to ensure all threads within a warp are activated during the distribution phase and subsequent gather steps.

For instance, on the Vortex GPU, the backend compiler inserts code to store the warp's thread mask and activates all threads before entering the distribution loop (line 11 in Figure 4.9). After completing the distribution loop, it restores the thread mask. Additionally, the backend compiler expands the **ISA Table**, incorporating the instructions in Table 4.2, to support kernel translation. With these compiler enhancements, SparseWeaver generates the GPU code required for efficient graph processing.

4.5.3 Weaver Implementation

SparseWeaver is implemented on the Vortex GPU by extending the Special Function Unit (SFU) to include Weaver, as depicted in Figure 4.8. When the GPU core decodes and executes Weaver instructions, it can issue either a register request or a decode request to Weaver.

The Sparse Work Information Table and Dense Work ID Table in Weaver can be realized using registers or shared memory. We opted to implement these tables in shared memory for several reasons. The Vortex GPU has a constrained register count but provides fast access to shared memory. Furthermore, SparseWeaver requires relatively few table accesses—only one read and write per graph topology data—compared to the large number of edges being processed. As a result, table access overhead is minimal and can be effectively hidden by the GPU pipeline's execution. Further analysis of this implementation is provided in Chapter 4.6.4.

Graph Name	Number of Nodes	Number of Edges
bio-human-gene1 (BH)	22,284	24,691,926
bio-mouse-gene (BM)	45,102	29,012,392
roadNet-CA (RN)	1,971,282	553,321
road-central (RC)	14,081,817	3,386,682
web-uk-2005 (UK)	129,633	23,488,098
graph500-scale19 (G500)	335,319	15,459,350
COLLAB (CO)	372,475	49,144,316
hollywood-2011 (HW)	2,180,653	228,985,632
web-wikipedia (WK)	2,936,414	104,673,033

Table 4.3: Graph dataset information [31]

4.6 Evaluation of SparseWeaver

We evaluate the performance of SparseWeaver using four graph algorithm operators and GCN on seven datasets, comparing it against four software schedules (VM, EM, WM, CM [7]) and an edge-generating hardware approach (similar to hardware-based schemes [54, 55, 56]) on the open-source RISC-V GPU, Vortex [63, 64, 65].

SparseWeaver is modeled on the Vortex GPU, which provides an open-source software stack, a cycle-level simulator, and an RTL hardware description. The simulator, Simx, achieves cycle-level accuracy within 6% of the RTL model [64], enabling accurate performance testing for SparseWeaver. Additionally, LLVM [67] and PoCL [66] were extended to support the Weaver ISA and include compiler optimizations for Vortex [69].

The evaluation is conducted using Vortex hardware configurations with two sockets, three cores per socket, 32 warps per core, and 32 threads per warp. L1 and L2 cache sizes are configured as 64KB and 1MB, respectively. To account for SparseWeaver's hardware overhead, the L1 cache size is reduced

to 32KB to accommodate 512 entries in the Sparse Work Information Table and the Dense Work ID Table per core.

Hardware overhead is analyzed by extending the Vortex RTL and synthesizing it using Quartus Prime Pro 8.1, targeting the Intel Stratix 10 FPGA. The evaluation utilizes seven datasets listed in Table 4.3 and examines four algorithms: PageRank (PR)[33], Connected Components (CC)[34], Breadth-First Search (BFS)[70], and Single Source Shortest Path (SSSP)[70].

4.6.1 Comparison with Software-based Schemes

Figure 4.10 shows that SparseWeaver improves performance for most algorithms. SparseWeaver achieves 2.36x speedups for the vertex mapping and 2.63x speedups for the edge mapping. SparseWeaver achieves 2.73x, 2.64x, 2.71x, and 1.60x speedups over the VM for BFS, SSSP, PR, and CC.

In detail, SparseWeaver surpasses all other software schedules across the four evaluated benchmarks. Its performance is particularly notable in the BFS and SSSP benchmarks, which involve destination and source filters that exacerbate load imbalances among vertices. These imbalances allow SparseWeaver to achieve significant speedups compared to other schedules. BFS exhibits a more significant speedup than SSSP due to the absence of edge weight processing in BFS.

For PR and CC, which process all edges during the gather step, the balanced workload provided by SparseWeaver enhances performance. This workload balance also enables coalesced memory access during edge information retrieval in the PR algorithm, a feature that the VM fails to exploit effectively. Consequently, all other schedules demonstrate improved performance over

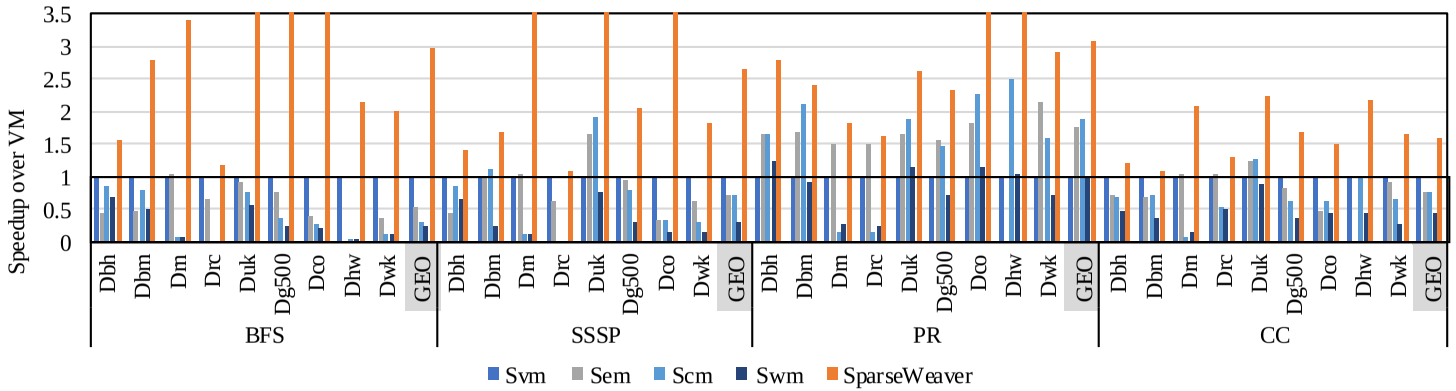
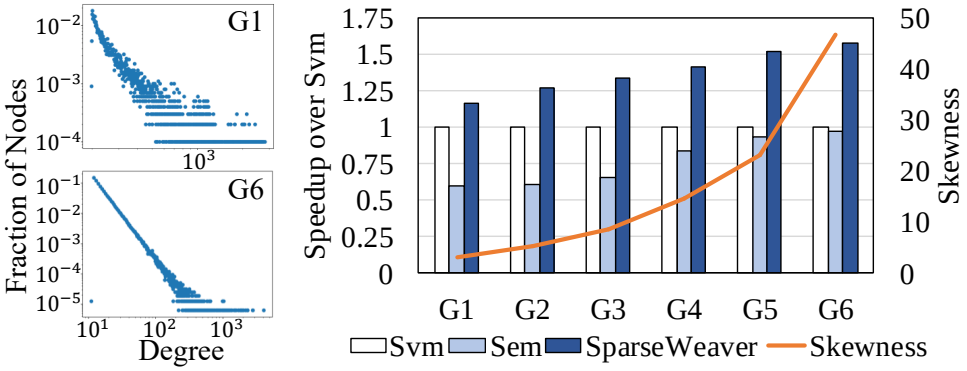


Figure 4.10: Performance comparison with the four software schedules (VM, EM, WM, CM) and SparseWeaver. Each graph shows speedups over VM. Four graph algorithms on seven graphs are used on Vortex GPU



(a) Fraction of Nodes (b) Speedup over VM

Figure 4.11: Skewness Sensitivity (a) shows the graph degree distribution of G1 (low skewness) and G6 (high skewness) (b) shows speedups over VM when increasing skewness

VM when running PR.

4.6.2 Skewness Sensitivity

To evaluate the sensitivity of each schedule to skewness [71] in graph data, we compared SparseWeaver with VM and EM schedules using the PageRank algorithm. Graph datasets were generated using the NetworkX Power-law graph generator, maintaining a fixed number of edges (1.9M) while varying the number of vertices (10k, 12k, 16k, 20k, 40k, 80k).

Figure 4.11a illustrates the degree distribution of the G1 and G6 graphs. G1, characterized by a smaller number of vertices and lower skewness, exhibits a narrow degree distribution and a short edge fraction tail. Conversely, G6, with a larger vertex count and higher skewness, displays a broader degree distribution and an extended edge fraction tail. The skewness increases as the graph index progresses from G1 to G6, resulting in progressively greater

workload imbalances.

The impact of skewness is evident when comparing EM and VM performances in Figure 4.11b. As workload imbalance worsens, their performances converge, even though EM incurs double, the memory reads for edges compared to VM. In contrast, SparseWeaver demonstrates a performance trend similar to EM, indicating that its workload balancing effectively mitigates the impact of data skewness.

4.6.3 Effect of Memory Configuration

Figure 4.12 shows the relationship between cycle count and the memory ratio to GPU core frequency, ranging from 1 to 6, using VM, EM, and SparseWeaver with the PageRank algorithm. Here, n represents the GPU core frequency, which is n times higher than the DRAM frequency. This trend highlights that graph processing is a memory-intensive workload. Across all frequency ratios, Sparse-Weaver outperforms both VM and EM. By effectively mitigating workload imbalance and reducing memory access, SparseWeaver achieves a lower total cycle count, leading to better performance.

4.6.4 Effect of Work Table Access

Figure 4.15 illustrates the impact of shared memory access on the Sparse Workload Information Table and Dense Work ID Table, with shared memory read overheads set to 10, 20, 40, 80, and 160 cycles. The results show that SparseWeaver effectively conceals shared memory read latency through the GPU pipeline. Tests were conducted on a Vortex configuration with eight cores, 32 warps, and 32 threads using the PageRank algorithm. Since only a

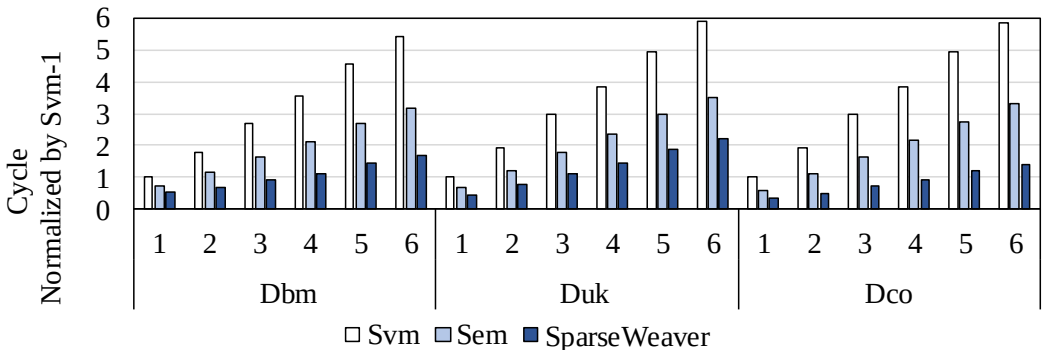


Figure 4.12: Execution cycle comparison by increasing memory and GPU cycle ratio normalized by VM - 1 ratio. The number n means GPU has n times higher frequency compared to DRAM frequency (n GHz GPU vs 1 GHz DRAM)

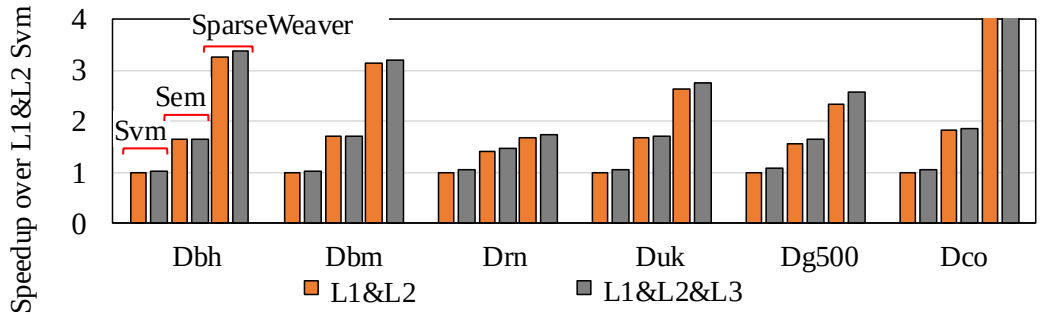


Figure 4.13: Performance comparison with L1&L2 and L1&L2&L3 cache with PageRank. Each graph shows speedups over VM with L1&L2 cache.

single scan is required per table entry, performance remained stable even as read latency increased, demonstrating that other instructions in the pipeline successfully mask the effects of cache read delays.

4.6.5 Cache and Memory Analysis

Figure 4.13 compares performance between two configurations: L1 & L2 cache versus L1 & L2 & L3 cache. The results indicate that the presence of an L2 cache significantly affects performance, while adding an L3 cache provides

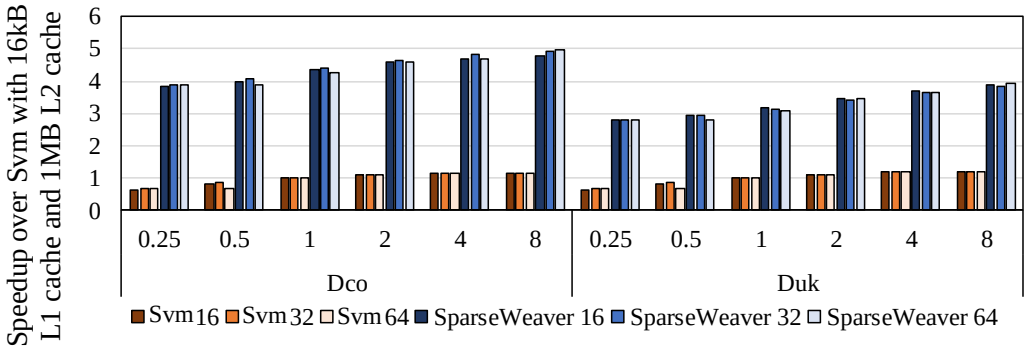


Figure 4.14: Performance comparison by L1 cache size (16KB, 32KB, 64KB) and L2 cache size (0.25MB, 0.5MB, 1MB, 2MB, 4MB, 8MB) using PageRank and two datasets. Each graph shows speedups over VM with 16KB L1 and 1MB L2 cache.

no noticeable improvement when comparing the L1 & L2 configuration with L1 & L2 & L3. Figure 4.14 further examines performance by varying the cache sizes: L1 from 16KB, 32KB, to 64KB, and L2 from 0.25MB, 0.5MB, 1MB, 2MB, 4MB, to 8MB. The results show that increasing cache size has minimal impact on performance, suggesting limited sensitivity to cache size in this context.

4.6.6 Hardware Overhead

To evaluate the area overhead of the hardware implementation, SparseWeaver was modeled in RTL for the Vortex GPU. The RTL was synthesized using Quartus Prime Pro 18.1, targeting the Intel Stratix 10 FPGA. For a single core, the logic for the Workload Info Table and Work ID Table (Figure 4.7) resulted in an additional 678 dedicated logic registers. Meanwhile, implementing the SparseWeaver FSM and associated logic to support the new instructions required an increase of 3109 adaptive logic modules (ALMs).

The per-core overhead incurred by SparseWeaver is minimal, with a 0.045

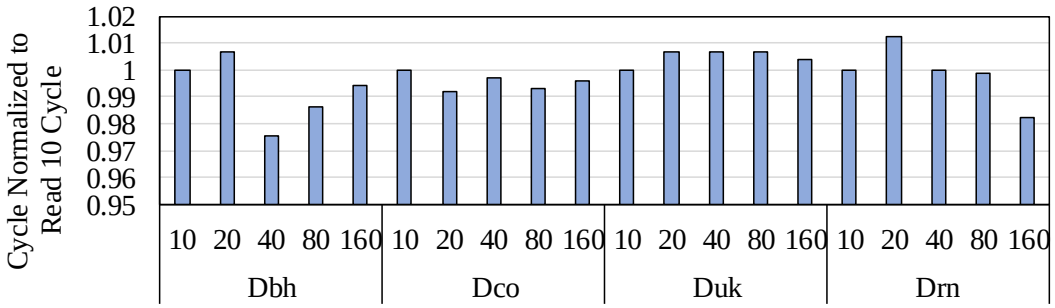


Figure 4.15: Execution cycle by changing cache read overhead from 10, 20, 40, 80, 160 cycles.

% increase in dedicated logic registers and a 2.96% increase in ALMs due to the Sparse Workload Info Table, Dense Work ID Table, and FSM. For a 16-core configuration, the ALM usage increases by only 2.01% compared to the default 16-core setup.

This reflects a negligible area overhead in hardware, as SparseWeaver does not increase the utilization of block memory, RAM blocks, or DSP blocks. A detailed summary of the FPGA resource usage is presented in Table 4.4, and Figure 4.16 visually illustrates the area increase for both single-core and 16-core GPUs.

In terms of developmental overhead, the hardware implementation required 251 additional lines of System Verilog code. Compared to the original codebase of 184,449 lines, this represents a mere 0.136%

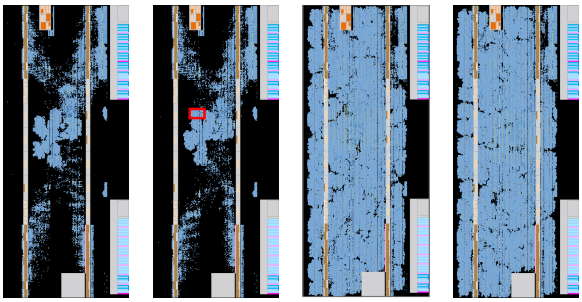
4.6.7 Push and Pull Breakdown

Figure 4.17 presents the performance breakdown for the Push and Pull directions.

During the registration phase, the cycle counts for Push and Pull are nearly

Vortex GPU configuration	Total ALMs	ALM % increase	Block memory % increase	RAM % increase	DSP % increase
1-core default	105,094	2.96%	0%	0%	0%
1-core w/ SparseWeaver	108,203				
16-core default	580,332	2.01%	0%	0%	0%
16-core w/ SparseWeaver	591,971				

Table 4.4: The area overhead of SparseWeaver in terms of FPGA resources utilized.



(a) (b) (c) (d)

Figure 4.16: Block utilization diagrams for different configurations of the Vortex GPU synthesized on the Stratix 10 FPGA where blue indicates a used block, and the red box indicates a major difference in the utilized blocks. (a) A single core of the GPU (b) A single core of the GPU with SparseWeaver (c) A 16-core GPU (d) A 16-core GPU with SparseWeaver

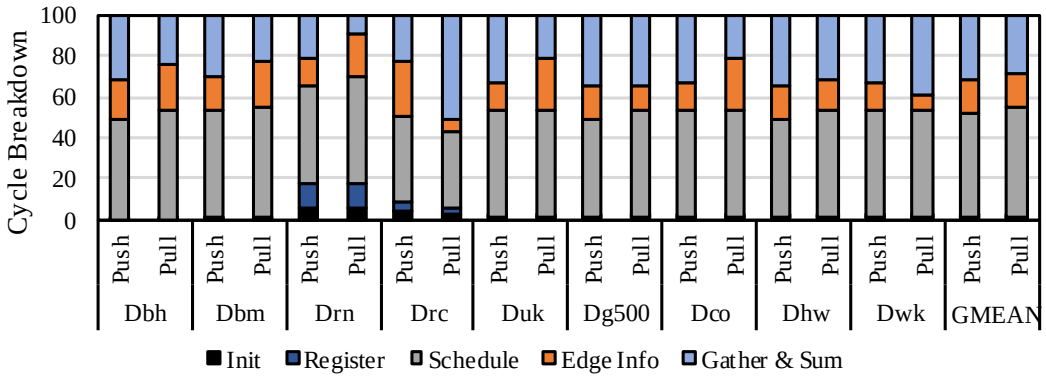


Figure 4.17: Execution cycle breakdown of the gather process with Push and Pull using PageRank. The breakdown includes five steps: Init, Registration, Work ID calculation, Edge information access, and Gather & Sum.

identical, with less than a 1% difference. Similarly, the summation cycles for Edge scheduling and Edge information access are comparable in both directions, as a symmetric graph dataset is used. However, the gather and sum cycles vary depending on the dataset. For instance, Push requires fewer cycles for datasets like D_{wk} and RC, whereas Pull demonstrates better performance for other datasets.

4.6.8 Case Study 1: Existing Hardware-based Scheme

In this case study, we compare SparseWeaver to edge-generating hardware (EGHW) mode, which is similar to existing hardware-based schemes [54, 55, 56]. In EGHW, all operations within the edge schedule (depicted in the blue box of Figure 4.5) are handled by hardware, excluding vertex filtering. This includes examining the graph topology of vertices and accessing edge information.

In EGHW mode, the GPU writes vertex IDs to a buffer in shared mem-

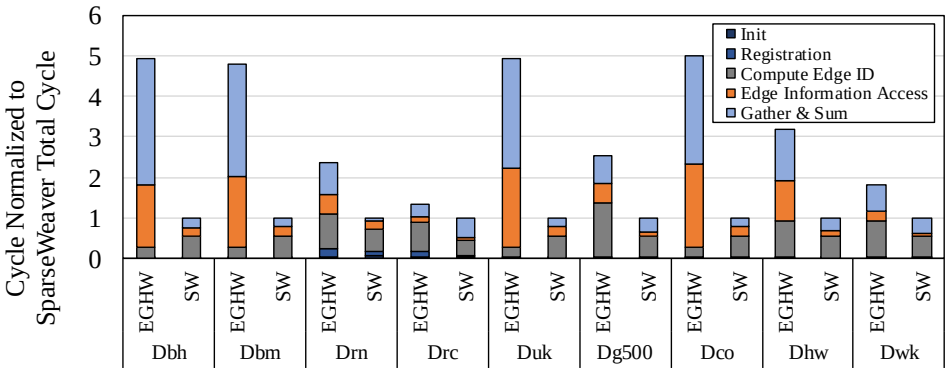


Figure 4.18: The execution time breakdown for the gathering process is compared between EGHW and SparseWeaver using the PageRank algorithm across seven graph datasets. The breakdown consists of five steps: Init, Registration, Work ID Calculation, Edge Information Access, and Gather. Execution times are normalized relative to the cycle count of the Init step in SparseWeaver.

ory, and EGHW accesses graph topology data by reading these IDs. EGHW then performs remapping and writes edge data, such as the opposite vertex ID and weight, back into the buffer. Consequently, the GPU must wait for edge information from EGHW before proceeding with gather and sum operations. By offloading graph topology and edge information access to Weaver, SparseWeaver enables hardware-level generation of edge information.

Figure 4.18 presents a performance comparison between EGHW and Weaver. SparseWeaver achieves a geometric mean speedup of 3.64x compared to EGHW. This improvement primarily stems from the distribution stage, including Work ID Calculation, Edge Information Access, and Gathering.

The performance disparity arises because EGHW does not effectively hide the overhead of memory reads for edge information and requires additional shared memory access to store and retrieve edge data from the buffer. Fur-

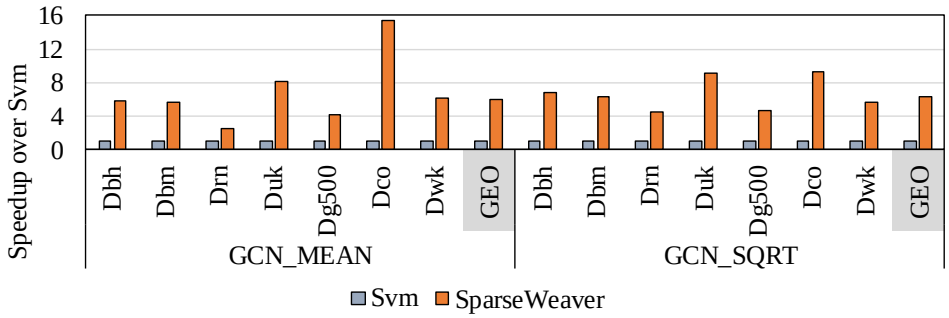


Figure 4.19: Performance of GCN operators with three different schedules, VM parallelized by weight, EGHW and SparseWeaver

thermore, stalls in Edge Information Access affect the Gather step due to dependency on edge data loads. Although adding pipelines and more advanced logic could improve Edge Information Access in EGHW, SparseWeaver achieves efficient memory access by leveraging the GPU pipeline in its design.

4.6.9 Case Study 2: Performance with GCN

This case study highlights the extensibility of SparseWeaver by comparing its performance in executing the Graph Convolution Network (GCN) operator [72] under vertex and weight parallelization strategies. The GCN experiment evaluates three kernels: initialization, sparse matrix-matrix multiplication (SpMM), and mean aggregation (GraphSum) across 16 weight dimension sizes. As a baseline, we modify the VM mapping to first parallelize the weight dimension and then the vertex dimension for executing the SpMM and GraphSum kernels. In this configuration, each thread gathers specific weights across the neighbor list of a vertex, eliminating the need for atomic operations during weight updates. Conversely, our approach continues to parallelize edge updates, iterating over the weight dimension while using atomic operations.

Figure 4.19 shows that SparseWeaver achieves a 6.15-fold speedup compared to VM. While SpMM benefits from weight parallelization and reduced reliance on atomic operations, resulting in improved performance with VM, SparseWeaver outperforms VM in the GraphSum kernel. This is due to its ability to lower the cost of coefficient calculations, which depend on the degrees of source and destination vertices. Since GraphSum requires more computation time overall, SparseWeaver demonstrates superior performance compared to VM.

4.6.10 Case Study 3: Comparison with GRAssembler

	GRAssembler with A30 (108 SM, 64 warps/core, 1200Mhz)				**SparseWeaver (8 cores, 32 warps/core)**		
	Tuning Time (sec)	**VM (ms)**	**Best (ms)**	**Speedup**	**VM (ms)**	**SW (ms)**	**Speedup**
HW	4502.83	18.92	8.78	2.16	5408.33	1016.67	5.32
UK	1446.57	1.45	0.69	2.11	400.83	173.33	2.61
CO	1710.66	2.63	1.40	1.88	916.67	216.67	4.23
RN	1139.41	0.47	0.32	1.47	453.01	247.68	1.83

Table 4.5: Speedup over VM Comparison using GRAssembler and SparseWeaver with PageRank

Table 4.5 compares the performance improvements of the existing Autotuner [18] and SparseWeaver relative to VM. Although the hardware configurations, including the number of parallel units, differ between the Nvidia GPU tested with the Autotuner and the SparseWeaver setup, the results indicate that SparseWeaver achieves better performance compared to VM. Notably, this is accomplished without the additional tuning time required by the Autotuner.

4.7 Discussion

4.7.1 Integrating into GRAssembler

Even though this research shows the effectiveness of GRAssembler using three different GPUs (NVIDIA RTX 3090, 4090, and A100), the GPU architectures, drivers, and even programming languages are evolving rapidly. As shown in Chapter 3, this research proposes a new auto-tuner to optimize graph processing for anonymous input graphs and algorithms on a GPU by decoupling schedules and storage formats. While our solution can find optimal configurations through auto-tuning, even as GPU architectures evolve, supporting new hardware features requires extensions. For instance, hardware features such as sparse cores and tensor cores are examples. Therefore, as future work, the auto-tuner could be extended by incorporating new options, such as enabling hardware-specific features.

Integrating SparseWeaver into GRAssembler or other auto-tuners could be achieved by extending hardware options. SparseWeaver represents an extension of GPUs through new microarchitectural features and often demonstrates greater effectiveness compared to software-based schedules for diverse graphs. Consequently, if the auto-tuner can be executed on GPUs supporting SparseWeaver, it could enable the hardware option SparseWeaver, potentially eliminating the need to explore software schedules while retaining the use of software schedules for GPUs that do not support SparseWeaver.

4.7.2 General Usage of SparseWeaver

We believe that SparseWeaver can extend its applicability to other sparse applications, particularly those originally using the CSR format, such as GPU hashing, MapReduce, Graph Neural Networks, or sparse matrix multiplication. These applications handle sparse data, such as hash tables containing sparse workload information. For example, Algorithm 4.1 shows a possible implementation of GPU hash lookup. The sparse workload information can store the position of each key-value pair within hash table buckets [73]. SparseWeaver can replace the second for loop to distribute hash operations across multiple threads, as the offset array contains workload information.

Algorithm 4.1: The GPU hash lookup [73]

```
   Input: keys : Input Hash Keys
          offset: offset array pointing to the ranges of bucket
1  foreach key ∈ keys do
2  |   bucket ← hash(key)
3  |   for i ∈ range(offset[bucket], offset[bucket+1]) do
4  |   |   if hashtable[i] == key then
5  |   |   |   ...
6  |   end
7  end
```

4.8 Summary

This research proposes a new collaborative hardware and software graph processing framework SparseWeaver. SparseWeaver effectively addresses the workload imbalance in the graph processing on GPU by converting the sparse operations into dense operations. Based on the analysis of common patterns in software schemes, we propose a hardware logic, called Weaver, new lightw-

eight hardware that is tightly integrated into the GPU pipeline with simple ISA. With only 0.045% additional dedicated logic registers and 2.96% additional ALMs in a single core, SparseWeaver achieves a performance speedup that is 2.36 times faster on real-world graph datasets compared with the software scheme.

CHAPTER 5

CONCLUSION

Executing an algorithm domain on a hardware domain encounters various optimization opportunities. Focusing on which component among fundamental components, such as hardware, algorithms, and memory, can offer multiple optimization points of view and opportunities. When different optimizations target different aspects, understanding the relationships between these optimizations and creating synergies can unlock new levels of performance enhancement. However, attempting to blindly implement all possible combinations of optimizations would undoubtedly result in a significant implementation burden. Therefore, it is evident that abstraction methods are required to explore optimization combinations efficiently across fundamental components.

This research aims to open up a new area of optimization of graph processing on GPU by decoupling the three key components, such as algorithm, schedule, and storage format. Through this approach, the research enhances the coverage, composability, extendability, and modularity of graph processing on GPUs. We begin by analyzing the characteristics of the existing optimizations, presenting a fundamental abstraction interface for each key component. This research proposes GRAssembler, a new GPU graph processing framework that efficiently integrates the decoupled schedule, storage format,

and algorithm without abstraction overhead.

Furthermore, by deeply exploring storage format and schedule during the abstraction process, this research uncovers new workload balancing optimization opportunities for storage format and schedule and expands the tuning space. In terms of storage format, this study identifies that no existing storage format provides fine-grained workload balancing with low memory usage while maintaining high performance. To address this, the study proposes CR^2, a new storage format composed of community-aware and degree-ordered subgraphs, optimized for GPU and highly skewed real-world graph characteristics. Regarding schedule, the study observes that existing schedules incur large overhead for workload rebalancing due to frequent access to shared memory and synchronization methods. To mitigate this, the research introduces SparseWeaver, a microarchitecture that converts sparse operations into dense operations to accelerate schedule process, ensuring balanced workloads across GPU threads with low area overhead.

References

[1] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua, "Graph processing on gpus: A survey," Jan. 2018.

[2] X. Chen and L. Pan, "A survey of graph cuts/graph search based medical image segmentation," *IEEE Reviews in Biomedical Engineering*, pp. 112–124, 2018.

[3] P. Boldi and S. Vigna, "The webgraph framework ii: Codes for the world-wide web," in *Data Compression Conference*, 2004, pp. 528–.

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, pp. 107–117, 1998.

[5] A. t. Balaban, "Applications of graph theory in chemistry," *Journal of chemical information and computer sciences*, 1985.

[6] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[7] K. Meng, J. Li, G. Tan, and N. Sun, “A pattern based algorithmic autotuner for graph processing on gpus,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 201–213.

[8] A. Brahmakshatriya, Y. Zhang, C. Hong, S. Kamil, J. Shun, and S. Amarasinghe, “Compiling graph applications for gpu s with graphit,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2021, pp. 248–261.

[9] S. Pai and K. Pingali, “A compiler for throughput optimization of graph algorithms on gpus,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 1–19.

[10] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, “To push or to pull: On reducing communication and synchronization in graph computations,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, 2017, pp. 93–104.

[11] H. Wang, L. Geng, R. Lee, K. Hou, Y. Zhang, and X. Zhang, “Sep-graph: Finding shortest execution paths for graph processing under a hybrid framework on gpu,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 38–52.

[12] H. Liu and H. H. Huang, “Simd-x: Programming and processing of graph algorithms on gpus,” in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, 2019, pp. 411–427.

[13] D. Merrill, M. Garland, and A. Grimshaw, “Scalable gpu graph traversal,” in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012, pp. 117–128.

[14] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, “Work-efficient parallel gpu methods for single-source shortest paths,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 349–359.

[15] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, “Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics,” in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 752–768.

[16] Z. Fu, M. Personick, and B. Thompson, “Mapgraph: A high level api for fast development of high performance graph analytics on gpus,” in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, 2014, pp. 1–6.

[17] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “Cusha: Vertex-centric graph processing on gpus,” in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, 2014, pp. 239–252.

[18] S. Jeong, Y. Lee, J. Lee, H. Choi, S. Song, J. Lee, Y. Kim, and H. Kim, “Decoupling schedule, topology layout, and algorithm to easily enlarge the tuning space of gpu graph processing,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2023, pp. 198–210.

[19] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, “Tigr: Transforming irregular graphs for gpu-friendly graph processing,” in 2018, pp. 622–636.

[20] S. Jeong, S. Cho, Y. Lee, H. Park, S. Heo, G. Kim, Y. Kim, and H. Kim, “Cr2: Community-aware compressed regular representation for graph processing on a gpu,” in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024.

[21] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “PowerGraph: Distributed Graph-Parallel computation on natural graphs,” in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Oct. 2012, pp. 17–30.

[22] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1999, pp. 251–262.

[23] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, “Csx: An extended compression format for spmv on shared memory systems,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 247–256.

[24] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” Nov. 2009.

[25] W. Liu and B. Vinter, “Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[26] D. Merrill and M. Garland, “Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2016.

[27] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, Dec. 2011.

[28] T. T. Dao and J. Lee, “An auto-tuner for opencl work-group size on gpus,” *IEEE Transactions on Parallel and Distributed Systems*, pp. 283–296, 2018.

[29] M. A. Hassaan, M. Burtscher, and K. Pingali, “Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 3–12.

[30] U. Meyer and P. Sanders, “Delta-stepping: A parallelizable shortest path algorithm,” *Journal of Algorithms*, pp. 114–152, Oct. 2003.

[31] R. A. Rossi and N. K. Ahmed, “The network data repository with interactive graph analytics and visualization,” in *AAAI*, 2015.

[32] C. Demetrescu, A. Goldberg, and D. Johnson. “9th dimacs implementation challenge - shortest paths.” (2009).

[33] S. Brin and L. Page, “Reprint of: The anatomy of a large-scale hypertextual web search engine,” *Computer networks*, pp. 3825–3833, 2012.

[34] J. Soman, K. Kishore, and P. Narayanan, “A fast gpu algorithm for graph connectivity,” in *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, IEEE, 2010, pp. 1–8.

[35] S. Beamer, K. Asanovic, and D. Patterson, “Direction-optimizing breadth-first search,” in *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–10.

[36] D. Li and M. Becchi, “Deploying graph algorithms on gpus: An adaptive solution,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1013–1024.

[37] *Deep Graph Library*, `https://www.dgl.ai/`.

[38] D. Zheng, M. Wang, Q. Gan, X. Song, Z. Zhang, and G. Karypis, “Scalable graph neural networks with deep graph library,” 2021, pp. 1141–1142.

[39] Y. Zhang, A. Brahmakshatriya, X. Chen, L. Dhulipala, S. Kamil, S. Amarasinghe, and J. Shun, “Optimizing ordered graph algorithms with graphit,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 158–170.

[40] *CuGraph*, `https://github.com/rapidsai/cugraph`.

[41] J. Shun and G. E. Blelloch, “Ligra: A lightweight graph processing framework for shared memory,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013, pp. 135–146.

[42] A. Raval, R. Nasre, V. Kumar, V. R, S. Vadhiyar, and K. Pingali, “Dynamic load balancing strategies for graph applications on gpus,” Nov. 2017.

[43] S. S. Hong, T. O. Kim, and K. Olukotun, “Accelerating cuda graph algorithms at maximum warp,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, 2011.

[44] A. Kyrola, G. Blelloch, and C. Guestrin, “Graphchi:large-scale graph computation on just a pc,” in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 31–46.

[45] V. Kalavri, V. Vlassov, and S. Haridi, “High-level programming abstractions for distributed graph processing,” *IEEE Transactions on Knowledge amp; Data Engineering*, 2018.

[46] M. Girvan and M. E. Newman, “Community structure in social and biological networks,” *Proceedings of the national academy of sciences*, 2002.

[47] H. Ino, M. Kudo, and A. Nakamura, “Partitioning of web graphs by community topology,” in *Proceedings of the 14th International Conference on World Wide Web*, 2005.

[48] S. Fortunato, “Community detection in graphs,” *Physics reports*, 2010.

[49] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proceedings of the 20th International Conference on World Wide Web*, 2011, pp. 587–596.

[50] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura, “Rabbit order: Just-in-time parallel reordering for fast graph analysis,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 22–31.

[51] H. Wei, J. X. Yu, C. Lu, and X. Lin, “Speedup graph processing by graph ordering,” in *Proceedings of the 2016 International Conference on Management of Data*, 2016.

[52] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, “Betweenness centrality on gpus and heterogeneous architectures,” in *Proceedings of the 6th Workshop on General Purpose ProcessorUsing Graphics Processing Units*, 2013.

[53] F. Khorasani, R. Gupta, and L. N. Bhuyan, “Scalable simd-efficient graph processing on gpus,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, IEEE, 2015, pp. 39–50.

[54] A. Segura, J.-M. Arnau, and A. González, "Scu: A gpu stream compaction unit for graph processing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 424–435.

[55] A. Segura, J.-M. Arnau, and A. González, "Irregular accesses reorder unit: Improving gpgpu memory coalescing for graph-based workloads," in *The Journal of Supercomputing*, 2022, pp. 762–787.

[56] Y. Lü, H. Guo, L. Huang, Q. Yu, L. Shen, N. Xiao, and Z. Wang, "Graphpeg: Accelerating graph processing on gpus," *ACM Trans. Archit. Code Optim.*, May 2021.

[57] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez, "Exploiting locality in graph analytics through hardware-accelerated traversal scheduling," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2018, pp. 1–14.

[58] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 472–488.

[59] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *2018 IEEE Interna-*

tional Symposium on Workload Characterization (IISWC), IEEE, 2018, pp. 134–145.

[60] K. Lakhotia, R. Kannan, S. Pati, and V. Prasanna, "Gpop: A cache and memory-efficient framework for graph processing over partitions," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 393–394.

[61] B. Coleman, S. Segarra, A. J. Smola, and A. Shrivastava, "Graph reordering for cache-efficient near neighbor search," in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022, pp. 38 488–38 500.

[62] R. Nasre, M. Burtscher, and K. Pingali, "Data-driven versus topology-driven irregular computations on gpus," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 463–474.

[63] B. Tine, K. P. Yalamarthy, F. Elsabbagh, and H. Kim, "Vortex: Extending the risc-v isa for gpgpu and 3d-graphics," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 754–766.

[64] F. Elsabbagh, B. Tine, P. Roshan, E. Lyons, E. Kim, D. E. Shim, L. Zhu, S. K. Lim, and H. Kim, *Vortex: OpenCL Compatible RISC-V GPGPU*, Feb. 2020. arXiv: `2002.12151 [cs]`.

[65] B. Tine, V. Saxena, S. Srivatsan, J. R. Simpson, F. Alzammar, L. Cooper, and H. Kim, "Skybox: Open-source graphic rendering on programmable risc-v gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[66] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "Pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, pp. 752–785, Aug. 2014.

[67] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," 2004, p. 75.

[68] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54*, p. 4, 2014.

[69] *Vortex: A RISC-V GPGPU*, `https://github.com/vortexgpgpu/vortex`.

[70] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable gpu graph traversal," *ACM Trans. Parallel Comput.*, Feb. 2015.

[71] D. Zwillinger and S. Kokoska, *CRC standard probability and statistics tables and formulae*. 1999.

[72] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, 2016. arXiv: 1609.02907.

[73] D. A. F. Alcantara, “Efficient hash tables on the gpu,” Ph.D. dissertation, 2011.

국 문 초 록

GPU에서 그래프의 균형 처리를 위한 스케줄과 저장 형식의 분리

올바른 스케줄와 저장 방식을 선택해야지만 그래프 알고리즘를 GPU에서 효율적으로 처리할 수 있다. 기존 GPU 그래프 처리 프레임워크는 알고리즘에 대해 최적의 스케줄와 저장 방식을 찾기 위해 튜닝 공간에 대해 반복적인 탐색 방식을 수행하지만, 그들의 스케줄들과 저장 방식들이 처리 모델 상에서 긴밀하게 결합되어 있기 때문에 저장 방식을 고려한 최적의 조합을 찾는데 어려움을 겪는다. 또한, 이러한 결합은 개발자가 튜닝 공간을 확장하는 것을 어렵게 만든다. 그러나 최적화 조합을 탐색하기 위해 가능한 모든 최적화 조합을 구현하는 것은 상당한 구현 부담을 초래한다. 따라서 기본 구성 요소 전반에 걸쳐 최적화 조합을 효율적으로 탐색하기 위해서는 스케줄과 저장 방식의 명확한 분리가 필수적이다.

본 학위 논문은 GPU에서의 그래프 처리 과정에서 세 가지 핵심 구성 요소인 스케줄, 저장 방식, 그리고 알고리즘을 분리하는 추상화를 제시함으로써 튜닝 공간을 확장하고자 하였다. 본 논문은 기존 최적화의 특성을 분석하고, 각 핵심 구성 요소에 대한 기본 추상화 인터페이스를 제시하며, 새로운 그래프 처리 모델을 제시한다. 또한, 이 연구는 스케줄, 저장 방식, 알고리즘을 분리 할 뿐만 아니라 추상화 오버헤드 없이 효율적으로 통합하는 새로운 GPU 그래프 처리 프레임워크인 GRAssembler 를 제안한다. 이러한 접근 방식을 통해 해당 연구는 GPU에서의 그래프 처리의 커버리지, 조합 가능성, 확장성, 모듈성을 향상시킨다.

더 나아가, 본 연구는 분리된 추상화가 기존 연구들을 통합할 뿐만 아니라 성능 향상을 위한 기회를 발견하는데 도움이 됨을 보였다. 첫째, 본 연구는 추상화 인터페이스의 동작을 고려함으로써 새로운 최적화를 제안 할 수 있음을 보였다. 저장 방식이 메모리 엑세스 패턴을 결정한다는 점에 초점을 맞추어, 이 연구는 그래프와 GPU의 특성에 맞춘 메모리 접근 패턴을 가능하게 하는 새로운 저장 방식인 CR^2 을 제안한다. 둘째, 기존 연구들의 추상화된 동작들을 분석함으로써

하드웨어로 쉽게 가속 가능한 부분을 찾을 수 있음을 보였다. 이 연구는 기존 스케줄이 균형을 맞추기 위해 정보를 공유하고 생성하는 과정에서 하드웨어의 부재로 인한 런타임 오버헤드를 가짐을 관찰하고, 스케줄을 가속화하기 위해 희소 연산을 밀집 연산으로 변환하는 새로운 경량 GPU 기능 유닛 마이크로아키텍처인 SparseWeaver를 제안한다.

결과적으로, 이러한 효율적인 분리와 통합 덕분에, GRAssembler 는 튜닝 공간을 336에서 4,480으로 확장하였으며, 최신 GPU 그래프 처리 프레임워크와 비교했을 때 기하 평균 성능이 30.4% 향상되었다. 또한, CR^2 저장방식은 기존 저장 방식들에 비해 1.53배의 성능 향상을 이루면서 메모리 사용량을 기하 평균 기준 32.1% 줄였으며, SparseWeaver는 전용 로직 레지스터 증가로 인한 0.045%의 낮은 영역 증가를 통해 기존 스케줄 방법보다 2.49배 빠른 실행 시간을 보여주었다.

핵심되는 말: 그래프 처리, GPU, 저장 방식, 스케줄, 최적화, 부하 분산, 자동 튜닝