PROCESSING GRAPHS AND SPARSE MATRICES EFFICIENTLY

BY

SERIF YESIL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Josep Torrellas, Chair
Professor David Padua
Professor Wen-mei Hwu
Assistant Professor Charith Mendis
Professor Adam Morrison, Tel Aviv University
Dr. José E. Moreira, IBM Research

ABSTRACT

Graph analytics applications are hard to optimize. Their performance is highly dependent on their input graph, which can vary significantly. One effective approach to solve graph analytics applications is to represent the input graph as a sparse matrix, and use the operators of sparse linear algebra. There are numerous methods developed to accelerate the resulting sparse matrix computations. Unfortunately, no single method can outperform all others for all types of matrices. In order to obtain high performance on a wide range of matrices, we need to be able to determine the best method to use for each individual matrix.

In this work, we target automated optimization of graph applications. First, we consider the data-driven graph algorithms with priority scheduling. We perform a detailed performance analysis of various priority schedulers and observe that the performance is application and input graph-dependent. Based on our insights, we develop the Priority Merging on Demand (PMOD) priority scheduler, a dynamic mechanism that can adapt to graph and application characteristics at runtime.

Next, we focus on topology-driven graph analytics, where all vertices of a graph are processed in every iteration. These applications can be represented as generalized Sparse Matrix-Vector multiplication (SpMV) operations. However, SpMV style computations are challenging with emerging power-law graphs due to their skewed and highly irregular memory behavior. To address these challenges, we propose Locality-Aware Vectorization (LAV), which can improve the locality and efficiency of vectorization for power-law graph analytics. Furthermore, we identify that the optimizations applied in LAV form an optimization search space encompassing many different SpMV methods. Moreover, various combinations of these optimizations can yield the best performance for matrices and graphs from different application domains. For this reason, we develop Matrix-Vector Performance Predictor (MVPP): a machine learning-based performance prediction model for SpMV predicting the best strategy for individual matrices.

Finally, we note that many applications can benefit from increasing compute intensity provided by Sparse-Matrix Matrix Multiplication (SpMM). In our final work, we develop Dense Dynamic Blocks (DDB), a hybrid approach to accelerate SpMM by utilizing both vector and emerging matrix units in recent processors. However, like SpMV, we observe that not all matrices benefit from DDB, and the best SpMM method should be selected on a per-matrix basis. Hence, we develop SpMM Optimizer (SpMM-OPT), a machine learning approach to select the best SpMM strategy in functional unit-rich processors.

ACKNOWLEDGMENTS

I want to express my gratitude to many people who have helped me succeed during my doctoral studies. First of all, I would like to thank my advisor Josep Torrellas for his mentorship and contributions. It was an excellent experience to work with him and learn from him. Furthermore, I would like to thank my committee members: Professor David Padua, Professor Wen-mei Hwu, Professor Charith Mendis, Professor Adam Morrison, and Doctor José Moreira, for their helpful comments and feedback. In particular, I would like to thank Prof. Morrison, whom I had the privilege to work with during the entirety of my doctoral education. Additionally, I would like to thank Dr. Moreira for his help and guidance during my internship and our continued collaboration.

I thank all of the members of the i-acoma group for their support, encouragement, and feedback at every step of my Ph.D. Specifically, I would like to thank Azin Heidarshenas and Tanmay Gangwani, with whom I worked closely, for their guidance and help. I also would like to express my gratitude to Apostolos Kokolis, Antonio Franques, Ronak Buch, and Tom Shull for their support and friendship.

I am grateful to my family, whom I received unconditional support for my whole life. They have encouraged me to pursue all my endeavors. Finally, I thank my life-long partner Funda Atik for being there for me during times of joy and hardship and for her unconditional support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Graphs are essential computational abstractions to represent relations between people and objects. These relations can be friendships in a social network, links among web pages, or road connections among cities. Many applications utilize graph structures. For instance, graph analytics algorithms are often used to analyze social, web, and e-commerce networks [1, 2, 3, 4]. One effective approach to solve graph analytics applications is to represent the input graph as a sparse matrix, and use the operators of sparse linear algebra. In this case, it is important to optimize the execution of sparse matrix primitives. Such primitives are frequently used in computational science and engineering applications, often as fundamental building blocks for many complex applications such as linear solvers, computational modeling and simulations [5, 6, 7, 8, 9, 10]. Recently, machine learning with sparse data using graphs has also gained attention [11, 12].

Graph-based computations are hard to optimize. This is because the performance of a given graph analytics application is highly dependent on the structure of the input graph. There is no one-size-fits-all solution to the challenges facing graph applications. In particular, when graphs are represented as matrices, the matrices are highly irregular and have significantly different characteristics. Attaining high performance for all types of sparse matrices is very challenging. In this thesis, we propose mechanisms to address these challenges. We provide various techniques to automate the optimization process for these applications.

First, we consider the task-based graph algorithms, such as the Single-Source Shortest Paths (SSSP), Breadth-First Search (BFS), path-finding algorithms like A*, data-driven PageRank [3, 13], Delaunay triangulation [14] for computational geometry, and minimal spanning tree (MST) finding [15] for network design. In these applications, each task performs vertex and/or edge updates and can also create new tasks. Usually, a set of active vertices and/or edges propagates information to their neighbors. Whenever a vertex or an edge has new information to propagate, a new task is created and added to a worklist. A scheduler then schedules these vertices to multiple threads to be executed in parallel. This approach is called the data-driven approach [13]. Many graph processing frameworks employ data-driven execution to process graphs efficiently [16, 17].

More often than not, a priority order imposed on the tasks processed increases efficiency. Such schedulers are called Concurrent Priority Schedulers (CPSs). However, these CPSs are hard to design. Specifically, the performance of CPS highly depends on the input graph and application characteristics. Furthermore, CPSs come with a fundamental tradeoff between enforcing the priority order and paying for synchronization overhead. If a CPS wants to

enforce a strict priority order, the application would require the minimum amount of work. However, enforcing the strict priority order increases synchronization overhead. By relaxing this order, thereby decreasing the synchronization overhead, we can obtain more efficient execution at the expense of performing superfluous work. Many CPSs were proposed to address this tradeoff. In the first part of this thesis, we analyze this fundamental tradeoff and various CPS designs from the literature. Then, inspired by our analysis, we propose a self-optimizing CPS design, Priority Merging on Demand (PMOD), to address these challenges. PMOD is robust and can dynamically adapt to the application and graph characteristics.

In contrast to the data-driven execution model, many applications require processing all vertices of a graph in every iteration of the application, namely the topology-driven execution model [13]. Examples of these applications include PageRank [3] and HITS [4]. These applications can be represented as generalized Sparse Matrix-Vector multiplication (SpMV) operations [18, 19, 20]. In fact, it can be faster to use SpMV style computation than data-driven execution [21] if the number of active vertices in the worklist is high. These ideas are also adopted in graph processing frameworks such as Ligra [17]. However, executing SpMV on large-scale power-law graphs results in highly irregular memory access patterns and poor cache utilization. Furthermore, we find that existing SpMV locality and vectorization optimizations are ineffective for the power-law graphs.

To address these problems, we propose Locality-Aware Vectorization (LAV). LAV is a new approach that leverages a graph's power-law nature to extract locality and enable effective vectorization for SpMV-like memory access patterns. LAV employs several optimizations: *column reordering*, which moves columns with a high number of nonzeros together to improve locality, *row reordering* to improve vectorization, and *segmenting* for utilizing last-level cache (LLC) capacity efficiently.

Moreover, SpMV is not only an essential kernel in graph analytics but also in many other domains. SpMV is used in different domains such as protein networks and physical and structural problems. In general, the sparsity and topological characteristics of the matrices used in different domains are very different. As a result, numerous methods have been developed to accelerate SpMV in different domains, including our LAV. Unfortunately, no single method consistently yields the best performance for all sparse matrices for SpMV. Therefore, to achieve high performance for a wide range of matrices, we need mechanisms to identify the best SpMV strategy for a given sparse matrix. We identify that optimizations applied in LAV can be used to form an optimization search space encompassing many different SpMV methods. Moreover, various combinations of these optimizations can yield the best performance for matrices and graphs from other application domains. For this reason, we develop Matrix-Vector Performance Predictor (MVPP): a performance prediction model

for predicting the best SpMV method for a given matrix. Given how challenging it is to pick the best SpMV method for a given input matrix due to the wide range of sparse matrix characteristics, it is arduous to develop a heuristic to get the job done. For this reason, we propose to leverage machine learning (ML) with a novel feature set for summarizing sparse matrices.

Finally, some applications can be made more efficient by using the more computationally intensive Sparse Matrix Dense Matrix Multiplication (SpMM) kernel. SpMM is an essential component of linear solvers [5, 6], and graph applications such as multi-source BFS and SSSP applications, recommender systems, and graph-based machine-learning [11, 12, 22]. Additionally, various CPU and GPU systems are now featuring matrix-multiply hardware facilities tailored for dense matrix operations such as POWER10's Matrix-Multiply Assist (MMA) facilities and NVIDIA's Tensor Cores [23, 24]. These specialized units can execute dense matrix-multiply operations on small matrices (e.g., blocks of size 4×4). These units are successfully utilized to maximize performance for dense matrix operations [23, 24, 25]. In our work, we propose Dense Dynamic Blocks (DDB) to utilize the new matrix-multiply units to speed up SpMM. DDB is a hybrid approach that maximizes the floating-point throughput by utilizing both vector and matrix units. We acknowledge that not all matrices benefit from using matrix units. For this reason, we develop SpMM-OPT. SpMM-OPT is a machine learning approach with a carefully crafted feature set to select the best SpMM strategy in a functional unit-rich environment.

1.1 CONTRIBUTIONS

Optimizing graph analytics applications and sparse matrix primitives is inherently very challenging. In this section give an overview of the challenges addressed in this work and our novel solutions.

1.1.1 Understanding Priority-Based Scheduling of Graph Algorithms

Many task-based graph algorithms benefit from executing tasks according to some programmer specified priority order. To support such algorithms, graph frameworks use Concurrent Priority Schedulers (CPSs), which attempt—but do not guarantee—to execute the tasks according to their priority order. While CPSs are critical to performance, there is insufficient insight on the relative strengths and weaknesses of the different CPS designs in the literature. Such insights would be valuable to design better CPSs for graph processing.

This work performs a detailed empirical performance analysis of several advanced CPS designs in a state-of-the-art graph analytics framework running on a large shared-memory server. Our study finds that all CPS designs but one impose significant overheads that dominate running time. Only one CPS—the Galois system's obim—typically imposes negligible overheads. However, obim's performance is input-dependent and can degrade substantially for some inputs. Based on our insights, we develop PMOD, a new CPS that is robust and delivers the highest performance overall.

1.1.2 Speeding Up SpMV for Power-Law Graph Analytics by Enhancing Locality & Vectorization

Graph analytics applications often target large-scale web and social networks, which are typically power-law graphs. Graph algorithms can often be recast as generalized Sparse Matrix-Vector multiplication (SpMV) operations, making SpMV optimization important for graph analytics. However, executing SpMV on large-scale power-law graphs results in highly irregular memory access patterns and poor cache utilization. Worse, we find that existing SpMV locality and vectorization optimizations are largely ineffective on modern out-of-order (OOO) processors— they are not faster (or only marginally so) than the standard Compressed Sparse Row (CSR) SpMV implementation.

To improve performance for power-law graphs on modern OOO processors, we propose *Locality-Aware Vectorization* (*LAV*). LAV is a new approach that leverages a graph's power-law nature to extract locality and enable effective vectorization for SpMV-like memory access patterns. LAV splits the input matrix into a dense and a sparse portion. The dense portion is stored in a new representation, which is vectorization-friendly and exploits data locality. The sparse portion is processed using the standard CSR algorithm. At the end, LAV achieves an average speedup of 1.5× over CSR and prior methods for several power-law graphs with only 3.3% storage overhead.

1.1.3 Predicting Performance of Sparse Matrix-Vector Multiplication with Machine Learning

Sparse Matrix-Vector Multiplication (SpMV) is an essential kernel in graph analytics and scientific computing. Numerous methods have been developed to accelerate SpMV. However, no single method gives consistent speedups across a wide range of matrices. For this reason, a performance prediction model is needed for predicting the best SpMV method for a given matrix.

However, predicting an SpMV method's performance for a given matrix is challenging. First, there is high variability in terms of performance. This variability of performance comes from the diversity of graph structures. We identify that the size, skew, and locality characteristics of a given matrix are the driving force for the success of a given SpMV method. Second, the publicly available real-world matrices do not cover a wide range of matrix characteristics regarding size, skew, and locality. We perform a detailed characterization of these real-world matrices and find that this set is highly biased towards a few SpMV methods.

In this work, we address these challenges. We develop a machine learning technique to predict the speedup of multiple SpMV methods for a given matrix. Our prediction model relies on a novel feature set to summarize a given matrix's size, skew, and locality characteristics. To train our model, we use random matrix generators to create matrices with diverse behavior to form our training set. Ultimately, our method predicts the magnitude of speedup for different SpMV methods for a wide range of matrices with high accuracy. In the end, MVPP achieves an average speedup of 2.4× with respect to the MKL baseline. Furthermore, MVPP achieves 1.13× speedup over MKL inspector-executor with 50% less preprocessing overhead.

1.1.4 Optimizing SpMM for Processors with Vector and Matrix Units

Recent processors have been augmented with matrix-multiply units that operate on small matrices, creating a functional unit-rich environment. Moreover, these units have been successfully employed on dense matrix operations such as those found in the Basic Linear Algebra Subprograms (BLAS). This work exploits these new matrix-multiply facilities to speed up Sparse Matrix Dense Matrix Multiplications (SpMM) for highly sparse matrices.

SpMM is difficult to optimize. The sparse matrices have highly irregular memory access behavior. Additionally, each matrix has unique characteristics, making it hard to find a single SpMM strategy that works well for all sparse matrices. The addition of matrix-multiply units makes this even more challenging.

In this work, we address these challenges. First, we design Dense Dynamic Blocks (DDB), a method to utilize the new matrix units. DDB has two specialized versions: DDB-MM and DDB-HYB. DDB-MM is a strategy that only utilizes the matrix-multiply facilities. On the other hand, DDB-HYB is a hybrid approach that maximizes the floating-point throughput by using vector and matrix units together. Furthermore, we design a prediction mechanism for identifying the best SpMM strategy for a given sparse matrix and dense matrix pair: SpMM Optimizer (SpMM-OPT). SpMM-OPT selects among vector unit oriented, matrix

unit oriented, and the hybrid strategies for the highest floating-point throughput while also taking the cache optimizations into account. We observed that DDB method can achieve up to 1.1 TFlops/s and 2.4 TFlops/s for double- and single-precision SpMM, respectively on a POWER10 processor. Furthermore, we compare SpMM-OPT against an oracle method and show that it can choose an effective SpMM method for a large set of sparse matrices.

1.2 THESIS ORGANIZATION

As described in the previous sections, in this work, we present application-specific solutions and auto-tuning techniques for graph analytics applications and sparse matrix primitives. Our solutions take advantage of the characteristics of the input matrices and utilize state-of-the-art wide vector and matrix-multiply units deployed in the current processors.

First, we study task-based graph algorithms with a focus on priority-based scheduling. In Chapter 2, we analyze the tradeoffs of various priority scheduling mechanisms. Furthermore, we describe our adaptive priority scheduler PMOD for graph analytics applications. Chapter 3 targets the acceleration of SpMV for the large power-law graphs. We propose LAV that leverages wide vector units of recent Skylake processors while employing cache optimizations. In Chapter 4, we describe a machine learning-based method to navigate through the complex search space of SpMV optimizations for a wide range of sparse matrix types. Finally, Chapter 5 presents an SpMM approach to exploit matrix-multiply units in recent processors for sparse matrices and its accompanying machine learning-based auto-tuner SpMM-OPT.

CHAPTER 2: UNDERSTANDING PRIORITY-BASED SCHEDULING OF GRAPH ALGORITHMS

2.1 INTRODUCTION

The fundamental role that graph algorithms play in many important applications motivates the use of parallelism to speed them up. As a result, there is a large body of work on programming models and runtimes for parallel graph processing (e.g., [16, 17, 26, 27, 28, 29]). Many of these frameworks use a task-based model on a shared-memory environment. In this model, the graph algorithm's computation is broken down into dynamically-created tasks that are scheduled to run in parallel. This is an attractive model, as it is very general, reasonably easy to program, and can be executed efficiently on large commercial shared-memory machines [16].

Task-based graph algorithms are usually unordered. This means that tasks can be processed in any order. However, many unordered algorithms benefit from executing tasks according to some programmer-specified priority order. For instance, consider the single-source shortest paths (SSSP) problem, which computes the shortest distance from a source vertex s to every vertex in the graph. It is more efficient to process vertices roughly ordered in increasing distance from s. If distant vertices are processed first, the execution will likely discover shorter paths to those vertices later, making the earlier computation on the distant vertices redundant.

Graph algorithms that benefit from task processing in priority order are ubiquitous. They include search algorithms, such as SSSP and Breadth-First Search (BFS), and path-finding algorithms, such as A*, which are used for gaming, transportation, and robotics [30]. They also include PageRank [13], which is widely used for graph analytics, Delaunay triangulation [14] for computational geometry, maximal flow computation [31] for optimization and scheduling, and minimal spanning tree (MST) finding [15] for network design.

To run such algorithms efficiently, graph frameworks use a *Concurrent Priority Scheduler* (CPS) data structure to pick tasks for execution roughly in their priority order [16, 32, 33, 34, 35, 36]. While CPSs are critical to the performance of these graph applications, there is insufficient insight on the relative strengths and weaknesses that the CPS designs in the literature exhibit in practice. In particular, many of the designs have never been evaluated against each other in a state-of-the-art graph analytics system on a large shared-memory machine. Further, there is no detailed low-level empirical analysis of the sources of CPS overhead. Only with this type of empirical analysis can one identify the bottlenecks and get insights into how to design better CPSs for graph processing.

This work addresses this open question. It performs a detailed empirical performance analysis of several advanced CPS designs using Galois [16], a state-of-the-art graph analytics framework, running on a 40-core, 4-socket, shared-memory machine. We study four state-of-the-art CPS designs that use different, representative structures: a SprayList [33] using a shared lock-free skip list, two variations of a distributed array of priority queues [34], and obim, the default CPS in Galois.

All the CPSs we study avoid synchronization bottlenecks by having processors pick *some* high-priority task rather than the highest-priority one. Despite this property, we find that all CPSs but obim impose significant overhead on the fine-grained tasks of graph algorithms, typically causing CPS overhead to dominate execution time. We further observe that obim's performance depends on having many tasks per each priority value. When there are few tasks per priority level, obim's execution becomes slower than sequential execution—even if the input creates abundant task parallelism. One currently has to *manually* tune the graph application to work around obim's performance stability problem.

Guided by our study, we develop *PMOD*, a new CPS that extends obim to dynamically and automatically adapt to the ranges of priorities exhibited by the input graph. This property makes PMOD's performance stable, automatically achieving comparable performance to obim with input-specific manual tuning. PMOD obtains the highest performance of all CPSs.

Contributions. We make the following contributions.

- We conduct the first extensive empirical analysis of different CPS algorithms proposed in the literature. Our analysis on a large shared-memory machine yields qualitative and quantitative insights about the tradeoffs in CPS design to drive future research on CPSs and graph applications.

- Our analysis points out a missing point in the CPS design space: a CPS that provides high performance in a consistent manner that does not depend on the input and that does not require manual tuning. We propose the new *PMOD* CPS, which has these properties.

2.2 BACKGROUND AND MOTIVATION

This work focuses on graph analytics applications running on a shared-memory machine. The graph, as well as data associated with vertices and edges, is stored in a standard graph representation, such as compressed sparse row (CSR) or column (CSC). The graph

representation does not affect our discussion or findings. The graph applications can read and write the data associated with vertices and edges and update the graph structure by adding/removing vertices and edges.

2.2.1 Priority Scheduling

Graph analytics systems employ different programming models for parallel graph algorithms [16, 17, 26, 27, 28, 29]. We focus on the popular *task-based* model, in which the algorithm is implemented using dynamically-created tasks that may run in parallel. The task-based programming model can express algorithms designed for other models and can run very efficiently on current machines. For example, Galois [16]—one of the best performing shared-memory graph analytics systems—uses this model.

In a task-based graph algorithm, each task performs vertex and/or edge updates and can also create new tasks. Algorithms can be ordered or unordered. An *ordered* algorithm *requires* tasks to execute according to a user-specified priority order. Figure 2.1a shows an example ordered algorithm implementing Dijkstra's single-source shortest paths (SSSP) algorithm [37]. The algorithm finds the distance from some source vertex s to all other vertices in a weighted directed graph G. Each task processes a vertex, and its priority is the length of the path it discovers from the source to that vertex. A task attempts to extend the shortest path by creating a task for each neighbor. Tasks must run in strict priority order.

Ordered algorithms have parallelism [38, 39], but mining it is hard since it requires speculating across ordering constraints. Efficiently performing such speculation requires special hardware (e.g., [39]) because the overheads of software-based speculation negate the parallelism benefits [38]. Consequently, graph analytics systems favor *unordered* algorithms.

```
1  Shared data: Graph G. Initially, in each vertex v, v.dist = ∞
2  sssp(s):
3    schedule( visit (s, 0))
4
5  visit(v, dist):
6    if v.dist ≠ ∞: return
7    v.dist = dist
8    foreach neighbor u of v:
9      if (u.dist == ∞):
10       d = dist + weight(v, u)
11       schedule( visit (u, d))
12
```

(a) ordered

```
13 sssp(s):
14   s.dist = 0
15   schedule( visit (s, 0))
16 visit(v, dist):
17   if v.dist ≠ dist: return
18   foreach neighbor u of v:
19     atomically:
20       d = dist + weight(v, u)
21       if (d < u.dist)
22         u.dist = d
23         schedule( visit (u, d))
```

(b) unordered

Figure 2.1: Task-based SSSP algorithm. The priority of a task is *dist*.

An unordered algorithm produces a correct result regardless of task execution order, making it easier to mine its parallelism. Figure 2.1b shows an unordered SSSP algorithm. Tasks now do not necessarily terminate if the vertex has already been visited. In addition, they update the distance only if they decrease it, which requires synchronization (Lines 19–23).

Many unordered algorithms still use priorities, running more efficiently when task execution mostly follows priority order but remaining correct when tasks execute out of priority order (i.e., *priority inversion*). For instance, executing an SSSP task out of order can lead to the task's distance update being overwritten later, thereby wasting the task's cycles. Therefore, graph analytics systems use *priority scheduling*, which attempts—but does not *guarantee*—to execute tasks in priority order.

2.2.2 Concurrent Priority Schedulers

A *Concurrent Priority Scheduler* (CPS) is a data structure that stores the set of pending tasks, and provides a way to add and remove tasks. A CPS supports two main operations: `Enq` and `Deq`. An `Enq`(t, p) operation enqueues a task t with priority p in the data structure. A `Deq`() operation dequeues a task to execute. The execution consists of threads repeatedly invoking `Deq` and executing the obtained task (which invokes `Enq` if it creates new tasks) until no tasks are left.

A CPS can be implemented by a concurrent Priority Queue (PQ) [40, 41, 42], in which a `Deq` returns the highest priority task (i.e., the one with minimal p value). In this case, all concurrent `Deq` calls contend on the same task, inducing synchronization overhead. Therefore, practical CPS designs *relax* the priority queue's semantics and return *some* high-priority task, not necessarily the highest-priority one.

We consider several representative state-of-the-art CPS designs:

SprayList. The SprayList [33] is a popular design that stores tasks in priority order inside a lock-free skip list [43].[1] A SprayList `Enq` inserts the task into the skip list, which is sorted by priority order. Lock-free skip list insertions are not serialized and run concurrently. A SprayList `Deq` operation removes a random high priority task, which it finds by performing a short random walk on the skip list. Different processors thus typically pick different tasks and do not contend. A `Deq` returns one of the $\approx p \log^3 p$ highest-priority tasks with high probability, where p is the number of processors.

Distributed Queues. These designs reduce contention by maintaining an array of concurrent PQs and allowing a processor to access a random PQ in each operation. Processors thus typically access different PQs and do not contend.

We consider two designs that differ in how operations access the PQs: MultiQueue and RELD. The MultiQueue [34] maintains an array of $q = cp$ concurrent PQs, where $c > 1$

[1] A skip list [44] is a randomized list-based data structure in which nodes are randomly linked into a hierarchy of linked lists. With high probability, each list contains about half of the nodes in the list below it, allowing searches to "skip" over multiple elements.

is a parameter, and p is the number of processors. An `Enq` inserts the task into a random PQ. A `Deq` picks two random PQs and removes the task of higher priority among the two. Recent work suggests that, in expectation, the MultiQueue `Deq` picks one of the $\approx p$ globally highest-priority tasks (i.e., over all PQs) [45].

RELD (random enqueue, local dequeue) maintains an array of p concurrent PQs, each of which is associated with a processor. As in the MultiQueue, an `Enq` inserts the task into a random PQ. A `Deq` dequeues from the requesting processor's PQ, blocking if it is empty. A hardware implementation of RELD is used by the Swarm architecture [39].

Galois obim. Galois' default CPS is obim (Ordered By Integer Metric) [16, 46], which strives to avoid communication and synchronization between processors. This is a lightweight, distributed design, with one *bag* (i.e., unordered queue) data structure per priority. Each bag is a distributed structure consisting of as many FIFO queues as sockets (i.e., NUMA domains) in the machine.

An `Enq` inserts a task into the bag associated with the task's priority, creating such a bag if it does not exist. A `Deq` finds a bag to dequeue from by traversing the bags in priority order until it finds a non-empty bag. The processor keeps dequeueing from this bag in subsequent `Deq`s until it becomes empty.

The bag is designed to minimize communication and synchronization by satisfying most `Enq` and `Deq` operations from private per-processor buffers so that processors access the shared FIFO queues only infrequently. A bag's FIFO queues hold *chunks* of c tasks (typically, $c = 64$). When a processor inserts tasks into the bag, it first buffers them in a private local chunk; once the chunk fills up, the processor enqueues it into the FIFO queue of the processor's socket. The tasks in the chunk then become visible to other processors. A processor dequeues a chunk from its socket's queue. If the queue is empty, the processor steals from one of the remote queues. It then consumes tasks from the dequeued chunk one at a time.

obim maintains the list of bags in a *global map* data structure, which is read and written by all threads. To reduce synchronization and cache coherence traffic, each thread caches the contents of the global map in a *local map*. When enqueueing a task, a thread looks up the bag associated with the task's priority in its local map. If not found, the thread creates a new bag and updates the global map accordingly. When dequeueing, if a thread fails to find work in the bags listed in its local map, it refreshes the local map with the information in the global map and tries again.

2.3 PRIORITY SCHEDULING INSIGHTS

2.3.1 Fundamental Tradeoff

The fundamental tradeoff in CPS design is that of communication and synchronization overhead versus unnecessary work performed. Specifically, if the CPS is such that tasks are obtained and processed in perfect priority order, the algorithm typically performs the least amount of work. However, the communication and synchronization operations necessary to obtain the tasks in such order are costly.

Instead, if the CPS obtains tasks without following strict priority order, there is a chance that some of the work performed will be superfluous; it will have to be repeated under more up-to-date conditions. However, by relaxing the priority order, the CPS can reduce communication and synchronization.

We classify the execution cycles of an unordered graph algorithm that uses a CPS as shown in Table 2.1. The algorithm's cycles spent processing tasks can be performing Good Work (*GWork*) or Useless Work (*UWork*). The algorithm's cycles spent in the CPS can be Enqueue (*Enq*), Dequeue (*Deq*), and Failed Dequeue (*FDeq*) cycles. The latter occurs when a dequeue fails to find a task to execute. The remainder (*Other*) cycles in the algorithm are spent running other framework code. Typically, if strict priority execution is maintained, *Enq* and *Deq* will be high, but *UWork* will be low. If priority execution is relaxed, the opposite will occur. Our goal is to find a balance for the best performance.

Table 2.1: Execution cycle breakdown of unordered graph algorithms.

Category	**Description**
Good Work (GWork)	Processing a task. The work ends up being useful.
Useless Work (UWork)	Processing a task. The work later proves useless.
Enqueue (Enq)	Pushing a task to the CPS data structure.
Dequeue (Deq)	Retrieving a task from the CPS data structure.
Failed Dequeue (FDeq)	Attempting and failing to retrieve a task from the CPS data structure.
Other	Executing other graph analytics framework code.

2.3.2 Addressing the Tradeoff

We posit that there are two main approaches to address the outlined tradeoff: one that emphasizes reduction in useless work and another that emphasizes reduction in communication/synchronization overhead.

Emphasis on Minimizing Useless Work. CPSs that emphasize retrieving tasks close

to the priority order invest in synchronization/communication operations to obtain high-quality tasks. Although modern CPSs avoid the contention bottleneck of dequeueing tasks from a single shared priority queue [42, 44], they still access globally shared data on each CPS operation, which typically incurs multiple cache misses.

Specifically, the SprayList maintains a global skip list, from which it dequeues a random task close to the head. In the distributed queue designs such as the MultiQueue and RELD, threads access a random queue in most operations. The MultiQueue picks a random queue to enqueue and dequeue, while RELD enqueues in a random remote queue and dequeues locally. The use of randomness in these CPSs causes every CPS operation that a thread performs to access (with high probability) different memory locations than those accessed by its previous CPS operation. These locations are also frequently written to (e.g., queue heads in the MultiQueue and RELD). Consequently, despite returning high-quality tasks, CPS operations in these designs incur multiple cache misses and are thus relatively time-consuming compared to the fine-grained tasks used in graph applications.

Emphasis on Minimizing Communication. The obim CPS [16] exemplifies a CPS design that prioritizes avoiding shared-memory communication, maximizing the locality of CPS operations, and minimizing their overhead. It maintains tasks in *per-priority distributed* unordered queues (bags). CPS operations on these bags are efficient and highly local. First, tasks can be inserted/removed at the per-priority queue tail/head without any list traversal, as their order is not important. Second, to amortize overheads, enqueues and dequeues are performed at a coarse grain by enqueueing and dequeueing a chunk of tasks at a time [16]. Such amortization is not trivial to add to the first approach to CPS designs.

For the obim design to be efficient, a worker thread must have a fast way to find the bag associated with a priority value. Further, the per-priority queues should contain many tasks.

In principle, this design is prone to useless work because threads working on a bag do not frequently search for a new bag that could have a higher priority to reduce communication. We shall see, however, that such useless work is typically rare in practice.

We call this CPS approach *Per-Priority Queue*, and the first approach, which includes the SprayList, MultiQueue, and RELD CPSs, *Combined-Priority Queue*.

2.3.3 Observations

We analyzed the execution of several graph algorithms on a large multi-socket shared-memory server with the CPS implementations described in Section 2.2.2. We ran the algorithms on many different graph inputs and various thread counts. Our main observations are shown in Table 2.2. Detailed measurements supporting these observations are presented

in Section 2.6.

Table 2.2: Observations on successful CPS designs.

Observation	Description
O1	Some task processing with priority inversion is a good choice if it is the result of a lightweight CPS.
O2	The number of tasks per priority is input-dependent. Typically, processing a task T generates new tasks with priorities not too different from T's priority. However, with some graph inputs, these new tasks have a very wide range of priorities, with negative performance effects for the Per-Priority Queue approach.
O3	Enqueueing and dequeueing a chunk of tasks at a time is very beneficial but does not seem compatible with the Combined-Priority Queue approach.

In *O1*, we observe that some task processing with out-of-order priorities (i.e., *priority inversion*) can be a good choice if done on communication-minimizing CPS implementations such as those of Section 2.3.2. These CPSs have very low-overhead operations while producing acceptable amounts of useless work. Even with the useless work, the overall result is higher performance than other CPSs, especially for large core counts.

The main reason why *O1* holds is shown in *O2*: when a task with priority p is processed, it often tends to generate other tasks with only slightly lower priorities ($p+\epsilon$, where ϵ is small[2]). This property means that if threads are working on the highest priority bags, newly created tasks do not change the highest priority, and so continuing to work on the bag does not create useless work. Moreover, even if we place these new tasks into the current bag (as discussed in Section 2.4), the bag will contain tasks with similar priorities, and so processing the new tasks will cause only minor priority inversion.

The properties of input graphs cause this behavior. Processing a vertex v may cause the insertion of its adjacent vertex v' in the work queue. But the priority of v' is often not much different than that of v. For example, in SSSP, the difference is the weight of the edge joining v to v'; in BFS, the difference is 1. As another example, in algorithms like MST, the priority is simply the degree (i.e., the number of edges) of a vertex. Such numbers are often not very different.

Observation *O2* also notes that, sometimes, applications or inputs generate tasks with a wide range of priorities, yielding a small number of tasks per priority value. This causes poor performance in CPSs using the Per-Priority Queue approach. If there are only a few tasks per priority, the private chunks where threads buffer created tasks typically fail to fill up, and thus the tasks they contain never become visible to other threads. Consequently, many bags will appear empty, while all the tasks are stored in threads' private chunks. This situation

[2]Higher p values mean lower priorities.

causes the algorithm to spend a lot of time searching for bags to work on. Moreover, threads will quickly empty any bags found and thus not benefit from locality.

On the other hand, the Combined-Priority Queue approach is more tolerant of this behavior. This is because it orders the tasks according to their priority in a queue. The queue is processed in the same way, irrespective of the ranges of priorities it contains.

This effect happens in SSSP with many *road network* graphs. The difference in priorities between two adjacent vertices is the weight of the connecting edge. This is the distance between the two corresponding vertices. A vertex may be connected to several vertices at widely different distances. As a result, the range of priorities can be very large, causing major dequeueing overheads with the Per-Priority Queue approach.

Finally, *O3* notes that a lot of execution overhead is eliminated by performing enqueueing and dequeueing of tasks in a coarse-grain manner—i.e., using chunks of tasks at a time. Such an approach is easy to support with the Per-Priority Queue approach. However, it is hard to support with the Combined-Priority Queue approach without destroying its robustness to the priority distribution (see *O2*). It is not possible to simply create chunks based on inserted tasks, because the resulting chunks would contain tasks with different priorities, and so these chunks would not be totally ordered in the queue. Alternatively, it is possible for each thread to buffer the tasks it creates in per-priority private chunks and insert filled-up chunks into the global queue (similarly to the Per-Priority Queue approach). However, this design suffers from the problem noted in *O2*—when there are few tasks per priority, these chunks will not fill up and will not be inserted into the queue.

We also find that large chunks are undesirable, as they lead to load imbalance among cores. Indeed, in a chunk-based environment, a core bundles up the work that it is generating in chunks before enqueuing the chunks in the worklist. Only at that point is the work visible to other cores. If chunks are large, it takes a long time for a core to fill up a chunk and make it globally visible. During that time, other cores may be idle looking for work.

2.4 PMOD: AN ADAPTIVE CPS

2.4.1 Main Idea

Based on our observations, we introduce a new CPS design that is able to minimize both communication/synchronization overhead and unnecessary work performed, hence delivering high performance. Our scheme is called PMOD (Priority Merging On Demand) and builds on the ideas in Section 2.3.2, which obim implements.

While obim's idea of keeping a queue per priority is often highly effective, it can sometimes result in subpar performance. Hence, in PMOD, the queues are *per priority groups*, and such groups change dynamically at runtime.

Specifically, PMOD dynamically identifies when the execution is using too many priority queues, and there are too few tasks per priority queue. This is an inefficient operating point because threads spend a substantial time searching for work. In this case, PMOD combines a set of consecutive priorities into a single queue. We call this process *Priority Merging*. This process can be repeated multiple times dynamically. Every time, the *Merging Factor* (or the number of consecutive priorities that are merged) increases. The Merging Factor is always a power of two.

PMOD also dynamically estimates when the execution is using too few priority queues. This is also an inefficient operation point because, by merging disparate priorities, threads run the risk of suffering priority inversion and executing useless work. In this case, PMOD separates the priorities into more queues. This is *Priority Unmerging*. It is done dynamically, in decreasing powers of two.

To see how the algorithm works, consider Figure 2.2. Figure 2.2a shows an environment with too many priority queues. Core i has a long list of priority queues in its local map (Section 2.2.2), but they are all currently empty, because no chunk has been filled-up and deposited in any of these queues. Core i wastes time traversing this list, and then has to go to the global map to obtain work. Our PMOD CPS measures the frequency of such global accesses. If the frequency is higher than threshold $Freq_{global}$, PMOD considers priority merging.

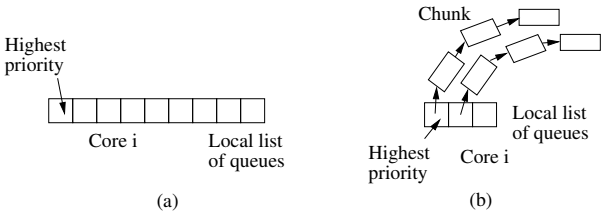


Figure 2.2: Run with too many (a) and too few (b) priority queues.

In this case, PMOD first computes the range of priorities of the tasks that have been recently enqueued in the work queue, and divides it by the Merging Factor. This gives the number of queues needed to cover the range (Num_{queues}). Then, PMOD computes the ratio between the number of tasks recently enqueued in the work queue and Num_{queues}. To reduce traversal overheads, we want this ratio to be equal to or higher than threshold *MinDensity*. If necessary, PMOD increases the Merging Factor (and hence consolidates the list of queues) so that this is the case. From now on, newly arriving chunks will be enqueued in the consolidated list of queues.

Figure 2.2b shows an environment with too few priority queues and too many tasks per queue. It is possible that core i performs substantial useless work. Our PMOD CPS regularly

measures the number of recent successful dequeues from each priority queue. If one such value is higher than threshold *MaxPops*, PMOD suspects that queues are too long and priority inversion may be taking place, and considers priority unmerging. Again, it computes the number of queues needed to cover the recent priority range (Num_{queues}). To minimize the amount of useless work, we want Num_{queues} to be greater or equal to Min_{queues}. Therefore, if this is not the case, PMOD decreases the Merging Factor and hence expands the number of queues. From now on, newly arriving chunks will be enqueued in the expanded list of queues.

When the list of queues is consolidated or expanded, care is taken not to create major priority inversion. We give details of the algorithm below.

The SSSP application in the Galois package [16] has a parameter called *delta* that right-shifts the priority values passed to the CPS, hence compressing the range of priorities. This compression can have an effect similar to PMOD's priority merging, but it is an *application-level* change that requires *manual*, per-input tuning, and is static. PMOD attains this effect and the opposite one (i.e., priority expansion) automatically, transparently to application and user, and dynamically.

2.4.2 PMOD's Priority Ordering

PMOD does not merge or unmerge task queues physically. Instead, it dynamically adjusts the mapping from application-supplied priority values to queues, creating new queues if necessary. To this end, PMOD maintains the base-2 logarithm of the current value of the Merging Factor in variable *lmf* — which stands for Logarithm of Merging Factor. A higher *lmf* means that more priorities are merged. PMOD groups tasks into queues not simply based on the priority value p of the task, but based on the pair $(p\texttt{>>}lmf_0, lmf_0)$ of the task, where `>>` denotes bitwise right-shift, and lmf_0 is the value of *lmf* at the time when the task was enqueued. Therefore, to find a queue, PMOD indexes the structure with the pair of values above. To enqueue a task, `Enq` (t, p) inserts task t into the queue $(p\texttt{>>}lmf, lmf)$, creating it if it does not exist.

The algorithm used by PMOD to order queues minimizes priority inversions. Given two queues, (p_1, l_1) and (p_2, l_2), which one has a higher priority? First, PMOD computes $l = \max(l_1, l_2)$. Then, it computes $p_1\texttt{>>}(l - l_1)$ and $p_2\texttt{>>}(l - l_2)$. The lowest value of these two is the queue with the highest priority. Note that by shifting by $l = \max(l_1, l_2)$, PMOD is helping the queue with the tasks with lower l_i, namely those inserted when *lmf* was lower. However, suppose that $p_1\texttt{>>}(l - l_1) = p_2\texttt{>>}(l - l_2)$. In this case, the queue with the lower l_i is given the higher priority. Again, tasks inserted when *lmf* was low are favored.

This ordering algorithm generally prevents low-priority tasks inserted after priority merging from taking precedence over high-priority tasks that were inserted before merging. Consider an example. Assume that *lmf=0* and that we have queues for tasks with priorities {(1,0),(5,0),(8,0),(10,0),(11,0),(32,0)}. Suppose that *lmf* increases to 3, and now a task with $p = 9$ is received to be enqueued. It should be enqueued in queue $(9\texttt{>>}3, 3) = (1, 3)$. Since this queue does not exist, PMOD creates the queue $(1, 3)$.

Now, we consider how PMOD orders the new queue with respect to the existing queues. According to the algorithm described above, it orders $(1, 3)$ after $(5, 0)$, since $l = 3$ and $(1\texttt{>>}0) > (5\texttt{>>}3)$. It also orders $(1, 3)$ after queues {(8,0), (10,0), (11,0)} because, while 1>>0 is equal to $[8, 10, 11]\texttt{>>}3$, its *lmf* is higher. Finally, it orders $(1, 3)$ before the $(32, 0)$ queue.

Note that with *lmf=3*, tasks with priorities 8–11 map to the same queue. Thus, the placement of the new queue relative to the existing queues indexed by $(8, 0)$, $(10, 0)$, and $(11, 0)$ is not crucial. What is crucial is to guarantee that the higher priorities $(0, 0)$ to $(7, 0)$ get processed first, by ordering the new queue after their queues. With this ordering, there may be some small priority inversion, but it is tolerable in practice—especially since the priority of the tasks created decreases monotonically during the execution. However, if PMOD only considered *p*>>*lmf*, $(1, 3)$ would be placed before $(5, 0)$. The same would be true for any arriving task with $p < 40$ while *lmf=3*. This would create a high priority inversion.

2.4.3 PMOD Flow

Figure 2.3a shows the `Deq` () and `Enq` () routines. When a `Deq` () invocation obtains work from the global map rather than obtaining the work locally, we call it a *synchronizing* dequeue. In this case, the *syncDeq* routine is executed. PMOD calls *mergeCheck()* and *unmergeCheck()* to make decisions on merging or unmerging. To make the decisions, PMOD uses some thread-local counters that count a set of events since the last merging/unmerging decision. Such events are the number of `Deq`s (*nDeqs*), synchronizing `Deq`s (*nSyncDeqs*), `Enq`s (*nEnqs*), and the range of priorities enqueued in the system (*minB* and *maxB*). A read of one such counter returns the aggregation of all the thread-local counters. After a merge/unmerge decision, the counters for all the threads are reset. These details are omitted from Figures 2.3a and 2.3b for brevity.

2.4.4 Merging and Unmerging

Figure 2.3b shows the *mergeCheck()* and *unmergeCheck()* routines. Consider *mergeCheck()* first. Merging is needed when there are too many priority queues and few tasks per queue.

```
1  Deq():        // dequeue routine
2      nDeqs++
3      // number of dequeue operations since last merge/unmerge
4      if(can dequeue locally)
5          return fastDeq()
6      else
7          return syncDeq()       // synchronizing dequeue

9  syncDeq():
10     nSyncDeqs++ // number synchronizing dequeue operations
11     mergeCheck() // check for, and potentially perform, merging
12     if(lmf not changed) // if merge didn't occur
13         unmergeCheck()
14         // check for, and potentially perform, unmerging
15     if(lmf changed)
16         // if merge or unmerge happened, reset counters
17         nEnqs = nSyncDeqs = nDeqs = 0
18         MaxB = Priority.MIN   // minimum priority value
19         MinB = Priority.MAX   // maximum priority value
20     // rest of the dequeue

22 Enq(task, taskPrio):          // enqueue routine
23     // number of enqueue operations since last merge/unmerge
24     nEnqs++
25     // keep track of the priorities created since
26     // last merge/unmerge operation
27     MaxB = max(MaxB, taskPrio)
28     MinB = min(MinB, taskPrio)
29     prio = taskPrio >> lmf
30     // proceed to enqueue in queue indexed by prio
```

(a) `Deq` and `Enq` routines.

```
1  mergeCheck():
2    if ((nSyncDeqs / nDeqs) ≤ Freq_global)
3      return
4    // calculate the number of priority groups
5    Num_queues = (MaxB>>lmf) − (MinB>>lmf)
6    // calculate the average number of tasks
7    // per priority group
8    fillRatio  = nEnqs / Num_queues
9    if ( fillRatio  < MinDensity)
10     // may merge priority groups
11     // to get closer to MinDensity
12     lmf += log2(MinDensity/fillRatio)

14 unmergeCheck():
15   if (nDeqs from single prio_group ≤ MaxPops)
16     return
17   // calculate the number of priority groups
18   Num_queues = (MaxB>>lmf) − (MinB>>lmf)
19   if (Num_queues < Min_queues)
20     // too few prio_groups, may unmerge
21     lmf −= log2(Min_queues / Num_queues)
```

(b) Merge and unmerge operations.

Figure 2.3: PMOD's `Enq` and `Deq` operations.

PMOD detects this condition by checking if the fraction of `Deq` calls that go to the global map (i.e., fail to find work locally), *nSyncDeqs/nDeqs*, is greater than $Freq_{global}$. If merging is needed, PMOD computes Num_{queues}, the number of queues needed to cover the priority range observed since the last *lmf* update (Line 5), and *fillRatio*, the average number of tasks that each of these Num_{queues} queues would have received since the last *lmf* update (Line 8). If *fillRatio* is lower than *MinDensity*, PMOD may cautiously increase the Merging Factor, so that Num_{queues} decreases.

Next, consider *unmergeCheck()* (Figure 2.3b). It is triggered when the number of `Deq`s from a single priority group since the last *lmf* update exceeds a threshold *MaxPops*. When triggered, the algorithm checks whether Num_{queues} is smaller than threshold Min_{queues}. If so, PMOD may cautiously decrease the Merging Factor.

2.5 EXPERIMENTAL SYSTEM

2.5.1 Graph Framework and CPSs

We run our experiments on a 40-core shared-memory machine. The machine has 40 Xeon E7-4860 cores running at 2.27 GHz, organized in 4 sockets of 10 cores each. Each core has

32 KB L1 instruction and data caches, and a 256 KB L2 cache. Each socket has a shared 24 MB L3 cache. The machine has 128 GB of memory.

We evaluate the CPSs using the Galois graph analytics framework [16]. Galois provides a programming model that supports the unordered execution of loop iterations. It executes the iterations in parallel, treating each iteration as a task. For instance, each iteration can operate on one vertex.

We implement (or use an existing implementation of) the four CPS algorithms described in Section 2.2.2, plus our proposed PMOD CPS described in Section 2.4. The CPSs are SprayList (SL), MultiQueue (MQ), Random-Enqueue Local-Dequeue (RELD), obim (some applications require variations called obim-O and obim-D that we describe in Section 2.5.3), and PMOD. Table 2.3 lists them.

Table 2.3: CPS algorithms evaluated.

Name	**Description**
SprayList (SL)	Concurrent priority-ordered skip list.
MultiQueue (MQ)	Array of concurrent priority queues.
Remote Enqueue, Local Dequeue (RELD)	
obim	Distributed structure (bag) per priority. For applications that are manually tuned for obim, we evaluate both the default and the optimized settings of obim, which we call obim-D and obim-O, respectively.
PMOD	Bag per adaptive priority group.

For the SprayList, we use the publicly available implementation by its authors (`https://github.com/jkopinsky/SprayList`). We base our MultiQueue implementation on the original authors' implementation (obtained by request). In executions with t threads, we use a MultiQueue with $4t$ priority queues. Our implementation replaces the coarse-grained locked sequential priority queues in the original implementation with lock-free skip lists, as we found that the skip lists perform better. We use the skip list implementation from the SprayList code. We implement RELD based on our MultiQueue code to obtain the most accurate comparison. For obim, we use the code provided by Galois. We set the chunk size to 64 after tuning experiments.

PMOD Parameters. For PMOD, we set $Freq_{global}$ to *1/chunk_size*, *MaxPops* to $4 \times$ *chunk_size*, *MinDensity* to 64, and Min_{queues} to 16. For all applications, we start with *lmf*=0.

We select $Freq_{global}$ to be *1/chunk_size* since, if we go to the global map at this frequency, we will have at least one chunk per priority. *MinDensity* is selected to support at least one chunk worth of tasks per priority bin. *MaxPops* and Min_{queues} are selected empirically. *MaxPops*

tries to eliminate the case where there are too many tasks per priority, and Min_{queues} sets the minimum number of different priority groups in the system.

2.5.2 Input Datasets

We evaluate the applications on the input graphs detailed in Table 2.4. However, we only show plots for representative inputs.

Table 2.4: Input graphs.

Graph	**# Vertices**	**# Edges**	**Size(MB)**
USA roads (*rUSA*) [47]	24 M	58 M	628
West USA roads (*rW*) [47]	6 M	15 M	165
Twitter40 (*tw*) [48]	42 M	1469 M	6K
Web-Google (*wg*) [49]	875 K	5 M	46
Soc-LiveJournal1 (*lj*) [50]	5 M	69 M	564

The input graphs have different characteristics. The USA roads (*rUSA*) and West USA roads (*rW*) graphs are road networks. Twitter40 (*tw*) is a real-world social network graph from Twitter; we use the largest connected component and assign edge weights using a random uniform distribution from the range $[0, 100]$. Web-google (*wg*) is the web graph released as part of the Google Programming Contest. Soc-LiveJournal1 (*lj*) is the friendship social network of the LiveJournal online community. The *wg* and *lj* datasets come from [51].

2.5.3 Applications

We evaluate the following applications: Single-Source Shortest Paths (SSSP), Breadth-First Search (BFS), PageRank (`PR` and `PR-D`), Minimum Spanning Tree (MST), and A*. All applications but A* are standard benchmarks in the Galois distribution; we implement A* from scratch.

Single-Source Shortest Paths: The SSSP algorithm in Galois is based on the delta-stepping algorithm [52]. Each task is associated with some vertex v and attempts to extend the shortest path from s to v. The priority of a task is the distance it assigns to its vertex.

Breadth-First Search: BFS uses breadth-first search to traverse a graph, where the weight of each edge is 1. Tasks are defined as in SSSP, with the priority now being the number of edges on the discovered path.

PageRank: We use a pull-push version of PR, in which the page rank of a vertex is calculated by iterating over its incoming edges (pull) and then propagating the change observed

to the vertex's outgoing neighbors (push) [13]. The priority of a task is the PR value of its vertex, which is a floating-point number. To be able to use obim, we need to convert the priorities to integers. We evaluate PR with two conversion methods: Taking the whole part of the floating-point number (PR), and taking the whole part plus the three digits after the decimal point (PR-D for "detailed").

Minimum Spanning Tree: MST uses Boruvka's algorithm to find a spanning tree over all vertices with minimum total edge weight. Each task is associated with a vertex. The task picks the vertex's minimum weight edge and merges the vertex and its neighbor connected by the edge, scheduling a task to visit the new vertex. The priority of a task is its vertex's degree.

A*: A* is a path-finding algorithm. It calculates the distance from a source vertex s to a destination vertex d. Unlike SSSP, the search is guided by a heuristic value. The heuristic value is the expected distance to vertex d from the currently visited vertex. To guide the search, the priority of a vertex is the sum of its distance to s plus its expected distance to d.

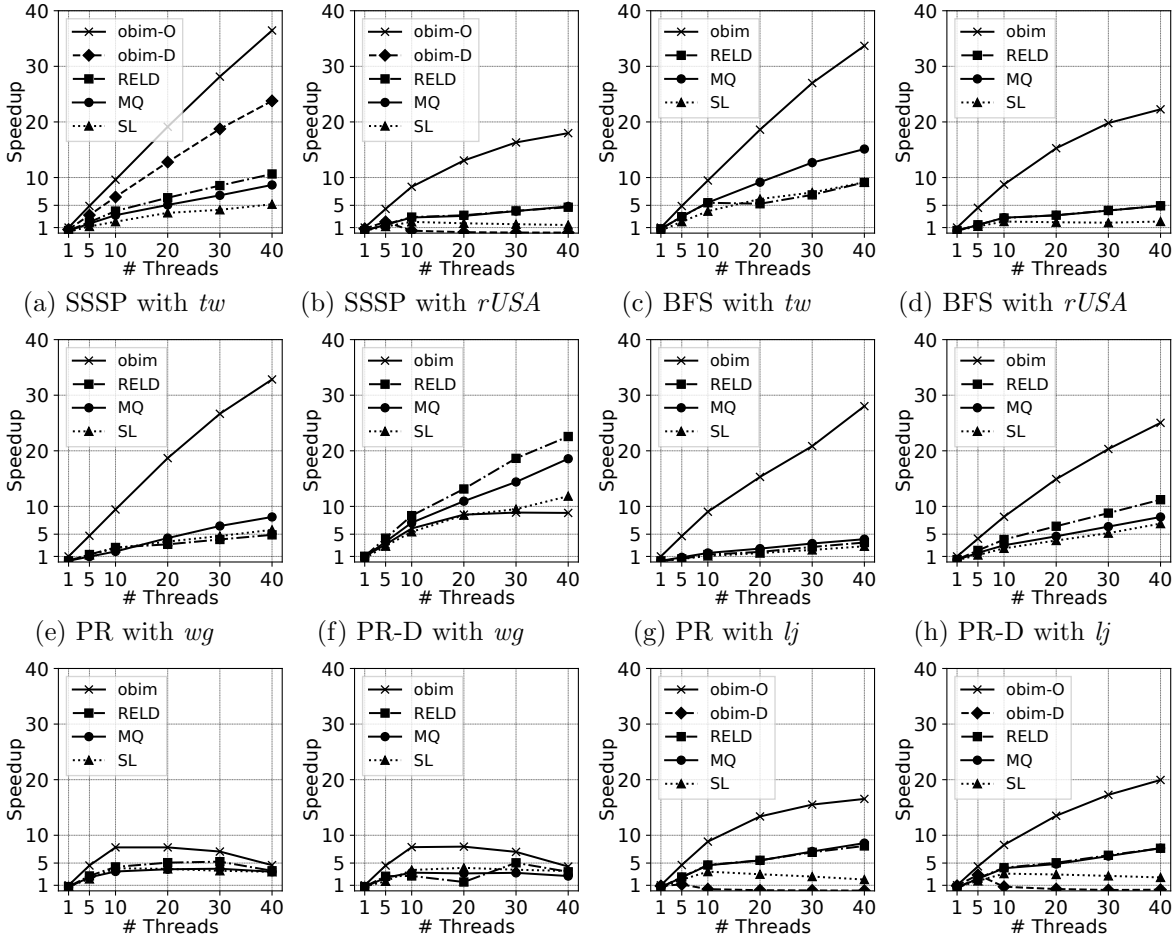
Manually tuned applications: Galois' SSSP application supports manual tuning to obtain the best performance for each input graph. It takes a *delta* (Δ) parameter and right-shifts priority values by Δ bits. This shift decreases the number of distinct priority values and significantly impacts the performance with obim. The other CPSs ignore Δ. The value of Δ specified on the Galois website is 8. However, we search for the values of Δ that attain maximum performance with obim. Such optimal values are $\Delta = 14$ for road network graphs, and $\Delta = 0$ for the other graphs. Hence, we evaluate two versions of SSSP: one with the default $\Delta = 8$ (obim-D) and one with the optimal Δ value that we identify through empirical search (obim-O).

Our A* code similarly supports a Δ parameter for scaling priority values with obim, and we evaluate two versions of A*: one with the default $\Delta = 8$ (obim-D) and one with an optimal Δ value (obim-O), which is 14 for all the graphs considered.

2.6 FINDINGS

2.6.1 CPS Performance Characteristics

Figure 2.4 shows the speedups obtained by the CPS schemes as we change the number of threads, relative to the single-threaded execution of obim. Recall that for SSSP and A*, we use versions of the applications that are manually tuned for obim, one with the default settings of obim (obim-D), and one with the optimized settings of obim (obim-O). For these



(a) SSSP with *tw* (b) SSSP with *rUSA* (c) BFS with *tw* (d) BFS with *rUSA*

(e) PR with *wg* (f) PR-D with *wg* (g) PR with *lj* (h) PR-D with *lj*

(i) MST with *rW* (j) MST with *rUSA* (k) A* with *rW* (l) A* with *rUSA*

Figure 2.4: Speedups of the CPS schemes relative to the single threaded execution of obim (or obim-O).

two applications, the speedups are relative to obim-O.

The figure shows that, generally, obim or obim-O yield the best performance. However, obim-D often has very poor performance: in SSSP with *rUSA*, A* with *rW*, and A* with *rUSA*, obim-D is the lowest curve, barely above 1. We also see that the speedups vary greatly with the application and input. obim or obim-O attain speedups of 20-40 for many applications and inputs; SL provides the worst average performance; and the distributed CPSs MQ and RELD are in between. We now consider the results in detail.

Search applications (SSSP, BFS, and A*): SSSP's performance under obim heavily depends on the input and on the Δ parameter. For instance, SSSP obtains nearly linear speedup under obim-O on the *tw* input (Figure 2.4a). Under obim-D, the speedup is lower, about 24 at 40 threads, but still higher than under the other CPSs. On the other hand, on the *rUSA* input, the speedup under obim-O does not exceed 20, and under obim-D, the speedup

collapses and the application runs slower in parallel than sequentially (Figure 2.4b). As we show later, this collapse is due to the lack of sufficient work per priority value, as described in observation *O2* (Section 2.3.3). In contrast, while BFS shows some input-sensitivity under obim, it is less drastic. Under *rUSA*, BFS still sees a significant speedup of 22. While this is less than the speedup obtained for *tw*, the performance of BFS on *rUSA* does not collapse (Figures 2.4c–2.4d). Finally, A*, which runs only on the road networks, behaves similarly to SSSP. Specifically, obim-O yields the best speedup, but the performance collapses with obim-D (Figures 2.4k–2.4l).

PR and PR-D: Recall that the difference between PR and PR-D is that the latter has a 1,000× wider range of priorities. Such change has a major effect on obim. Specifically, PR with *wg* under obim yields a speedup of 32 for 40 cores, making obim the best CPS (Figure 2.4e). However, PR-D under obim becomes substantially slower. On PR-D, all the other CPSs do better than obim, which delivers a speedup lower than 10 (Figure 2.4f). This effect is also due to observation *O2* in Section 2.3.3. Although not shown because the figure shows speedups relative to obim, the other CPSs are much less affected by the range of priorities. In summary, obim needs many tasks per priority value to perform well, whereas the remaining CPSs are much less sensitive to this metric.

MST: Unlike the other applications, MST does not scale past a single socket (10 threads) under any CPS. Under obim, MST enjoys almost linear scalability within the socket, but subsequently degrades and obtains a speedup of 5 at 40 threads (Figures 2.4i–2.4j). Under the other CPSs, the speedup never exceeds 5, even within a socket. The reason for MST's lack of scalability is that, unlike the other applications, MST merges vertices as it executes, thereby decreasing the number of vertices and the available parallelism [38].

2.6.2 CPS Performance Analysis & Observations

We now analyze the reasons for the CPS performance trends and empirically support the high-level observations made in Section 2.3. Figure 2.5 breaks down the 40-threaded execution time of the applications under the different CPS schemes. We use the categories detailed in Table 2.1: performing *useless* work (*UWork*), performing *good* work (*GWork*), executing CPS `Enq` (*Enq*), executing CPS `Deq` that returns a task (*Deq*), executing CPS `Deq` that fails to return a task (*FDeq*), and executing non-CPS Galois framework code (*Other*).

CPS Overhead Determines the Execution Time. From Figure 2.5, we see that the CPS overhead (*Enq*, *Deq*, and *FDeq*), rather than useless work, typically determines the execution time. Generally, the relaxed priority scheduling performed by the CPSs creates negligible

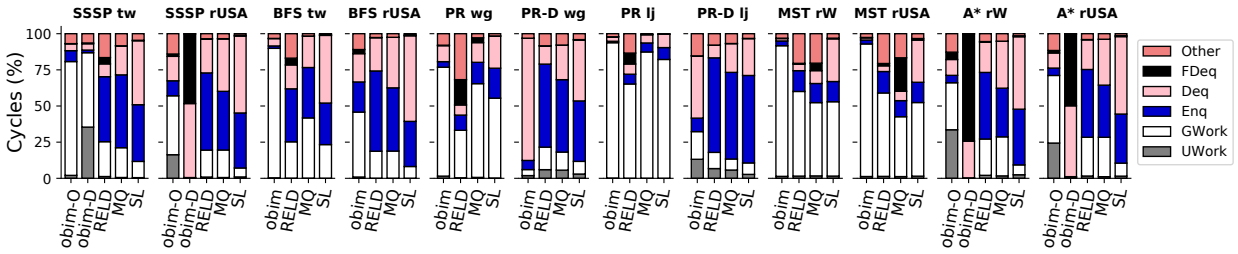


Figure 2.5: Breakdown of the normalized execution cycles for 40-threaded executions.

useless work. Only obim sometimes creates non-negligible useless work, which we explore shortly. For the most part, applications under obim spend less than 10% of their time inside the CPS (*Enq*, *Deq*, or *FDeq*). The exceptions are PR-D and the search applications with the road networks, which we discuss shortly. With the other CPSs, applications typically spend 50%–90% of their time in CPS code.

The differences in CPS time are explained by Figure 2.6, which shows the cycles per `Enq` and `Deq` operations. Observe that obim's `Enq` and `Deq` operations are orders of magnitude cheaper, on average, than in the other CPSs. The main factor behind this difference is that obim buffers both enqueued and dequeued tasks in *chunks*, thereby amortizing communication costs by improving cache locality.

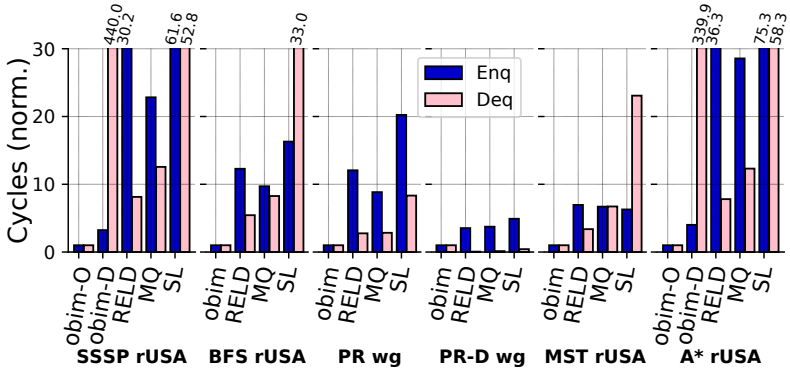


Figure 2.6: Cycles per `Enq` and `Deq` operation for 40-threaded executions normalized to obim (or obim-O).

Another factor behind CPS time is data structure complexity. First, chunk insertion and removal from obim's FIFO queues are $O(1)$ operations, whereas searching a skip list is an $O(\log n)$ operation, where n is the size of the skip list. Second, MQ and RELD distribute tasks among multiple skip lists, resulting in shorter skip lists than the single skip list maintained in SL. Thus, `Enq`s in SL are slower (Figure 2.6). Finally, removing a task in MQ and RELD is an $O(1)$ operation (removing the head of some skip list), whereas SL performs a random walk and is thus slower (Figure 2.6).

As obim is the most competitive CPS, we now study its performance in detail.

Performance Sensitivity to Priority Values (O2). Performance with obim decreases when there are few tasks per priority. In this case, threads that enqueue tasks do not manage to fill their chunks. Since chunks are thread-local, such tasks remain invisible to other threads. Threads spend substantial time going over many priority bags trying to find work. This case manifests itself as higher Deq time in the application, and more cycles per `Deq` operation. Moreover, in some cases, threads do not find work at all, even though there are private tasks pending execution, leading to load imbalance. *FDeq* cycles capture such unsuccessful dequeue attempts.

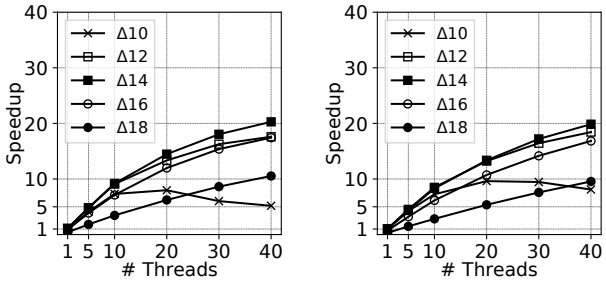
The search applications on the road networks and PR-D experience this effect. For example, the road networks' edge weights are drawn from large ranges. There are many priority bags, with on average 1.5 tasks per priority. Figure 2.5 shows the result of this effect. Under obim-D, SSSP on *rUSA*, and A* on *rW* and *rUSA* spend all of their time searching for tasks to execute. The combination of *Deq* and *FDeq* cycles accounts for all the execution cycles. Note that *FDeq*—which is small in all other CPSs and inputs—can be over 50% of the execution time. A similar effect is shown in Figure 2.5 for obim in PR-D with *wg* and *lj*.

Useless Work vs. CPS Efficiency Tradeoffs (O1). The Δ parameter in SSSP and A* right-shifts priority values by some bits, effectively *compressing* the priorities into fewer bags. Increasing Δ is thus a tradeoff. It increases the average number of tasks per priority bag, which makes chunking more effective and helps obim find tasks faster. However, lumping together tasks with highly different priorities increases the chance of running tasks out of priority order and performing useless work. Figure 2.5 shows this tradeoff. For SSSP on *rUSA*, and A* on *rW* and *rUSA*, obim-O (which uses Δ =14) reduces the fraction of the execution in the CPS (*Enq*, *Deq*, and *FDeq*) to 25% or less. This is compared to ≈100% in obim-D. However, 20–35% of the time in obim-O is now spent on useless work. Still, obim-O is so lightweight that, despite executing useless work, it is faster than SL, MQ, and RELD (Figures 2.4b, 2.4k, and 2.4l). These CPSs have little useless work (Figure 2.5), but their overhead is so high that the work quality becomes a second-order effect.

Useless work is not free, however. When the number of tasks per priority is large, avoiding useless work pays off. For instance, for SSSP on the *tw* input, running with obim-D ($\Delta = 8$) leads to 35% of the time being spent on useless work, and a maximum speedup of about 24, whereas with obim-O ($\Delta = 0$), useless work is negligible and speedup is nearly linear.

To study this tradeoff, we consider SSSP and A* (which use the Δ parameter), and vary Δ. Figure 2.7 shows the speedups of SSSP and A* for *rUSA* (*rW* shows similar behavior

and is omitted), as we vary Δ from 10 to 18. The speedups are normalized to single-threaded runs with $\Delta = 10$. Figure 2.8, on the other hand, shows detailed cycle breakdowns similar to Figure 2.5.



(a) SSSP with *rUSA* (b) A* with *rUSA*

Figure 2.7: Speedups of obim for different Δ values for SSSP and A*, relative to the single-threaded execution with $\Delta = 10$.

From Figure 2.7, we see that the highest speedups are attained with $\Delta = 14$ (the optimal value used in Section 2.6.1). As Δ moves higher or lower than 14, the speedups decrease. While $\Delta = 12$ and $\Delta = 16$ deliver acceptable speedups, values further out do not. For example, with $\Delta = 10$, the speedups at 40 threads are 5 and 8.

To understand this behavior, consider the cycle breakdown in Figure 2.8. The figure corresponds to 40-threaded executions of SSSP and A*. When Δ is below optimum, there are many priority bags and few tasks per bag. Many tasks remain invisible, buffered in non-filled thread-local chunks. Consequently, idle threads cannot find work efficiently. As shown in the $\Delta = 10$ bars, *Deq* and *FDeq* consume the large majority of cycles. As Δ increases, the contribution of *Deq* and *FDeq* decrease, but useless work appears. When Δ is above the optimum, many different priorities are placed in the same bag, increasing the useless work.

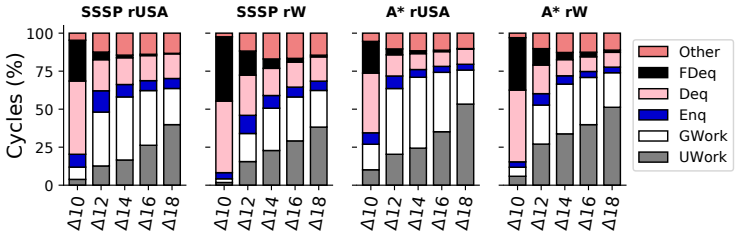
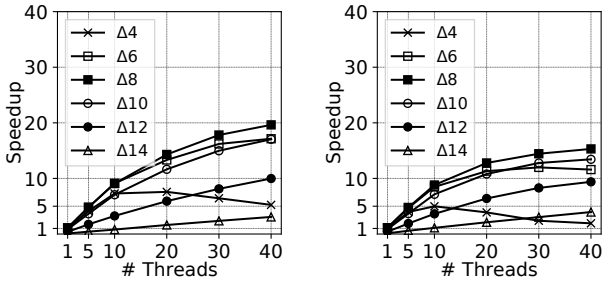


Figure 2.8: Breakdown of execution cycles for 40 threaded-executions of SSSP and A*, while varying Δ.

Optimal Priority Merging Depends on Input Data (O2). How aggressively priorities need to be compressed for obim to attain good performance depends on the priority value distribution *rather than the graph structure*. To show this, we repeat the above experiments scaling down all the edge weights in the graph by 64. This change doesn't alter the shortest

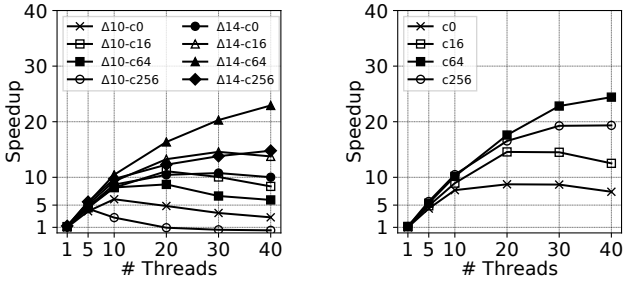
paths or the graphs' topology, but changes the range of priority values. Figure 2.9 shows speedups for SSSP running *rUSA* and *rW* for different Δ values. We can see that now, the optimal Δ decreases to 8.



(a) SSSP with *rUSA* (b) SSSP with *rW*

Figure 2.9: Speedups of obim for different Δ values for SSSP with edge weights scaled down by 64.

Amortizing Communication Using Chunking (O3). Processing tasks in chunks reduces obim's `Enq` and `Deq` operation cost to $O(1)$ on average. For `Enq` and `Deq`, obim accesses shared data structures only once in c operations, where c is the chunk size. Here, we show that chunking is an important factor in obim's performance, and analyze the interaction of chunking and priority compression. Figure 2.10 shows the speedups of SSSP and BFS for *rUSA* as we vary the chunk size from 0 (*c0*, chunking disabled) to 256 (*c256*). *rW* shows similar behavior and is omitted.For SSSP, we have curves for Δ equal to 10 and 14.



(a) SSSP with *rUSA* (b) BFS with *rUSA*

Figure 2.10: Speedups of obim for different chunk sizes for SSSP and BFS, relative to the single-threaded execution with no chunking (and $\Delta = 10$ for SSSP).

The speedups are normalized to single-threaded runs with no chunking and, for SSSP, $\Delta = 10$. We see that the highest speedups are attained with the default chunk size of 64 for both SSSP (with the optimal Δ of 14) and BFS. Both larger and smaller chunk sizes decrease the speedups. For example, with chunking disabled, the speedups at 40 threads are 2-3 times lower.

To understand this behavior, consider the cycle breakdown of the 40-threaded executions in Figure 2.11. Without chunking, obim accesses shared structures on each `Enq` and `Deq` operation. Hence, the *Enq* and *Deq* categories account for ≈80-90% of the cycles. As we increase the chunk size, this fraction goes down. However, larger chunks are harder to fill. Tasks thus remain buffered and inaccessible to other threads. This causes other threads to either work on low-quality tasks (*UWork*) or fail to find tasks altogether (*FDeq*).

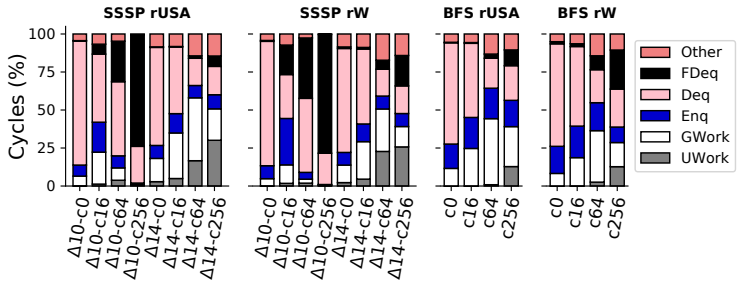


Figure 2.11: Breakdown of execution cycles for 40 threaded-executions of SSSP and BFS while varying the chunk size. For SSSP, we show bars with the suboptimal $\Delta = 10$ and the optimal $\Delta = 14$.

Chunk Size: Load Balancing vs. Overhead Tradeoff. A main reason for the inefficiency with the small suboptimal Δ equal to 10 is that, with few tasks per priority bag, chunks do not get filled, and thus, tasks remain invisible to other threads. Now we evaluate whether decreasing the chunk size solves this problem. Figure 2.10a shows the speedups of SSSP using $\Delta = 10$ with varying chunk size. A smaller chunk size of 16 yields the best performance, instead of 64. Small 16-entry chunks fill faster and become visible faster, which almost eliminates *UWork* cycles, but they impose high `Enq` overhead (Figure 2.11). Consequently, the speedup with 16-entry chunks is only 30% better than with the default of 64, and the best execution time with suboptimal $\Delta = 10$ remains $\approx 3\times$ slower than with the optimal $\Delta = 14$ and chunk size of 64.

Summary of Findings. With the right amount of priority compression, chunks of size 64 serve well to amortize communication without hurting load balancing, whereas adjusting chunk size does not compensate for suboptimal priority compression. The optimal level of priority compression depends on the priority distribution and cannot be determined statically or based on graph topology. This motivates our proposed PMOD CPS.

2.6.3 Effectiveness of PMOD

PMOD Speedups. Table 2.5 presents an overall comparison of CPSs at 40 cores. In the first row, we pick a given CPS and, for each application, compute the speedup of that

CPS over the best CPS for that application. We then take the geometric mean over all the applications. The resulting number indicates how close that CPS is to being the optimal choice as the default CPS for all applications. PMOD's value of 0.93—the highest among all CPSs—means that PMOD is always comparable to the application-specific best performer. Only obim-O has a similar number, but it is not a viable choice for a default CPS since it requires extensive, workload-specific, manual tuning to achieve this result.

Table 2.5: Geometric mean of the CPS speedup compared to best CPS for each application, and geometric mean of speedup for 40 cores.

	PMOD	**obim-D**	**obim-O**	**RELD**	**MQ**	**SL**
W.r.t best CPS for the app.	0.93	0.55	0.89	0.35	0.34	0.18
Speedup for 40 cores	17	10.6	17	6.7	6.5	3.6

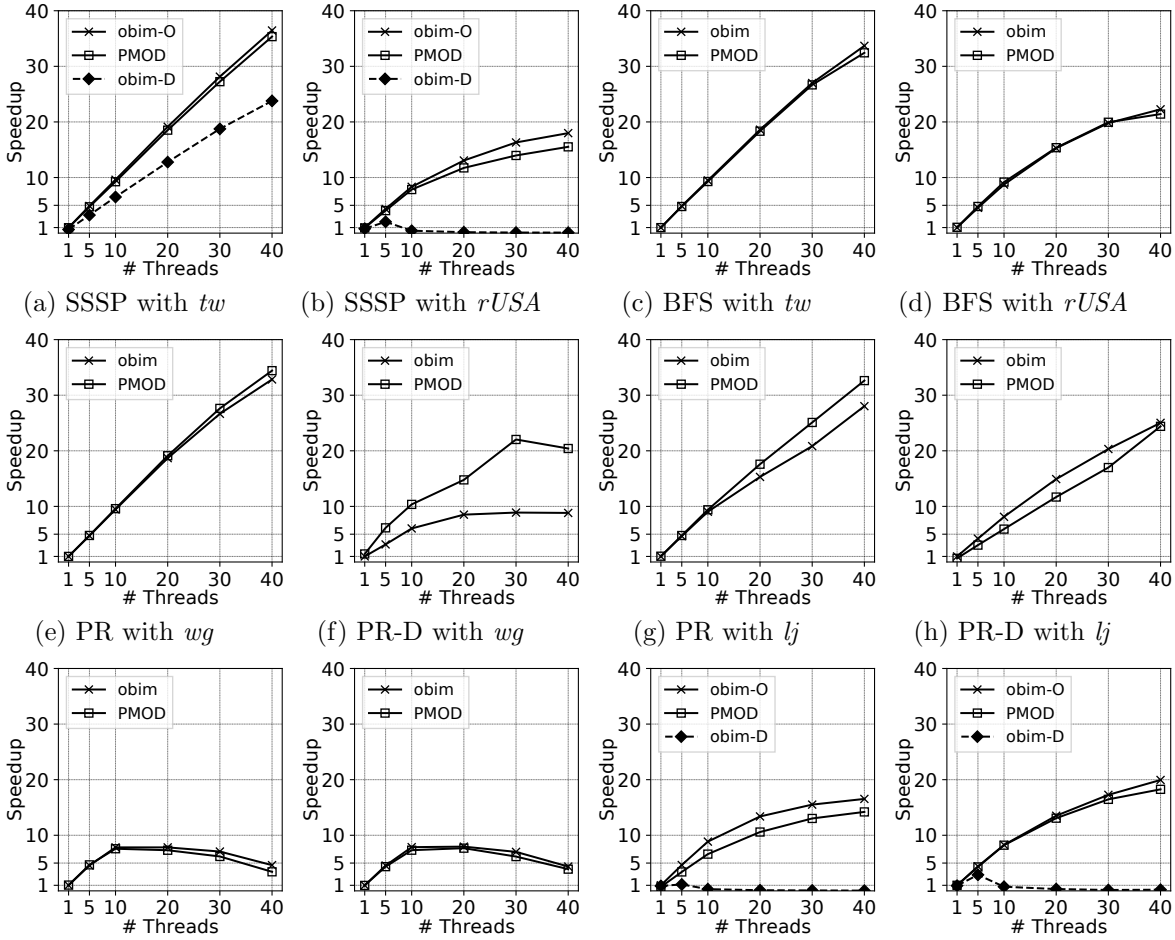
The second row shows the geometric mean speedup of each CPS over a single-threaded obim execution. PMOD is much better than the CPSs that do not require manual tuning. The same is true for obim-O, but obim-O requires manual tuning.

We further evaluate PMOD by comparing its execution time to the obim variants. Figure 2.12 shows the speedups of the applications under PMOD, obim, and, when applicable, obim-O and obim-D. All curves are relative to the single-threaded execution time of obim (or obim-O, when applicable).

We see that PMOD performs as well as obim (and obim-O when applicable) in all cases. Recall that obim-O is obtained through manual tuning, searching for and identifying the optimal Δ. What makes PMOD attractive is its ability to match obim-O's performance without any tuning.

In fact, PMOD outperforms obim in PR-D on the *wg* input. PMOD is twice as fast as obim for 40 threads. The reason is that, as discussed in Section 2.6.2, obim does not work well with the large range of priority values in PR-D. Instead of having the programmer manually work around this problem, as was done in SSSP with the Δ parameter, PMOD adapts to the observed priority ranges and successfully speeds up the application, obtaining a 20× speedup.

To gain further insights, Figure 2.13 breaks down the 40-threaded execution time of the applications in these experiments. For the most part, the breakdowns in PMOD are very similar to those in obim. Importantly, when we have obim-O and obim-D bars, PMOD is similar to obim-O, while obim-D has either many *Deq* and *FDeq* cycles, or many *UWork* cycles. In the case of PR-D on the *wg* input, PMOD has a higher fraction of *GWork* cycles than obim.



(a) SSSP with *tw* (b) SSSP with *rUSA* (c) BFS with *tw* (d) BFS with *rUSA*

(e) PR with *wg* (f) PR-D with *wg* (g) PR with *lj* (h) PR-D with *lj*

(i) MST with *rW* (j) MST with *rUSA* (k) A* with *rW* (l) A* with *rUSA*

Figure 2.12: Speedups of PMOD and obim variants relative to the single-threaded execution of obim (or obim-O).

PMOD Dynamics. We now illustrate how PMOD adjusts its priority merging over time. When an application starts, *lmf* is zero. Table 2.6 shows the sequence of values that *lmf* takes as our applications execute. We show data for each application and dataset shown in Figure 2.12. The table shows the final value of *lmf*, the number of changes, and the values that *lmf* takes.

The data shows that *lmf* increases monotonically. For SSSP with *rUSA* and A*, where under obim-O we manually set Δ to 14, PMOD converges to *lmf* values of 13 and 16, which are close to the optimal Δ. As a result, PMOD's performance is close to obim-O's. Moreover, applications such as PR-D, which do not use Δ, benefit from PMOD's merging mechanism. For instance, PMOD automatically sets the *lmf* value for PR-D to 9 and 10.

lmf often goes through multiple changes. The timing of the changes differs across applications, datasets, and number of threads. Often, the merge operations occur in the beginning

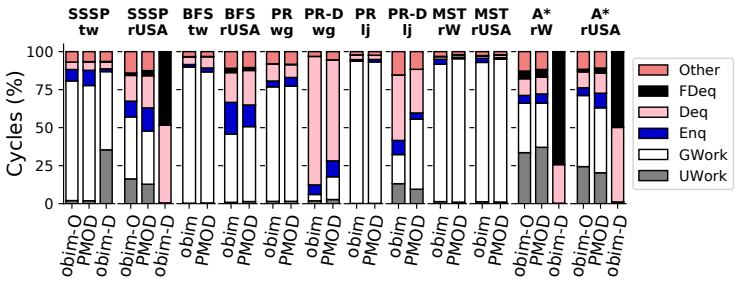


Figure 2.13: PMOD vs. obim execution cycle breakdown (40 threads).

Table 2.6: *lmf* dynamics. The Table shows the final *lmf* values, the number of *lmf* changes, and the sequence of actual *lmf* values.

App.	Dataset	Final lmf	# Changes	lmf Values
SSSP	*tw*	0	0	0
	rUSA	13	2	0-12-13
BFS	*tw*	0	0	0
	rUSA	2	1	0-2
PR	*wg*	3	2	0-1-3
	lj	2	2	0-1-2
PR-D	*wg*	10	1	0-10
	lj	9	2	0-1-9
MST	*rW*	3	2	0-2-3
	rUSA	3	3	0-1-2-3
A*	*rW*	16	1	0-16
	rUSA	13	2	0-12-13

of the execution, during the first 1% of `Deq` operations. For example, this happens in SSSP with the *rUSA* input. In this case, PMOD quickly increases *lmf* first to 12, and then to 13, which becomes its final value. However, this is not always the case. For instance, in MST, the first merge occurs only when around 40% of `Deq` operations have been executed.

The number of threads also affects merging. We compare the time of the first merge operation in executions with 40 threads and with 1 thread. Although not shown in Table 2.6, we find that for SSSP on *rUSA* with 1 thread, only about 60 `Deq` operations execute before the first merge. For SSSP with 40 threads, the first merge only occurs after 15-20 K `Deq` operations. For MST, the behavior is different. In both single- and 40-threaded executions, the first merge operation occurs when around 40% of the `Deq` operations have been executed.

2.7 IMPLICATIONS ON COMPUTER ARCHITECTURES

Our analysis provides insights into the bottlenecks of concurrent priority scheduling for graph algorithms in large servers. It is sobering to see that sophisticated skip list-based CPSs are overwhelmed by enqueueing and dequeueing overheads. This is despite employing

scalable data structures that perform searches in parallel without synchronization and avoid synchronization hotspots in updates. We do not believe that hardware support in large NUMA servers should focus on improving synchronization for such CPS designs. Instead, it should focus on improving the obim and PMOD approaches to CPS.

PMOD typically devotes a large fraction of cycles to *GWork*, as shown in Figure 2.13. However, the figure also shows that there are still some cases where *GWork* is a small fraction of the total cycles. PMOD is still sometimes a victim of the fundamental CPS tradeoff: either it suffers from *Deq*/*FDeq* cycles, or from *UWork*. We need to replace PMOD software structures with hardware that frees PMOD from this tradeoff. One example is hardware to make partially-full chunks quickly available to idle threads. Another is fast communication of the work list and the partially-full work list across sockets.

2.8 RELATED WORK

Several graph analytics frameworks [16, 17, 26, 27, 28, 29] have been developed for shared-memory machines. Ligra [17] abstracts away graph traversals through mapping computations over a subset of vertices or edges in parallel. Julienne [27] builds upon Ligra by grouping together similar graph entities, such as vertices, edges, or other graph motifs, into buckets.

Many concurrent priority queues [33, 34, 35, 36, 53] have been introduced for task-based priority scheduling. Techniques such as Flat Combining [54] and Elimination [55] are adopted by Calciu et al. [53] to reduce enqueue/dequeue overheads without compromising on priority constraints. In contrast, relaxed priority schedulers [16, 33, 34, 35, 36] trade off priority constraints for lower synchronization. Lenharth et al. studied the performance of priority queues as graph analytics CPSs [32]. However, they did not consider relaxed priority queues like SprayList or MultiQueues.

There has been much work on algorithm-specific optimizations of different graph problems, e.g., for SSSP [30], BFS [56], and MST [57]. However, our focus is on optimizing generic graph frameworks and not on targeted optimizations.

2.9 CONCLUSIONS

Graph processing frameworks use CPSs to execute tasks largely according to their priority order. CPSs are performance-critical, but there has been little insight on the relative strengths and weaknesses of the different CPS designs. We addressed this question with a detailed empirical performance analysis of four state-of-the-art representative CPS designs

on a 40-core shared-memory machine. We observed that in all CPSs but obim, the overall cost of enqueueing and dequeueing is typically higher than the task execution time. This is despite employing scalable data structures. Further, the obim CPS, which is designed to reduce enqueueing and dequeueing overheads at the expense of sometimes executing useless work, also has limitations. While it typically performs best, it leads to significant slowdowns under some priority distributions. With these insights, we developed the new PMOD CPS. It is based on the obim approach but dynamically adapts to the ranges of priorities exhibited by the application. PMOD is robust and delivers the best performance overall.

CHAPTER 3: SPEEDING UP SPMV FOR POWER-LAW GRAPH ANALYTICS BY ENHANCING LOCALITY & VECTORIZATION

3.1 INTRODUCTION

Graph analytics algorithms are often used to analyze social, web, and e-commerce networks [1, 2, 3, 4]. These networks are typically *power-law graphs* — i.e., their degree distribution follows a power law. In these networks, a small fraction of the vertices have a degree that greatly exceeds the average degree.

Graph algorithms can often be recast as generalized Sparse Matrix-Vector multiplication (SpMV) operations [18, 19, 20]. Examples range from iterative algorithms (e.g., PageRank [3] and HITS [4]) to traversal algorithms (e.g., path/diameter calculations [58]). SpMV-based graph algorithms are faster and have a better multi-core scalability than general graph processing frameworks [58], making SpMV an important kernel to optimize for efficient graph analytics.

Executing SpMV efficiently on real-life power-law graphs is challenging. The reason is that these graphs are large (millions to billions of vertices) and highly irregular, causing the SpMV memory access patterns to have low locality. Moreover, the data-dependent behavior of some accesses makes them hard to predict and optimize for. As a result, SpMV on large power-law graphs becomes memory bound.

To address this challenge, previous work has focused on increasing SpMV's Memory-Level Parallelism (MLP) using vectorization [59, 60] and/or on improving memory access locality by rearranging the order of computation. The main techniques for improving locality are binning [61, 62], which translates indirect memory accesses into efficient sequential accesses, and cache blocking [63], which processes the matrix in blocks sized so that the corresponding vector entries fit in the last-level cache (LLC). However, the efficacy of these approaches on a modern aggressive out-of-order (OOO) processor with wide SIMD operations has not been evaluated.

In this work, we perform such an evaluation using an Intel Skylake-SP processor. We find that, on large power-law graphs, these state-of-the-art approaches are not faster (or only marginally faster) than the standard Compressed Sparse Row (CSR) SpMV implementation. Moreover, these approaches may cause high memory overheads. For example, binning [61, 62] essentially doubles the amount of memory used.

We then propose *Locality-Aware Vectorization* (*LAV*), a new SpMV approach that successfully speeds-up SpMV of power-law graphs on aggressive OOO processors. LAV leverages the graph's power-law structure to extract locality without increasing memory storage.

LAV splits the input matrix into a dense and a sparse portion. The dense portion contains the most heavily populated columns of the input, which—due to the power-law structure—contain most of the nonzero elements. This dense portion is stored in a new vectorization-friendly representation, which allows the memory accesses to enjoy high locality. The sparse portion is processed using the standard CSR algorithm, leveraging the benefits of OOO execution. Overall, LAV achieves an average speedup of 1.5× over CSR and prior optimized schemes across several graph, while reducing the number of DRAM accesses by 35% with only a 3.3% storage overhead.

Contributions. We make the following contributions:

- We analyze existing SpMV approaches with power-law graphs and show their shortcomings in modern OOO processors.
- We show that by using a new combination of known graph preprocessing techniques, we can extract a high-locality dense portion from a sparse power-law matrix.
- We propose LAV, a new SpMV approach that processes the dense portion with vectorization and the sparse portion with the standard CSR algorithm.
- We evaluate LAV with 6 real-world and 9 synthetic graphs, which are at least one order of magnitude larger than those used in the majority of the previous works. We show that LAV (1) consistently and significantly outperforms previous approaches, including CSR; (2) has minimal storage overhead and small format conversion cost; and (3) significantly decreases data movement for all levels of the memory hierarchy.

3.2 BACKGROUND

Every graph G can be represented as an adjacency matrix $\mathbf{A}$, in which element $\mathbf{A}_{i,j}$ is nonzero if there is an edge from vertex i to vertex j. In this work, we therefore use "matrix" and "graph" interchangeably. Real-world power-law graphs typically have many vertices (millions to billions), but most vertices only have relatively few neighbors. Therefore, the adjacency matrices of these graphs are sparse.

Many graph algorithms iteratively update vertex state, which is computed from the states of its neighbors. Each iteration can be implemented with *generalized SpMV* [58], where the multiply and add operations are overloaded to produce different graph algorithms.

Consequently, we consider the SpMV problem of computing $y = \mathbf{A}x$, where $\mathbf{A}$ is the graph and y and x are dense output and input vectors, respectively, representing the set of updated

vertices. The computation of every element of y (for row i of A) is $y_i = \sum_{j=0}^{n-1} \mathbf{A}_{i,j} \cdot x_j$, for 0 $\leq i \leq$ m-1, where m is the dimension of y and n is the dimension of x, i.e., the number of graph vertices.

The elements of $\mathbf{A}$ are read only once, but one can reuse the elements of x and y, whose size typically far exceeds the Last-Level Cache (LLC) capacity. The main challenge is how to reuse elements of x, since the sparseness of $\mathbf{A}$ makes the distribution of accesses to x elements irregular.

3.2.1 CSR Matrix Representation

Large, sparse matrices require a compact in-memory representation. The compressed sparse row format (CSR) (or one of its variants) is the popular choice for graph processing frameworks [16, 17].

In CSR, three arrays are used to represent matrix $\mathbf{A}$: `vals`, `col_id`, and `row_ptr`. The `vals` array stores all of the nonzero elements in matrix $\mathbf{A}$. Within the `vals` array, all the elements in the same row are stored contiguously. The `col_id` array stores the column index of the nonzero elements. The `row_ptr` array stores the starting position in the `vals` and `col_id` arrays of the first nonzero element in each row of matrix $\mathbf{A}$. An example CSR representation is shown in Figure 3.1. In this work, we use CSR as our baseline.

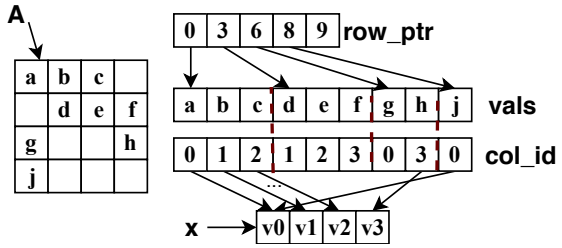


Figure 3.1: CSR format.

Algorithm 3.1 shows a single iteration of CSR SpMV. The algorithm iterates over the matrix $\mathbf{A}$ row by row and calculates the dot product of the row in matrix $\mathbf{A}$ and the input vector x (Lines 3-5). Parallelization is straightforward. Each row can be executed in parallel without the need for synchronization.

3.2.2 Vector Instructions

The Intel Skylake-SP implements the AVX-512 extension, which offers 512-bit vector instructions. This extension allows for 8 double-precision operations (or 16 single-precision op-

Algorithm 3.1 Implementation of CSR SpMV.

```
1: for i←0 to m-1 in parallel do
2:     sum←0
3:     for e ← row_ptr[i] to row_ptr[i+1]-1 do
4:         sum←sum + vals[e] × x[col_id[e]]
5:     end for
6:     y[i]←sum
7: end for
```

erations) to proceed in parallel. Skylake also supports SIMD gather and scatter instructions, which can load/store random elements from/to a given array. Gather/scatter operations are useful for the random accesses performed in SpMV. Masked versions of gather/scatter are also provided, enabling the selective execution of operations on SIMD lanes. Table 3.1 describes the vector instructions used in this work.

Table 3.1: Vector operations used. Instructions with the *_mask* extension allow for selective operations on lanes.

Operation	**Details**
`load[_mask](addr[, mask])`	loads 16 (8) 32-bit (64-bit) packed values to a vector register from addr.
`gather[_mask](ids, addr[, mask])`	gathers 16 (8) 32-bit (64-bit) values from an array, starting at addr, from the indices provided in the ids vector.
`scatter[_mask](ids, addr, vals[, mask])`	scatters 16 (8) 32-bit (64-bit) values in *vals* to the array starting at addr in the indices provided in the ids vector.
`fp_add / fp_mul`	performs the element-wise addition/multiplication of two vector registers.

3.3 PREVIOUS SPMV APPROACHES

The main approaches to improve SpMV performance for large-scale power-law graphs on general-purpose processors are: (1) using vectorization to increase memory-level parallelism (MLP) and (2) improving memory locality and thereby cache effectiveness.

Vectorization can decrease the number of instructions executed, and increase the number of in-flight memory accesses and the floating-point (FP) throughput. There are many proposals for vectorization of SpMV [59, 60, 64, 65, 66, 67].

To improve locality, several graph reordering techniques have been proposed [68, 69, 70]. However, they require time-consuming preprocessing. We instead focus on recent techniques for increasing locality with lightweight preprocessing.

In this work, we focus on the CSR5 [59], binning (BIN) [61], cache blocking (BCOO) [63], and CVR [60] techniques, which are used in the most competitive SpMV algorithms. CSR5

provides efficient vectorization, BIN and BCOO improve locality, and CVR combines both approaches.

CSR5 [59]. CSR5 creates a compact, sparsity-insensitive representation of the input matrix that can be processed efficiently by vector units. CSR5 takes an input matrix in CSR format and partitions the `col_id` and `vals` arrays of CSR into equally-sized small 2D tiles. The size of a tile ($w \times \sigma$) is set as follows: w is set to the number of SIMD lanes, and σ is optimized for the specific architecture. The tiles can be processed in parallel.

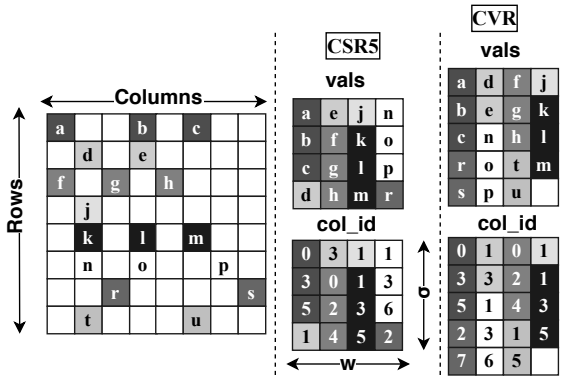


Figure 3.2: Tile layout in CSR5/CVR ($w = \#$ of SIMD lanes). In CSR5, we omit an incomplete tile and metadata for brevity.

Figure 3.2 shows an example of a tile created for a given matrix. A row may end up in multiple tiles (e.g., the 7^{th} row in Figure 3.2 spans multiple tiles). For this reason, CSR5 uses the segmented sum approach to compute the final value for a row. Overall, this structure creates good load balancing across threads and high utilization of SIMD units.

CVR [60]. Compressed Vectorization-oriented sparse Row (CVR) is a compact vectorized representation for SpMV. In CVR, each row of the matrix is processed by a single SIMD lane. Once a SIMD lane finishes processing a row, the next non-empty row from matrix $\mathbf{A}$ to be processed is scheduled on the emptied SIMD lane. In addition, CVR implements a sophisticated work stealing method to balance the SIMD lanes. When there are no more rows left to fill the empty SIMD lanes, an empty SIMD lane will steal elements from non-empty SIMD lanes. Figure 3.2 shows the memory layout for `vals` and `col_id` arrays for CVR.

CVR implements a vectorization mechanism that increases MLP by utilizing vector units efficiently. In addition, it improves locality by scheduling multiple rows to be processed in parallel, which may overlap accesses to a given cache line of x by different SIMD lanes.

BIN [61, 62]. Binning (BIN) always performs sequential or high-locality memory accesses. BIN splits the SpMV execution into two phases. In the first phase, it reads the graph edges sequentially to generate every vertex update (i.e., some $\mathbf{A}_{i,j} \cdot x_j$ that should be added to y_j). The updates are buffered with sequential writes into *bins*, each of which is associated with a cache-fitting fraction of the vertices. In the second phase, BIN applies the updates in each bin. While applying updates results in irregular memory accesses, they target only a cache-fitting fraction of vertices, and so enjoy high locality. Overall, BIN reduces the cycles-

per-instruction (CPI), but executes more instructions, including memory accesses, and needs extra storage for the bins.

Cache Blocking [63]. The idea of cache blocking is to keep r and c elements of the vectors y and x, respectively, cached while an $r \times c$ block of the matrix gets multiplied by this portion of the x vector. We consider BCOO, our hand-optimized implementation of blocking that keeps the x vector portions LLC-resident. BCOO stores the blocks in coordinate format (COO), i.e., as a list of $(row, column, value)$ tuples sorted by row id. To efficiently exploit parallelism, BCOO divides blocks among cores in a row disjoint fashion, which enables writing y vector elements without atomic operations.

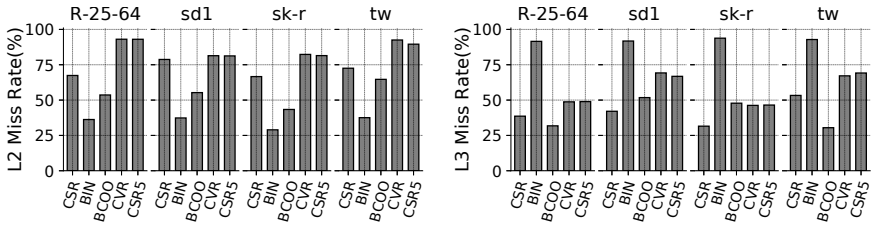
3.4 ANALYSIS OF PRIOR SPMV APPROACHES WITH POWER-LAW GRAPHS

We find that on aggressive OOO processors, the existing techniques described are not faster (or only marginally so) than a simple CSR implementation. We analyze this behavior on a representative Intel Skylake-SP processor. In our experiments, we use three real-world graphs, *sd1*, *sk*, and *tw*, and a large random graph, *R*-25-64, with 2^{25} nodes and ≈64 average degree. Section 3.6.3 describes these graphs.

1. CSR5 and CVR benefit only from limited amounts of locality due to relying on the input's layout. These two techniques take the input matrix and build special matrix representations with the goal of keeping SIMD lanes busy. However, the amount of locality exhibited by the memory reads in these representations depends on the structure of the input matrix. Sadly, we find that the locality resulting from the original input matrix is often insufficient.

CSR5 [59] partitions the sequence of all nonzero elements in a CSR matrix (i.e., the `vals` and `col_id` arrays) into 2D tiles of the same size. The tiles are populated based on the row-major order of appearance of the nonzero elements within the input matrix. As a result, CSR5 does not guarantee high locality for accesses to the x vector. For example, a row with many nonzeros may be spread over several tiles. Also, a row may occupy multiple SIMD lanes in a tile. In both cases, the accesses are determined by the graph structure, which may not exhibit any spatial or temporal locality in terms of accesses to columns (x vector). Even if there are multiple rows in a single tile, they may access disjoint sets of elements from the x vector and, therefore, fail to exploit spatial locality.

In CVR [60], each row of the matrix is processed to completion by a single SIMD lane. Once a lane finishes a row, it begins processing the next non-empty row. As a result, CVR can have two lanes processing columns that are far apart (e.g., columns at the start of one



(a) L2 Miss Rate (b) L3 Miss Rate

Figure 3.3: L2 and L3 miss rates for different techniques.

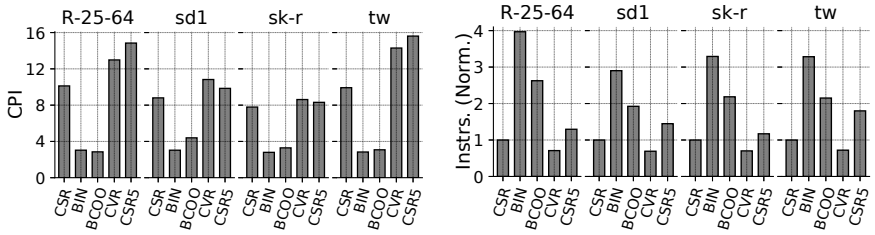
row and at the end of another row). The result will be poor locality. However, if the overlap between the rows is high, we can observe good locality.

Figure 3.3 shows the miss rates of L2 and L3 caches in all of the techniques. We see that the miss rates of CVR and CSR5 are higher than in CSR. Hence, CVR and CSR5 do not improve locality over CSR.

2. The locality-aware techniques improve cache locality but have a limited impact on performance due to the increased number of instructions. Binning [61, 62] and cache blocking (BCOO) target large graphs and try to improve locality by either regularizing memory accesses or restricting them to fit in the LLC. We observe, however, that on a modern OOO processor, these techniques do not reduce execution time over CSR significantly, even though they reduce the miss rate of the L2 and, in the case of BCOO, of the L3 (Figure 3.3), and the average cycles per instruction (CPI). The reason is that they execute more instructions. Furthermore, for BIN, the amount of data moved from memory is similar to the other techniques.

Figure 3.4 shows the CPI (a) and the instruction count (b) of the different techniques. We see that BIN decreases the CPI substantially compared to most of the other techniques. However, it is the technique with the most instructions executed. BIN was effective in previous generations of OOO processors that did not do a good job at hiding memory latency because their Reorder Buffer (ROB) and Load Queue (LQ) were smaller. Thus, the CPI gains were more significant, and BIN was able to tolerate the increase in the number of instructions. On the other hand, BCOO improves the CPI like BIN while increasing the number of instructions less. As a result, it is able to improve performance marginally.

3. Vectorization provides limited MLP improvements since modern OOO cores keep a high number of memory requests in flight due to their deep ROB, LQ, and other buffers. In an SpMV iteration, we observe that most reads of x incur cache misses. The reason is that the access pattern is irregular, and the vector size exceeds the capacity of the LLC. Because of this bottleneck, in an in-order or narrow issue processor



(a) CPI (b) Instructions

Figure 3.4: CPI and number of instructions executed for different techniques.

such as Intel's Xeon Phi, vectorization is effective at increasing MLP: a vector read issues several memory reads concurrently, whereas a scalar narrow-issue core can only issue very few reads at a time. However, modern OOO processors extract significant instruction-level parallelism (ILP) from the SpMV code and so can sustain over ten in-flight memory accesses at a time. Hence, vectorization of the code does not significantly increase the number of memory accesses in flight.

As an indicator of MLP, we measure the average occupancy of the Line Fill Buffer (LFB) in L1 for CSR and the vectorization techniques. The LFB is an internal buffer that the CPU uses to track outstanding cache misses; each LFB entry is sometimes called a Miss Status Handling Register or MSHR. We compare CSR (which is a scalar approach) to CVR and CSR5 (which are vector approaches). The result is shown in Figure 3.5. The maximum LFB occupancy for the machine measured is 12 [71]. The figure shows that the LFB occupancy of CSR is already very similar to the one of the vectorized CVR and CSR5. Consequently, one does not need vectorization to attain high MLP on an aggressive OOO processor.

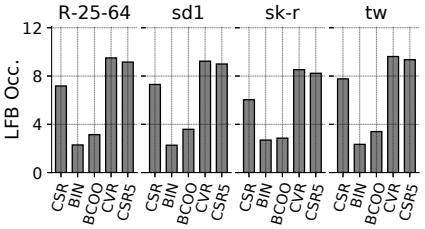


Figure 3.5: LFB occupancy for different techniques.

3.5 LAV: LOCALITY-AWARE VECTORIZATION

The previous observations showed that, on modern OOO processors, the bottleneck of current SpMV techniques is not low MLP. Instead, we observe that current techniques either do not exploit locality or are only able to extract locality by increasing the number of instructions executed, resulting in little or no overall gains.

Based on these observations, we propose *Locality-Aware Vectorization* (*LAV*), a new approach that speeds-up SpMV by harvesting locality from power-law graphs without increas-

ing the number of instructions. LAV relies on a combination of lightweight preprocessing steps that create a compact matrix representation.

In this section, we describe the main idea, the preprocessing steps, the matrix representation, the rationale behind the design choices, and LAV's SpMV algorithm.

3.5.1 Main Idea

To improve locality while keeping a compact data representation, LAV takes the input matrix A and divides it into two portions. The first one, called the *Dense* portion, has the majority of the nonzero elements. The second one, called the *Sparse* portion, includes the remaining nonzero elements. Operations on the dense and sparse portions of the matrix access disjoint subsets of elements from the input vector x. LAV uses different formats and processing mechanisms for the dense and sparse portions. The dense portion is formatted for locality and processed with a vectorized SpMV implementation. The sparse portion is formatted for compactness and processed with a CSR-based SpMV implementation.

Creating the dense portion for locality. The dense portion contains most of the nonzero elements in matrix **A**. It is obtained by picking the most heavily populated columns of **A** until, all together, they account for a fraction T of the nonzero elements of **A** (e.g., 80% of the nonzero elements). For our target real-world power-law graphs, where the degrees of the vertices follow a power-law distribution [72, 73], this dense submatrix will only contain a small fraction of the original columns (e.g., 20% of the columns) [74]. In addition, the columns are then divided into *segments*, each of which fits into the LLC. As we will see, the data in the dense portion is stored in a compact, vectorization-friendly format that allows for fast processing.

Creating the sparse portion for compactness. The sparse portion is the rest of the columns. The data is stored in the CSR representation, which is compact. The data is processed with the standard CSR implementation since we do not expect to obtain locality benefits during its processing.

3.5.2 Splitting the Matrix into Dense and Sparse Portions

To split the matrix into dense and sparse portions, LAV uses the following three simple matrix transformations.

1. Column Frequency Sorting (CFS). CFS sorts the columns in descending order of nonzero element count, changing the physical layout of the matrix **A** and input vector x.

Since the degree distribution of the vertices in our target graphs follows a power law, the first few columns of **A** will include the vast majority of nonzero elements. Thus, CFS's reorganization of **A**'s columns results in *frequently accessed elements of x being stored in the same, or close by, cache lines*. The result is improved cache line and overall cache utilization, which speeds-up computations [75].

To sort the columns, CFS logically moves them by relabeling the column IDs according to the number of nonzeros in descending order. CFS permutes the x vector accordingly.

2. Segmenting. Segmenting takes the output of CFS and partitions matrix **A** into dense and sparse portions. The dense portion is further divided into *segments* of consecutive columns. Each segment contains S columns, where S is chosen so that the corresponding part of the input vector x fits in the LLC. (For example, we use $S = 5\,M$ in our evaluation; see Section 3.7.3.) The number of segments s in the dense portion is chosen so that the dense portion includes a fraction T (e.g., 80%) of all the nonzero elements in **A**. Segmenting can be efficiently implemented if combined with CFS. CFS already generates a sorted list of columns according to their count of nonzero elements. After this list is composed, segmenting creates the segments.

3. Row Frequency Sorting (RFS). RFS sorts the rows in a segment in decreasing count of nonzero elements in the row. The resulting order will determine the order of execution of rows in the segment. With this change, we will be able to execute rows with similar numbers of nonzero elements at the same time in different SIMD lanes. As a result, LAV minimizes load imbalance between different SIMD lanes.

RFS does not actually relabel the rows in a segment. Instead, it only generates a list of row IDs for the execution order. Hence, it is a very lightweight operation.

Parallelizing LAV's transformations. All of the steps above can be efficiently parallelized. Specifically, CFS and RFS require counting the number of nonzero elements in each column and row, respectively. These counts can be computed in parallel. Our current implementation uses atomic operations to safely update counts in parallel, but techniques exist [76] that would eliminate the use of atomics. In practice, the number of nonzeros in each column/row is often available without computing, since many graph analytics frameworks [17] store the matrix in both CSR and CSC formats. In this case, the number of nonzeros can be obtained from the offset arrays.

Once the nonzero element counts are known, relabeling the columns for CFS simply requires parallel sorting. Segmenting requires a parallel prefix sum on the relabeled columns to determine the cut-off point for the number of columns per segment. RFS simply requires sorting the row IDs in decreasing count of nonzeros.

3.5.3 LAV Walk-Through

Figure 3.6 shows an example of LAV's transformations. An initial matrix **A** is shown in Figure 3.6a. To highlight the effect of CFS, the x vector is shown on top of matrix **A**. As Figure 3.6a shows, initially, the nonzero elements of **A** are scattered over the rows and columns.

The first step performs CFS on **A**, relabeling all the column IDs with the new order. This is shown in Figure 3.6b. Note that the input vector x is permuted according to the new order of the columns. As shown in Figure 3.6b, the majority of the nonzero elements now reside in the first few columns.

The second step divides **A** into segments of a fixed number of columns. The number of columns per segment, S, is maximized under the constraint that the corresponding entries of the input vector fit into the LLC. The matrix is then partitioned into dense and sparse portions. The dense portion consists of the first s segments that contain a T fraction of the nonzero elements. The sparse portion contains the remaining columns. Figure 3.6c shows the dense and sparse portions of **A** with $S = 4$ and $T = 0.7$, where a single segment suffices to obtain the dense portion.

The next step applies RFS to each of the segments of the dense portion. This is shown in Figure 3.6d. Note that the output vector y is permuted according to the new order of the rows. We will later see that neither the matrix nor the y vector is physically reordered; instead, LAV creates an indirection vector to record the reordering.

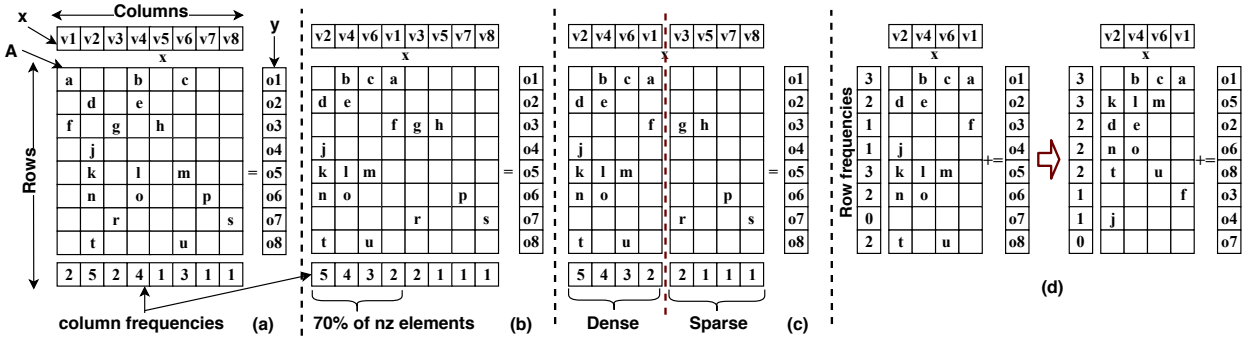


Figure 3.6: LAV's transformations on an 8×8 matrix. In the example, the threshold for creating the dense portion is T=0.7 and the segment size is S=4.

3.5.4 LAV Matrix Representation

LAV's dense portion is composed of segments of consecutive columns. After RFS, each segment is divided into *chunks* of rows, where each chunk has as many rows as the number

of SIMD lanes in the machine. Figure 3.7 continues the example of Figure 3.6 and shows, on its left side, the segment divided into two chunks. We use four rows per chunk because we assume four SIMD lanes. The two chunks have different colors.

Each row is next compressed such that zero elements are not stored. Since the different rows of a given chunk may have different numbers of nonzero elements, some SIMD lanes will remain empty towards the end as they run out of nonzero elements. To handle this case, LAV creates a mask for each SIMD lane, expressing the empty elements in the lane with zeros. To represent a segment, LAV repeats the same operation on all the chunks of the segment and appends them together.

The right side of Figure 3.7 shows the memory layout of the segment for the example. A segment in LAV is represented by five arrays: `vals`, `col_id`, `chunk_offsets`, `out_order`, and `mask`.

The `vals` and `col_id` arrays hold the values and column IDs, respectively, of all the chunks in the segment. These are 2D arrays whose one dimension is equal to the number of SIMD lanes, and the other is the sum of the lengths of the chunks of the segment. The length of a chunk is equal to the maximum number of nonzero elements in any row of the chunk. This layout of `vals` and `col_id` is efficient because it will enable using aligned vector loads.

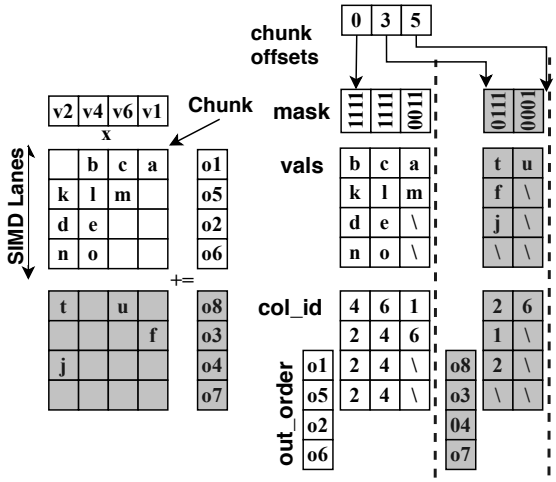


Figure 3.7: Vectorization-friendly layout of the dense portion in LAV. The example assumes 4 SIMD lanes.

The `chunk_offsets` array indicates the starting point of every chunk within the `vals` and `col_id` arrays. When a chunk finishes execution, the partial sums generated by the rows are accumulated into the correct position in the output vector y.

Furthermore, with RFS, LAV generates a new execution order for each segment. This execution order is stored in the `out_order` array. The `out_order` array stores only the IDs of the rows. Finally, some SIMD lanes in a chunk may be empty. Hence, LAV has an extra array called `mask` to distinguish empty elements in the `vals` and `col_id` arrays.

3.5.5 Rationale Behind LAV's Design Choices

We now explain the rationale behind LAV's design choices.

Reuse Memory. LAV uses CFS to group columns with a high number of nonzeros together.

Combined with segmenting, this transformation limits the memory footprint required to process the dense portion. The dense portion will be processed one segment at a time. Thus, by sizing a segment appropriately, we ensure that the portion of the x vector that is accessed while processing one segment fits in the LLC.

Balance SIMD Lanes. LAV uses RFS in segments to balance the SIMD lanes. Recall that each lane calculates the results for a single row. In a given chunk, consecutive rows have a similar number of nonzero elements. Thus, while processing this chunk, all SIMD lanes have roughly the same amount of work. This approach avoids computation load imbalance.

Combining CFS and RFS increases the possibility of memory access overlap. Indeed, rows that are scheduled together on different lanes often access the same (or close by) elements of the x vector.

We also apply RFS to the sparse portion. This way, we find empty rows at the end and do not need to process them.

Handle Sparse Portion Efficiently. Using CSR for the sparse portion minimizes the number of instructions needed per irregular access and the storage cost. The fewer bookkeeping instructions enable the processor pipeline to keep more memory accesses in flight simultaneously, thereby maximizing MLP.

3.5.6 SpMV Algorithm with LAV

The LAV algorithm first processes the dense portion of the matrix and then the sparse one. Algorithm 3.2 shows the steps taken to process the dense portion. The sparse portion is processed with the traditional CSR algorithm.

Algorithm 3.2 works by operating on one segment at a time (Line 8), processing all the nonempty chunks in the segment in parallel, with multiple threads (Line 10). While processing a chunk, LAV utilizes wide SIMD units. The rows in the chunk are processed in parallel by SIMD lanes, and the partial sums generated by the rows are accumulated into a vector register (`row_sums` in Lines 21-28). The `mask` values produced during the data layout generation (Section 3.5.4) are used to disable computation for lanes with empty elements. After the completion of the chunk, the output values that it has generated are accumulated into the output vector y. This operation involves reading the accumulated values up until this segment in output vector y (Line 20), and adding to them the contributions of this chunk (Lines 30-32).

Finally, there is the case when the number of rows in the last chunk of a segment is less than the number of SIMD lanes. In this case, we calculate a mask (`row_mask`) that is used

to omit all the operations in the idle SIMD lanes. This operation is shown in Lines 14-16 of Algorithm 3.2.

Algorithm 3.2 SpMV algorithm with LAV.

```
1: procedure SpMV(segments, x, y, m, n, nLanes)
2:     // segments is the list of segments created for vectorization
3:     // x is the input vector, y is the output vector
4:     // m number of rows, n number of cols, nLanes number of SIMD lanes
5:     for r=0 to m-1 do  // initialize y vector to zeros
6:         y[r] ← 0
7:     end for
8:     foreach segment s in segments do  // all segs. in dense portion
9:         // chunks of a segment processed in parallel
10:        for c=0 to s.num_chunks-1 in parallel  do
11:            row_sums←(0, . . . , 0)
12:            row_mask←0xFF
13:            // last chunk may have all the lines completely idle
14:            if c is the last chunk in the segment then
15:                row_mask ← 0xFF >> (nLanes-(m - c*nLanes))
16:            end if
17:            // Get the rows to be updated by this chunk
18:            row_ids←load_mask(s.out_order[c*nLanes], row_mask)
19:            // Get the prev values from y for rows of this chunk
20:            prev←gather_mask(row_ids, &y, row_mask)
21:            for i← s.chunk_offsets[c] to s.chunk_offsets[c+1]-1  do
22:                ms←s.mask[i] // load the mask for the SIMD lane
23:                cols←load_mask(s.col_ids[i*nLanes], ms)  // get column ids
24:                vals←load_mask(s.vals[i*nLanes]], ms) // get vals
25:                x_vec←gather_mask(cols, &x, ms) // get vals from x
26:                mul←fp_mul(vals, x_vec)
27:                row_sums←fp_add(row_sums, mul)
28:            end for
29:            // accumulate sum calculated by chunk
30:            row_sums←fp_add(row_sums, prev) // Aggregate over segs.
31:            // update y vector
32:            scatter_mask(row_ids, &y, row_sums, row_mask)
33:        end for
34:    end foreach
35:    // CSR for the sparse portion
36: end procedure
```

3.6 EXPERIMENTAL SETUP

3.6.1 Test Environment

We use a state-of-the-art Intel Skylake-SP shared-memory machine with vector instruction support. Our machine has 2 processors, each with 20 2.4 GHz cores, for a total of 40 cores. Each core is OOO and has a private 32 KB L1 data cache and a private 1 MB L2 cache. Each processor has a shared 28 MB LLC. The machine has 192 GB of 2666 MHz DDR4 main memory, distributed across 12 DIMMs of 16 GB each. Details of our system are summarized in Table 3.2.

We use Ubuntu Linux 14.04 with kernel version 4.9.5. All codes are compiled with the

Table 3.2: System characteristics.

Component	Characteristics
CPU	Intel Xeon Gold 6148 CPU @ 2.40GHz 20 cores per processor, 2 processors
Cache	Private 32 KB instruction and data L1 caches Private 1 MB L2 cache, 28 MB LLC per processor
Memory	192 GB, 12 DIMMs (16 GB each), DDR4 2666 MHz
Vector ISA	avx512f, avx512dq, avx512cd, avx512bw, avx512vl

Intel compiler version 19.0.3, using the *-O3 -xCORE-AVX512* compiler flags, and parallelization is done with OpenMP. When implementing LAV, we use intrinsics provided by the Intel compiler for vectorization. CVR, CSR5, and LAV implementations use 512-bit vector instructions. All SpMV implementations use double-precision floating-point arithmetic. We use *numactl* to interleave allocated memory pages across the NUMA nodes. Dynamic voltage and frequency scaling (TurboBoost) is disabled during the experiments.

In all the experiments, 40 threads are used, and each thread is pinned to a core. In the scalability experiments (Section 3.7.5), when we have fewer threads than cores, we assign threads to the two sockets in a round-robin manner. All experiments run 100 iterations of SpMV. Unless stated otherwise, execution time only includes execution of SpMV. For microarchitectural analysis, we use the LIKWID performance monitoring tool [77].

3.6.2 Formats and Techniques for Comparison

We compare LAV to CSR, CSR5 [59], CVR [60], BCOO [63], and BIN [61] (Section 3.3). We also compare to the inspector-executor SpMV implementation in Intel's Math Kernel Library (MKL [78]), which optimizes the SpMV execution according to the input's sparsity level.

We implement CSR, BCOO, and BIN ourselves; for BIN, we closely follow [61]. We use the CSR5 [79] and CVR [80] implementations provided by their authors.

Optimizations. Our compiler automatically vectorizes the baseline scalar CSR code. We manually tune CSR's OpenMP scheduling parameters. For CSR5, we use the tile size suggested in [59]. CVR does not have any tuning parameters. For BIN, we apply the optimizations in [61]. For instance, the target bin and position in the bin of each update are not calculated at runtime but stored with the matrix. To represent bins, we use the optimization from [62]. Each thread works on a sequential portion of each bin. We find that the best bin size is 65 K rows by sweeping bin sizes from 1 K to 262 K. With this bin size, each portion of the y vector approximately fits into the L2 cache. For BCOO, we try to divide the work equally among threads in a row disjoint manner (see Section 3.3). We also manually tune

the size of the cache block and find that blocks that include 2.5 M vertices give the best performance. Finally, for LAV, we use 5 M as the segment size (S) and 80% as the fraction of nonzeros in the dense segment (T).

3.6.3 Input Graphs

We use 6 real-world graphs and 9 synthetic power-law graphs. Table 3.3 summarizes the properties of the graphs.

Table 3.3: Input graphs.

Input	**#rows (M)**	**#nnz (M)**	**Avg. #nnz per row**	**Max. #nnz per row (K)**
fr[81, 82]	65.61	3,612.13	55.06	5.21
pld[83]	42.89	623.06	14.53	3,898.56
sd1[83]	94.95	1,937.49	20.41	1,309.80
sd[83]	101.72	2,043.20	20.09	1,317.32
sk ***(sk-r)***[69, 82]	50.64	1,949.41	38.50	12.87
tw[82, 84]	41.65	1,468.37	35.25	2,997.47
R-*X*-*Y*	2^X	$\approx 2^X \times Y$	$\approx Y$	200-1K
kron[82, 85]	134.22	4,223.26	31.47	1,572.84

Real-world graphs. We use the following large graphs with power-law degree distributions [72, 73]: com-Friendster (*fr*), Pay-Level Domain graph (*pld*), 1st Subdomain graph (*sd1*), Subdomain/Host graph (*sd*), sk-2005 (*sk*), and twitter7 (*tw*). We obtain *fr*, *sk*, and *tw* from [82], *pld*, *sd1*, and *sd* from [83]. The number of vertices (or rows or columns) per graph ranges from 41 M to 101 M. In prior work, only BIN was evaluated with graphs of these sizes; CSR5 and CVR used orders of magnitude smaller graphs. The graphs are also very sparse. The number of nonzero elements (#nnz) ranges from 623 M to 3.6 B. The average number of nonzero elements per row (i.e., degree) ranges from 14 to 55, and the maximum number ranges from 5.2 K to 3.9 M.

We observe that, unlike for all the other inputs, SpMV for *sk* has high L1 hit rates. For this reason, we also evaluate *sk-r*, a version of *sk* that randomized the vertex IDs.

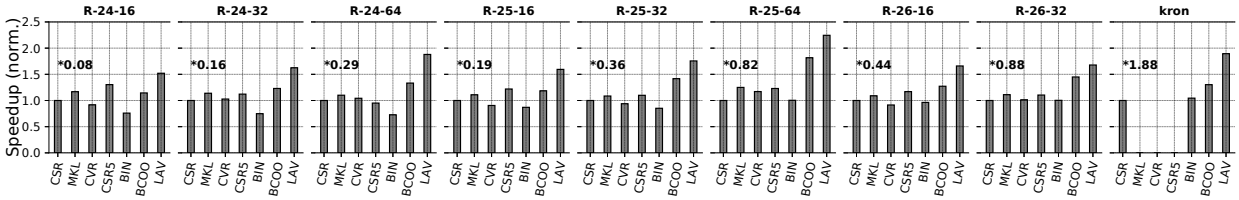
Synthetic graphs. We generate 8 synthetic Kronecker graphs that show power-law behavior, as in the inputs of the Graph500 benchmark [86]. They are named R-X-Y, where X is the scale of the graph (i.e., the number of vertices is 2^X) and Y is the approximate average degree. The number of vertices for these graphs varies between 16 M and 67 M, and the average number of edges per vertex is 16–64. These graphs provide insight into the

performance of the evaluated techniques for different graph sizes and sparsity levels. We also use *kron*, a synthetically generated graph in the GAP benchmark suite [85].

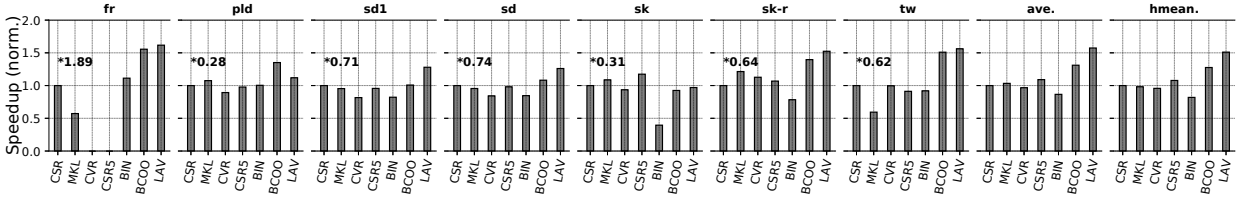
3.7 EXPERIMENTAL RESULTS

3.7.1 Performance Results

Figure 3.8 compares the speedup of MKL, CVR, CSR5, BIN, BCOO, and LAV over CSR, running a single SpMV iteration. The figure shows bars for each synthetic input graph (a) and for each real-world graph plus the arithmetic and harmonic means across all the graphs (b). The number marked with * above the bars is the execution time of a single SpMV iteration with CSR in seconds. For the *kron* and *fr* inputs, we are unable to evaluate CVR and CSR5 because their implementations use 32-bit indices, and these two large graphs cannot be represented. MKL with *kron* runs out of memory.



(a) Synthetic graphs



(b) Real-world graphs plus arithmetic and harmonic means

Figure 3.8: Comparison of the speedups of the different implementations over CSR for all the input graphs. The numbers marked with * show the execution time in seconds of a single SpMV iteration with CSR.

LAV delivers an average speedup of 1.5× over our optimized CSR. On the other hand, CVR, CSR5, and BIN provide comparable average speedups as CSR. As discussed in Section 3.4, these vectorization and locality-optimized approaches are not effective on our aggressive OOO processor, which has an increased ability to hide memory latency. MKL is also not faster on average than CSR. BCOO is the only technique that is faster than CSR on average, achieving a 1.28× speedup.

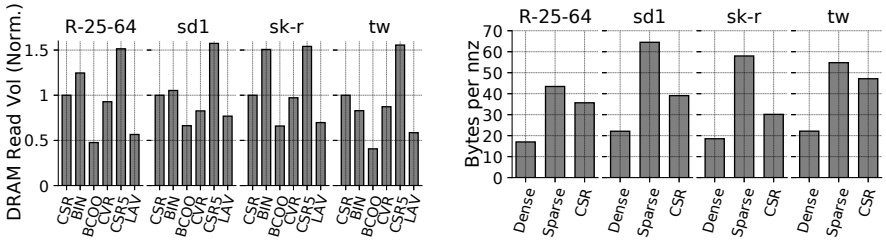
Generally, LAV's speedup over CSR increases as the input's average degree increases, which makes LAV's dense portion contain larger fractions of the nonzero elements. Overall, LAV is the fastest in 14 out of 16 inputs. One of the inputs where CSR outperforms LAV is *sk*. *sk* shows good locality behavior even for the baseline CSR, making the locality optimization techniques not effective. Even so, LAV does not hurt performance compared to CSR on this input. For *pld*, LAV outperforms CSR and other methods except BCOO.

LAV is designed to improve locality, i.e., to minimize data movement from memory and throughout the cache hierarchy. The two causes for data movement are: (1) the accesses to the nonzero elements of matrix $\mathbf{A}$, and (2) the irregular accesses to the x vector. Nonzero element accesses cause data movement proportional to the storage size of the SpMV technique since each nonzero element is accessed once in a single SpMV iteration. Data movement due to x vector accesses is dictated by the locality of the accesses and, hence, the amount of cache reuse. LAV improves in both reduced storage size and increased locality of x vector accesses. LAV reduces storage size by creating a compact representation using RFS; LAV increases the locality of x vector accesses by using CFS and segmenting.

3.7.2 Data Movement

To understand the data movement characteristics of the evaluated techniques, we measure the total volume of data read from DRAM for each input. It can be shown that none of the techniques reaches the system's maximal memory bandwidth (reported by the STREAM benchmark [87]) for any of the inputs.

Figure 3.9a shows the amount of data read from DRAM in four representative inputs normalized to CSR. We can see that both LAV and BCOO substantially decrease the DRAM transfer volume. The reduction achieved by LAV is 35% on average and can be as high as 50%.



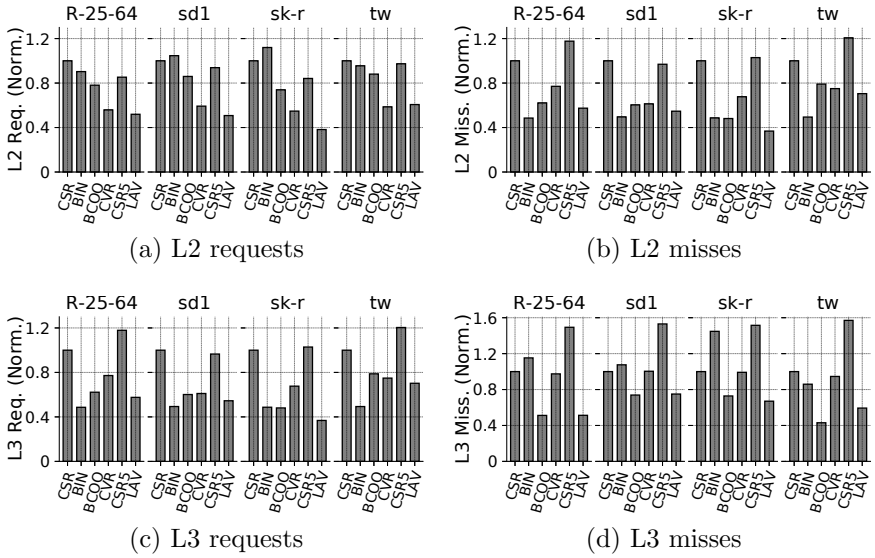
(a) Data read from DRAM (b) Bytes read per nonzero

Figure 3.9: Data transfer comparison.

To analyze the benefits of LAV's dense portion, Figure 3.9b shows the number of bytes

read per nonzero element for LAV's dense and sparse portions, and for CSR. LAV's dense portion keeps the bytes transferred per nonzero element very low. It can be shown that, across all inputs, LAV's dense portion transfers an average of 27.3 bytes per nonzero element, compared to CSR's 37.4. In contrast, LAV's sparse portion transfers more bytes per nonzero element than CSR. This is because the sparse portion is sparser than the overall graph. However, this overhead does not impact execution time much because the sparse portion is—by design—only a small portion of the matrix.

Figure 3.10 analyzes the data movement in the cache hierarchy for selected inputs. The figure shows the number of L2 and L3 requests and misses of the techniques normalized to CSR. From the figures, we see that LAV minimizes the requests and misses in both levels of the cache hierarchy. In fact, it is the only technique that minimizes data movement across all levels of the memory hierarchy. This is because it does not increase the memory storage for the matrix, and it improves the locality of accesses to x. (In Section 3.7.4, we further discuss LAV's storage overhead and show that it is negligible.)



(a) L2 requests (b) L2 misses

(c) L3 requests (d) L3 misses

Figure 3.10: L2 and L3 request and miss counts.

The other techniques do not reduce data movement as much. Specifically, CSR5 and BIN suffer from increased storage. CSR5 increases data movement from memory to L3 and from L2 to L3. This is because it has auxiliary data to facilitate efficient vector execution. BIN improves L2 locality but has a high L3 cache miss rate. It ends up with same number of L3 misses or more than in CSR. CVR has reduced the data movement from L3 to L2. However, it has the same amount of data movement as CSR from memory to L3. BCOO has low data movement between memory and L3, and between L3 and L2. However, it has more L2 requests than LAV.

3.7.3 Analysis of LAV

Dense vs. sparse portions. Figure 3.11 shows the breakdown of execution time and the number of nonzero elements for LAV's dense and sparse portions. By construction, the dense portion contains at least 80% of the nonzero elements. However, its size is also dictated by the size of the segments from which it is composed. The segment size is such that the

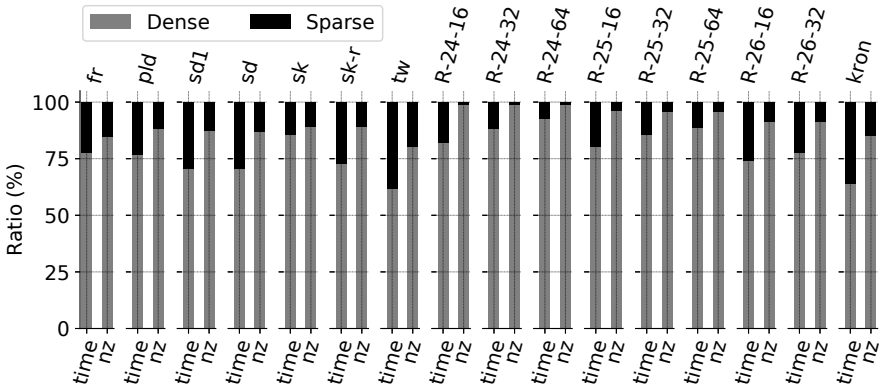


Figure 3.11: Percentage of execution time and nonzero elements in LAV's dense and sparse portions.

corresponding portion of the x vector is roughly equal to the L3 size (5 M columns in our case). Therefore, the last segment may push the number of nonzero elements in the dense portion above 80%. In particular, when the number of vertices is small (e.g., the R-24-XX and R-25-XX graphs), the dense portion contains more than 95% of the vertices.

The execution times of the dense and sparse portions are not proportional to their number of nonzero elements. Specifically, the sparse portion's execution time is often higher than its share of the nonzero elements. Although the percentage of nonzero elements in the sparse portion is less than 20% in all cases, the sparse portion execution can take up to 38% of the execution time. The reason is that the sparse portion does not enjoy the dense portion's high locality and consequently requires more data from DRAM per each nonzero element (Figure 3.9b).

Finding the best segment size. LAV can use LLC-sized segments without tuning. To show this, we vary the segment size for selected inputs and show that a segment size similar to the LLC size produces good results. Figure 3.12 shows the execution time of LAV for different segment sizes. We test segment sizes from 2 to 10 million columns. As can be seen from the figure, the best performance is achieved between 2 M and 5 M columns, and the differences in this range are not significant. The 2 M–5 M column range roughly corresponds to 16 MB–40 MB storage, which is approximately our LLC size. In our experiments, we use 5 M as the segment size.

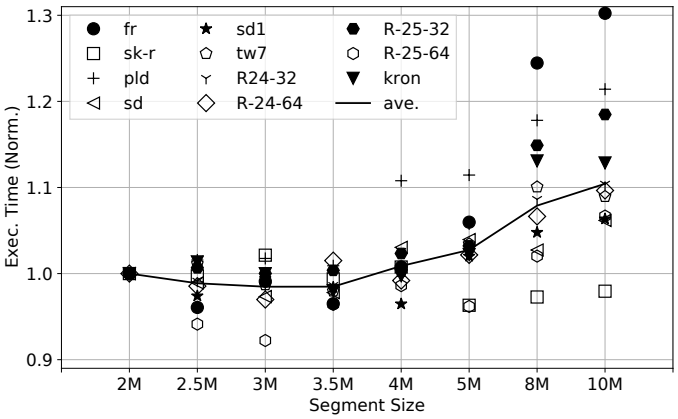


Figure 3.12: Effect of the segment size in number of columns on execution time. Execution times are normalized to a 2M-column segment size.

3.7.4 LAV Overhead

Table 3.4 reports the overhead of constructing the LAV matrix representation, in terms of time and memory storage, compared to constructing a graph's CSR representation.

Memory Overhead. LAV's storage is compact, thanks to segmenting, which limits the number of columns processed at a time, and prevents rows in the dense portion from having a large number of zero elements. As shown in the table, the memory overhead over CSR is small. On average, it is 3.35%.

Format construction time. On average, constructing the LAV's format takes 1.6× the time of building a CSR representation from triplets. Both CSR and LAV initially sort the input's edges by column ID and row ID to construct the matrix rows. Sorting dominates the execution time. In addition, LAV requires computing the number of nonzero elements per column, which has significant overhead.

LAV's higher construction time pays for itself by yielding a faster SpMV execution. Table 3.4 evaluates this tradeoff by showing the number of LAV SpMV iterations required to make LAV faster than CSR (i.e., we are comparing construction plus SpMV execution times). We consider three possible starting points for constructing LAV. First, from an unordered list of nonzeros elements, which is the common textual representation of edge lists. Second, from a CSR representation. Third, when given both CSR and CSC representations. The latter setup exists in graph frameworks that use pull/push-based graph processing, which require both the original matrix and its transpose (for the pull and push directions, respectively).

For the large majority of the graphs used in this work, we see that 10–70 SpMV iterations are enough for the end-to-end execution time of LAV to be better than CSR, irrespective

Table 3.4: LAV memory overhead and number of iterations to amortize LAV's cost.

Input	Number of Iterations			Memory
	Unordered	CSR	CSR+CSC	Overhead
fr	14.10	22.13	11.78	3.12%
pld	81.12	122.64	82.98	8.52%
sd1	40.85	61.84	39.81	4.94%
sd	71.27	70.02	45.70	4.99%
sk	∞	∞	∞	1.98%
sk-r	13.51	31.86	17.85	1.98%
tw	25.63	35.72	21.07	2.25%
R-*24-16*	37.34	49.08	31.66	4.09%
R-*24-32*	46.16	41.71	24.96	2.80%
R-*24-64*	37.88	33.43	18.66	2.09%
R-*25-16*	42.94	37.54	23.29	3.89%
R-*25-32*	42.42	32.75	17.55	2.62%
R-*25-64*	19.98	23.91	14.06	1.93%
R-*26-16*	34.00	29.88	18.40	3.70%
R-*26-32*	28.55	28.82	16.25	2.44%
kron	18.54	21.78	13.15	2.31%

of how the LAV's format is constructed. We believe this overhead is acceptable since these matrices are generally used as inputs for multiple algorithms, or many iterations of SpMV are executed in each application.

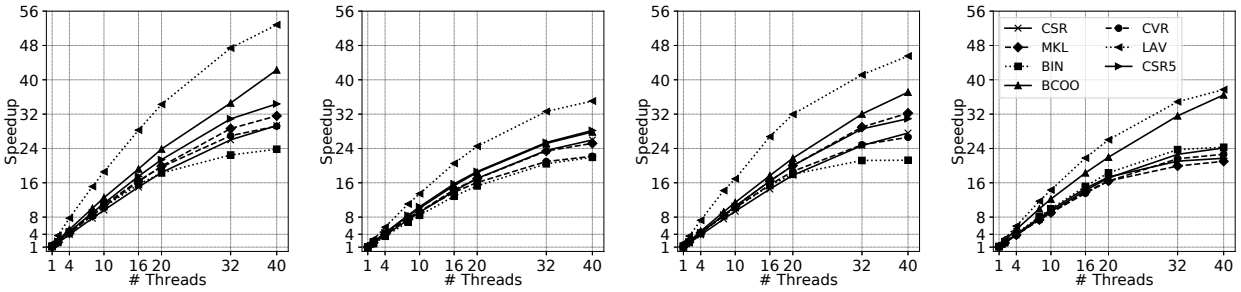
The only exceptions are *sk* and *pld*. In *sk*, a LAV SpMV iteration is slower than a CSR SpMV iteration. Hence, LAV cannot catch up with CSR. In *pld*, the difference in execution time per SpMV iteration is small. Therefore, LAV takes 81–123 iterations to be faster than CSR.

3.7.5 Scalability Analysis

Strong scaling is defined as scalability with respect to a fixed problem size. Figure 3.13 compares the strong scaling of LAV to the other techniques over a representative subset of the input graphs (*R*-25-32, *sd1*, *sk-r*, and *tw*). All curves show speedup values for different numbers of threads, normalized to the single-threaded CSR execution time.

Overall, LAV gives the same or better performance compared to the other techniques at every thread count, except on the *pld* and *sk* inputs (not shown in Figure 3.13). For *pld*, LAV and BCOO perform similarly up to 16 threads, but BCOO outperforms LAV subsequently, achieving a 20% higher speedup for 40 threads. For *sk*, all techniques have comparable speedups (10–12× for 40 threads), with MKL being the best performer for all thread counts.

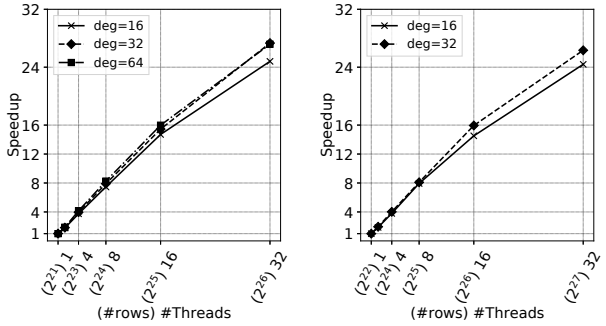
LAV's speedups vary greatly across the different input graphs. For example, while LAV achieves a 53× speedup at 40 threads on *R*-25-32, it only achieves 35× on *sd1*. This



(a) R-25-32 (b) sd1 (c) sk-r (d) tw

Figure 3.13: Strong scaling of LAV and other techniques. All points are normalized to the single threaded CSR execution.

variability is expected since the computation's memory access pattern depends on the input graph, and so the benefits from CFS and segmenting differ greatly. The maximum speedups achieved by LAV, BCOO, MKL, CSR, CSR5, CVR, and BIN with 40 threads are 53×, 42×, 35×, 31×, 31×, 29×, and 26×, respectively. If single-threaded executions are considered, LAV is still the best performer. On average, LAV achieves 1.6× speedup over CSR, while CVR, CSR5 BCOO, BIN, and MKL only attain 1.17×, 1.09× 1.07×, 1.05×, and 1.01×, respectively.



(a) Per-thread work: 2^{21} rows (b) Per-thread work: 2^{22} rows

Figure 3.14: Weak scaling of LAV with random graphs.

We now consider weak scaling, that is maintaining a fixed problem size per thread as we increase the number of threads. Figure 3.14 compares the weak scaling of LAV for different amounts of per-thread work. To keep the per-thread work fixed, we generate a random graph (as described in Section 3.6) with a scaled number of rows for each thread count. Figure 3.14a shows weak scaling behavior when per-thread work is 2^{21} vertices, for graphs with average degrees of $\{16, 32, 64\}$. Figure 3.14b shows weak scaling when per-thread work is 2^{22} vertices, for graphs with average degrees of $\{16, 32\}$. In both figures, the data points

of each thread count are relative to the single-threaded execution time on the corresponding random graph.

LAV has almost linear scaling up to 16 threads in both Figure 3.14a and 3.14b. Above 16 threads, the scaling is sub-linear due to memory hierarchy bandwidth saturation. Additionally, LAV has slightly better scaling behavior on the graphs with larger average degrees (32 and 64). Changing the amount of work per thread does not change LAV's scaling behavior.

3.8 RELATED WORK

Locality optimizations. CSR segmenting in Cagra [88] preprocesses the graph, dividing it into smaller LLC-fitting sub-graphs. This approach is similar to our technique, but it does not consider the density of the segments and vectorization. Binning techniques [61, 62] are discussed in detail in Section 3.3. Milk [89] proposes language extensions to improve the locality of indirect memory accesses that are also observed in SpMV calculations. There are also compile-time and runtime techniques to accelerate programs with indirect memory accesses [90, 91]. Moreover, partitioning techniques can also be used for improving locality [92]. There are also locality optimizations targeting different primitives, such as sparse matrix dense matrix (multi-vector) multiplications [6, 93].

The effectiveness of locality optimizations and auto-tuning approaches were previously evaluated on older hardware generations [63, 94]. We provide a similar analysis on a state-of-the-art multi-core system, which shows that modern hardware renders prior techniques significantly less effective and motivates LAV, which is very effective on modern hardware.

Graph reordering. A large body of previous work targets relabeling vertices of graphs to provide better locality [68, 69, 70, 95]. These sophisticated techniques achieve high locality but incur large overheads. Rabbit Ordering [70] reduces preprocessing time using parallelization. However, it was recently shown that for many graph algorithms, Rabbit Ordering only yields limited end-to-end performance benefits [75]. Finally, FEBA [95] tries to find dense subblocks and uses graph reordering to improve locality and computational efficiency.

Graph Processing Platforms. Several platforms reformulate graph algorithms as sparse matrix operations. GraphMat [58] maps a vertex program into generalized SpMV operations and outperforms state-of-the-art graph processing frameworks. Pegasus [18] builds a large-scale system on Hadoop using generalized matrix-vector multiplication.

Vectorization. Many different SpMV vectorization methods have been proposed [59, 60, 64, 65, 66, 67]. Their main aim is to maximize the vector unit utilization. Liu et al. [66] propose to use finite window sorting, which is similar to RFS but only considers a small

block of rows. VHCC [65] devises a 2D jagged format for efficient vectorization of SpMV. Kreutzer et al. [64] use an RFS-like sorting for a limited number of rows. These works do not exploit the opportunities provided by CFS and segmenting. Furthermore, the matrices evaluated in these works are at least an order of magnitude smaller than our inputs. Finally, the significant imbalance between the number of nonzeros per row in power-law graphs limits these prior works' applicability to these graphs. Of these related approaches, we compare to CSR5 [59] and CVR [60].

Vectorization is also studied in the graph applications domain. For instance, SlimSell [96] proposes a vectorizable graph format for accelerating breadth-first search. Grazelle [97] is a graph processing framework providing a new graph representation extended for efficient vectorization.

3.9 CONCLUSIONS

In modern OOO processors, existing techniques to speed up SpMV of large power-law graphs through vectorization and locality optimizations are not effective. To address this problem, we propose LAV, a new SpMV approach that leverages the input's power-law structure to extract locality and enable effective vectorization. LAV splits the input matrix into a dense and a sparse portion. The dense portion is stored in a new matrix representation, which is vectorization-friendly and exploits data locality. The sparse portion is processed using the standard CSR algorithm. We evaluate LAV on several real-world and synthetic graphs on a modern aggressive OOO processor, and find that it is faster than CSR (and prior approaches) by an average of 1.5×. LAV reduces the number of DRAM accesses by 35% on average.

We believe that LAV's ideas are applicable beyond CPU architectures. For example, in GPUs, LAV's segmenting idea can be generalized to consider GPU memory as the last level of the memory hierarchy. Moreover, CFS and similar optimizations may be useful in GPUs to improve memory coalescing. Future work could explore these opportunities.

Another promising line of future work is to incorporate LAV into graph frameworks such as Ligra [17] and GraphMat [58]. This work would require supporting the use of frontiers for dense pull-based implementations [17] and enabling masking for the output vector as in GraphBLAS [98].

Finally, users currently need to manually decide whether to employ LAV based on their domain knowledge about the structure of the input graph (e.g., whether the graph is skewed or not). In the next section, we explore how LAV's optimizations can be generalized and SpMV method selection process can be automated.

CHAPTER 4: PREDICTING PERFORMANCE OF SPARSE MATRIX VECTOR MULTIPLICATION WITH MACHINE LEARNING

4.1 INTRODUCTION

In Chapter 3, we have discussed Locality-Aware Vectorization (LAV) for power-law graph analytics applications. However, the use of SpMV is not limited to graph analytics. And in fact, SpMV is one of the most frequently used kernels in computational science and engineering applications. In addition to graph analytics [1, 2, 3, 4], it is also an essential computational building block for solving large sparse linear systems, economic modeling, eigenvalue problems, and even information retrieval problems [8, 9, 10]. Moreover, many applications utilizing the SpMV kernel are iterative, executing SpMV many times with the same sparse input matrix. As a result, SpMV often consumes many execution cycles, making it an important target to optimize.

Executing SpMV efficiently for a wide range of input matrices from different domains is challenging. The reason is that these matrices may be highly irregular, causing the SpMV memory access patterns to have low locality. Moreover, the data-dependent behavior of some accesses makes them hard to predict and optimize for. As a result, many SpMV methods were invented for improving SpMV performance [59, 60, 65, 99, 100, 101]. Each such method represents the matrix differently to overcome the irregularity challenges mentioned above and benefit from vector (SIMD) instructions.

Unfortunately, no single method consistently yields the best performance for all sparse matrices due to their different characteristics. Therefore, we need mechanisms to identify the best SpMV method for a given matrix to achieve high performance for a wide range of sparse matrices and graphs. Traditionally, auto-tuners are used for this purpose by creating simplified analytical models [63, 102].

However, such simple models, which rely on a small set of matrix characteristics such as the number of rows, columns, and nonzeros, fail at predicting performance in the face of the rich space of matrices that come from numerous different domains such as graph analytics, engineering problems, and many others. The reason for this failure is that an SpMV method can behave completely differently on different matrices with the same number of rows, columns, and nonzeros due to the matrices' skew and locality behavior.

In this work, we address this problem. We propose the Matrix-Vector Multiplication Performance Predictor (MVPP). MVPP is a machine learning approach that can estimate the potential speedup of a given matrix for a variety of SpMV methods and choose an efficient SpMV method. It is a coordinated effort between creating an extendable machine

learning (ML) system, a set of carefully designed features to summarize locality, skew, and size characteristics of sparse matrices, and a representative training set for matrices.

First, we unify a wide range of SpMV vectorization methods such as SELLPACK [99], Sell-c-σ [100], and LAV [101] in a single optimization framework. Then we perform a detailed analysis of many sparse matrices and SpMV methods. Our study reveals the requirements for a successful ML-based performance predictor and the characteristics of sparse matrices. Finally, based on this analysis, we develop an ML prediction model for SpMV method speedup.

Our model relies on a set of novel matrix features to identify sparse matrices' size, skew, and locality characteristics. MVPP uses general statistics such as mean, standard deviation, Gini index, and p-ratio of nonzero distributions of rows and columns to summarize skew properties. To identify matrix locality characteristics, MVPP uses statistics for the distribution of nonzeros to a 2D tiled version of the matrix. Furthermore, MVPP uses additional statistics designed to summarize the structure of the tiles. We also observe that a representative matrix set is needed for an ML method to succeed. We construct such a training set by generating matrices with diverse locality and skew behavior. In the end, MVPP achieves an average speedup of 2.4× with respect to Intel's Math Kernel Library (MKL) baseline. Furthermore, MVPP achieves 1.13× speedup over MKL inspector-executor with ≈50% less preprocessing overhead.

4.2 BACKGROUND AND SPMV OPTIMIZATION SPACE

In this work, we again consider the SpMV problem of computing $y = \mathbf{A}x$, where $\mathbf{A}$ is a sparse matrix and y and x are dense output and input vectors, respectively. The details of the SpMV computation are described in Section 3.2. In this section, we discuss the baseline implementation of SpMV with the Compressed Sparse Row (CSR) format with a focus on scheduling decisions and the optimization space of SpMV using vectorization. Finally, we describe our unified vectorization format Segmented Reordered Vector Packing (SRVPack) and its SpMV implementation.

4.2.1 SpMV with CSR Format and Scheduling Decisions

CSR is the most popular sparse matrix representation. In this work, we again use CSR as our baseline. Section 3.2.1 describes the details of the CSR format.

Algorithm 4.1 shows the implementation of SpMV with the CSR format. The algorithm iterates over the matrix rows and calculates the dot product of the row and the input vector.

It can be parallelized over the rows. There are different ways to schedule rows to threads in a parallel implementation. We consider three options: dynamic (Dyn), static (St), and static contiguous (StCont). Dyn and St assign K rows at a time to threads—either dynamically or round-robin statically. StCont divides the rows by the number of threads and assigns the resulting chunks of contiguous rows statically to threads.

Algorithm 4.1 Implementation of CSR SpMV.

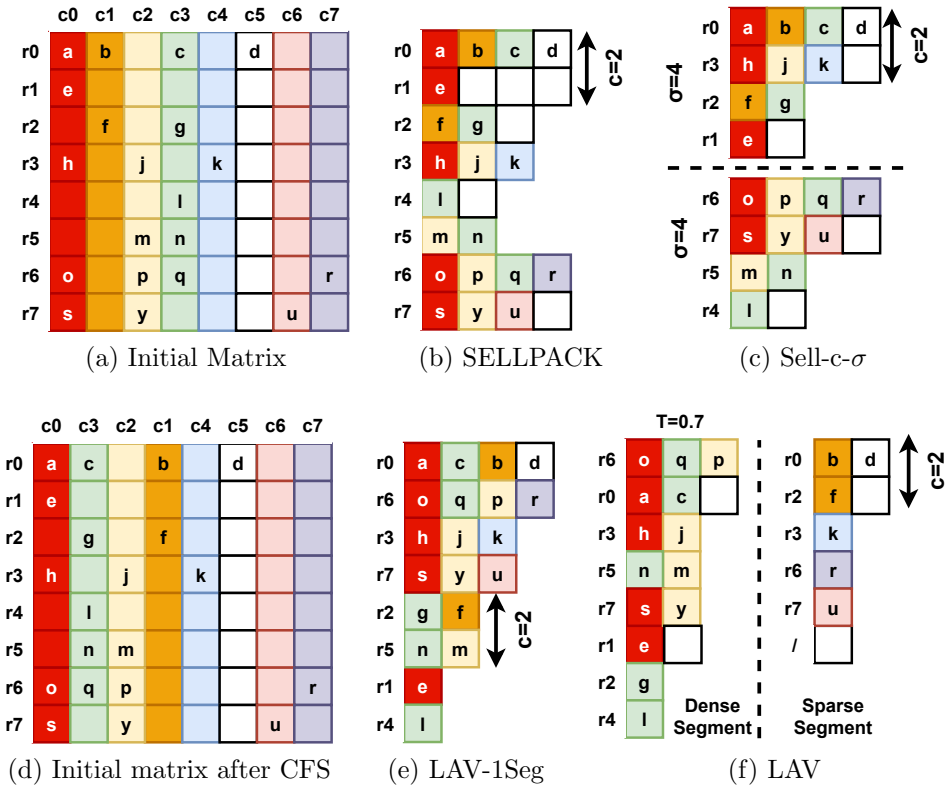
```
1: for i←0 to m-1 in parallel SCHEDULE(POLICY) do
2:     sum←0
3:     for e ← row_ptr[i] to row_ptr[i+1]-1 do
4:         sum←sum + vals[e] × x[col_id[e]]
5:     end for
6:     y[i]←sum
7: end for
```

4.2.2 SpMV Optimization Space with Vectorization

We consider vectorized SpMV implementations, where threads use vector instructions to compute over multiple rows simultaneously. Our goal is to evaluate a wide range of vectorization optimizations. We consider three types of optimizations. The first one is *zero padding minimization* [100], which reduces the unnecessary compute and memory overhead introduced by zeros. Such zeros often appear when multiple rows with different numbers of nonzeros are processed together with a vector instruction. The second optimization is *column reordering* [101], which moves columns with a high number of nonzero elements together. The goal is to place the input vector elements frequently accessed together in the same cache line, increasing locality and reducing the impact of irregular memory accesses. The third optimization is *segmenting* [101], which processes sets of columns at a time. The goal is to fit frequently-accessed portions of the input vector into the Last-Level Cache (LLC), thereby improving LLC locality and reducing DRAM accesses. Note that LAV embodies all of these optimizations. RFS achieves zero-padding minimization while we reorder the columns with CFS to improve locality. Finally, LAV employs segmenting to create LLC-sized dense segments.

To model different levels of these optimizations, we pick three effective vectorization methods, namely SELLPACK [99], Sell-c-σ [100], and LAV [101]. In addition, we also use specialized versions of Sell-c-σ and LAV. We consider all of them next. In our discussion, we use the example sparse matrix shown in Figure 4.1a.

Sliced ELLPACK (SELLPACK). SELLPACK groups c consecutive rows of the sparse matrix into *chunks*, packing nonzeros of c adjacent rows together. All rows in a chunk are



(a) Initial Matrix (b) SELLPACK (c) Sell-c-σ

(d) Initial matrix after CFS (e) LAV-1Seg (f) LAV

Figure 4.1: Different formats of an example sparse matrix.

processed together with the same vector instructions. As a result, if the rows in a chunk have different numbers of nonzeros, they are padded to the same length. Figure 4.1b shows the SELLPACK format of the example matrix with c=2. SELLPACK can introduce many zeros due to padding, especially when the input matrix has an unbalanced distribution of nonzeros across rows. The chunk size is typically given by the width of the machine's vector unit and determines the degree of padding. For example, Intel Skylake processors support 4- and 8-wide vector operations. For SELLPACK, we use StCont and Dyn scheduling because we see they are generally the fastest.

Sell-c-σ. Sell-c-σ extends SELLPACK to reduce zero padding. Sell-c-σ first considers groups of σ consecutive rows of the matrix. Then, it reorders the rows within each group based on their number of nonzeros in descending order. With this operation, each of the resulting groups of c consecutive rows is likely to have rows with a similar number of nonzeros. Hence, as vectorization is applied, the amount of padding will be smaller. Figure 4.1c shows the Sell-c-σ format of the example matrix with σ=4 and c=2.

σ needs to be tuned for each matrix. The best value of σ depends on the distribution of nonzeros across the rows of the matrix. If most rows have similar numbers of zeros, σ can

be small, and the padding will be tolerable. However, if the number of nonzeros is highly imbalanced, keeping padding tolerable requires large σ values. In this case, many rows get reordered, which typically causes the loss of spatial and temporal locality for the input vector because close-by rows tend to have similar nonzero patterns. For Sell-c-σ, we use StCont and Dyn scheduling because we see they are generally the fastest.

Sell-c-R. Sell-c-R sets σ to the number of rows in the matrix. This method is beneficial for matrices with a very imbalanced distribution of the nonzeros to rows. For these matrices, Sell-c-R reduces padding at the expense of poor cache locality for the input vector. We refer to such reordering of all rows as *Row Frequency Sorting* (RFS) [101]. Sell-c-R uses Dyn scheduling because the large difference of nonzeros across rows creates thread load imbalance with any static scheduling.

LAV-1Seg. In matrices for power-law graphs [73, 103], the number of zeros in both rows and columns is highly imbalanced. As a result, the input vector has poor temporal and spatial locality and suffers frequent LLC misses. To address this problem, LAV-1Seg (for LAV *with one segment*) orders the columns based on decreasing numbers of nonzeros—a technique referred to as *column frequency sorting* (CFS) [101]. Then, it applies the transformations of Sell-c-R. With CFS, columns with similar nonzero distributions are often processed in close temporal succession, reusing input vector elements.

To see the LAV-1Seg format, consider Figure 4.1d, which shows the initial matrix after performing CFS. Then, LAV-1Seg applies the Sell-c-R transformation. The result is Figure 4.1e.

LAV. If the matrix is large, input vector elements are evicted from the LLC before reusing them. Consequently, the complete LAV algorithm [101] takes the matrix after CFS and partitions it into groups of consecutive columns called *Segments*. Then, it applies the transformations of Sell-c-R to each segment. A segment should be small enough so that the input vector elements corresponding to the segment fit in the LLC and get reused. In practice, LAV finds that it typically only needs to partition the matrix into two segments. The first segment includes a fraction T of the nonzeros, where the best T is generally ≥ 0.7. It is called the dense segment, as it takes the most populated columns of the matrix. The second segment is called the sparse one.

When a segment is processed, the corresponding input vector elements are loaded into the LLC and reused. Since both LAV-1Seg and LAV use RFS, we only consider the Dyn scheduling scheme, similarly to Sell-c-R. Figure 4.1f shows the resulting matrix format with LAV. Since we pick T=0.7, the dense segment includes columns 0, 3, and 2. After performing RFS, the order of rows in each segment is as shown in the figure.

Table 4.1 summarizes the details of SpMV methods, their parameters, and their optimizations.

Table 4.1: Summary of SpMV methods.

Method	Params	Optimizations	Description
CSR	SCH	None	Performance of Dyn, St, and StCont depends on skew characteristics of the matrix.
SELLPACK	c, SCH	Vectorization	c affects amount of padding, and depends on the width of the vector hardware and on the matrix structure
Sell-c-σ	c, σ, SCH	RFS (σ rows), Vectorization	σ controls the sorting of rows, and should be chosen to decrease padding while not hurting locality.
Sell-c-R	c	RFS, Vectorization	Specialized version of Sell-c-σ with $\sigma = R$
LAV-1Seg	c	CFS, RFS, Vectorization	Specialized version of LAV with a single segment.
LAV	c, T	CFS, RFS, Segmenting, Vectorization	The higher the matrix skew is, the higher the chosen T should be.

4.2.3 Unifying the Format for Vectorization

We do not want to keep a different matrix representation for each of the different optimizations we may apply. Hence, we devise a single matrix representation from which the compiler can quickly extract the information needed for vectorization (for SELLPACK, Sell-c-σ, Sell-c-R, LAV-1Seg, and LAV). We call this representation Segmented Reordered Vector Packing (SRVPack).

To support all these optimizations, SRVPack partitions the matrix data into segments if needed. For SELLPACK, Sell-c-σ, Sell-c-R, and LAV-1Seg, we create a single segment. On the other hand, LAV creates two segments, a dense and a sparse segment, as described in Section 3.5. Note that the segments are created based on columns. Therefore, each segment contains the nonzeros of the columns in the segment. Then, we apply RFS in each segment, which changes the order of rows.

Figure 4.2 shows the representation of the matrix in Figure 4.1a in the new format. Each segment has a `row_order` array to keep track of the new row order. All the segment's c-row chunks are placed in sequence, one after the other, following the order given by the segment's row order array. In each chunk, if a row has fewer columns than the rest of the rows in the chunk, it is padded with zeros (or the annihilator of SpMV for generalized form). Similar to CSR, there is a `vals` array that stores the values of the elements and a `col_ids` array that stores the corresponding column indices. These 2D arrays have a Y-dimension equal to c and an X-dimension equal to the sum of the lengths of the chunks of the segment. Finally,

an offset array stores the index of the first nonzero of each chunk. Unlike LAV, we don't use a `mask` array to eliminate the computation for zero entries.

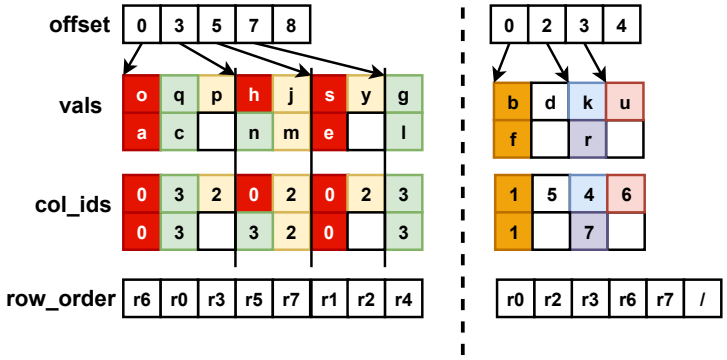


Figure 4.2: SRVPack representation of the matrix in Figure 4.1a to support all formats.

4.2.4 Using the SRVPack Format

Algorithm 4.2 computes SpMV given an SRVPack input. It operates similarly to LAV (Algorithm 3.2). The main difference is that instead of using compiler intrinsics for vectorization, it relies on the compiler for vectorization. Furthermore, it doesn't utilize `mask`s and doesn't have a CSR portion.

Algorithm 4.2 processes one segment at a time (Line 8), processing all the nonempty chunks in the segment in parallel with multiple threads (Line 10). While processing a chunk, SRVPack utilizes wide SIMD units. The rows in the chunk are processed in parallel by SIMD lanes, and the partial sums generated by the rows are accumulated into a vector (*row_sums* in Lines 18–28). Instead of using masked vector operations, we rely on zero multiplications to have a common implementation across platforms. After completing the chunk, the output values generated are accumulated into the output vector y. This operation involves reading the accumulated values up until this segment in output vector y (Line 16) and adding to them the contributions of this chunk (Lines 31–32).

4.3 CHALLENGES OF PREDICTING SPMV PERFORMANCE

Our goal is to develop a practical approach to pick the method (i.e., the algorithm) that delivers the fastest SpMV execution. To this end, in this section, we characterize the execution of the different methods of Section 4.2 on a wide range of matrices. For our experiments, we use 136 large matrices from the SuiteSparse matrix collection [82] and 408 large matrices created with the RMAT random graph generator [104]. SuiteSparse contains

Algorithm 4.2 SpMV algorithm with SRVPack.

```
1: procedure SpMV(segments, x, y, m, n, nLanes)
2:     // segments is the list of segments created for vectorization
3:     // x is the input vector, y is the output vector
4:     // m number of rows, n number of cols, nLanes number of SIMD lanes
5:     for r=0 to m-1 do  // initialize y vector to zeros
6:         y[r] ← 0
7:     end for
8:     foreach segment s in segments do  // all segs. in dense portion
9:         // chunks of a segment processed in parallel
10:        for c=0 to s.num_chunks-1 in parallel  do
11:            // rows_sums[v], row_ids[v], prev_sums[v] are vectors with width v
12:            row_sums[v]=(0, . . . , 0)
13:            #pragma simd
14:            for j=0 to v-1 do
15:                row_ids[j]=s.row_order[c*v+j]) // Get the row ids processed by this chunk
16:                prev_sums[j]=y[row_ids[j]]  // Get the prev values from y for rows of this chunk
17:            end for
18:            for i← s.offsets[c] to s.offsets[c+1]-1  do
19:                // cols, vals, x_vec, mul_vec, rows are vectors with width v
20:                #pragma simd
21:                for j=0 to v-1 do
22:                    cols[j]=s.col_ids[i*v+j]  // Get column ids
23:                    vals[j]=s.vals[i*v+j]] // Get vals
24:                    x_vec[j]=x[cols[j]] // Get vals from x
25:                    mul_vec[j]=vals[j] * x_vec[j]
26:                    row_sums[j]=row_sums[j] + mul_vec[j]
27:                end for
28:            end for
29:            #pragma simd
30:            for j=0 to v-1 do
31:                row_sums[j]=row_sums[j] + prev_sums[j] // Aggregate over segs.
32:                y[row_ids[j]]=row_sums[j]
33:            end for
34:        end for
35:    end foreach
36: end procedure
```

mostly scientific matrices, although it also has some social and web network graphs. RMAT is widely used (e.g. [85, 86]) and can generate a much wider range of matrix types. In particular, we use it to create more matrices like those of social and web network graphs. RMAT is also used in the Graph500 benchmark [86] and GAPBS [85] benchmark suite. We discuss the details of RMAT and its parameters in Section 4.4.5.

To ensure that the matrices are large enough to get representative results and consistent performance, we use matrices with 1–67 million rows and columns. However, we limit the maximum number of nonzeros to 2 billion to ensure the matrices fit in the memory of a single shared-memory machine.

Our analysis provides several insights:

1. The highest-speedup method varies across matrices. Figure 4.3 shows the distribution of the best performing method for SpMV for the 136 SuiteSparse matrices. We see that different methods are faster for different matrices. For example, only 34 of these matrices achieve the best performance with CSR, which is the default format for many BLAS

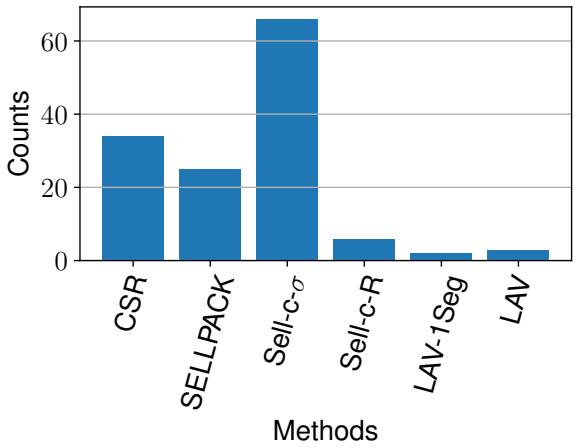


Figure 4.3: Distribution of the fastest method for SpMV in SuiteSparse matrices.

and graph frameworks. On the other hand, Sell-c-σ is the fastest method for 66 matrices. We also perform these experiments with MKL and observe that MKL doesn't yield the best performance for any of the matrices tested.

2. Within a method, the magnitude of the speedup varies. Even when a method is the fastest for a set of matrices, its actual speedup may vary considerably. To see it, Figure 4.4 shows the speedups of each method for the 136 SuiteSparse matrices normalized to the highest CSR performance for that matrix. In the figure, the horizontal line at 1.0 corresponds to the CSR implementation with the best scheduling policy. We also report the results for MKL [78], which also uses the CSR representation. The matrices are grouped in the X-axis by their best method and the speedup. Furthermore, the plot shows the names of a few of the matrices.

Consider SELLPACK, which is the fastest in 25 consecutively placed matrices. In these matrices, its speedup ranges from 1.05 to 1.31×. On the other hand, Sell-c-σ is the fastest for 66 matrices and, for these, the speedup ranges from 1.00 to 1.76×. Predicting the magnitude of the speedup is important since it tells how many SpMV iterations are needed to amortize the cost of constructing the chosen matrix format.

3. Selecting correct parameters for a method affects the speedups substantially. For example, even simple scheduling choices affect the speedups substantially. Figure 4.5 shows the performance of CSR implementations that use *Dyn*, *St*, and *StCont* scheduling. For each of the 136 SuiteSparse matrices, the figure shows the speedup of the different implementations normalized to the highest CSR performance. We see that a simple parameter like the scheduling choice significantly impacts performance, sometimes causing 10× slowdowns. *Dyn*, *St*, and *StCont* give the best performance for 28, 16, and 92 matrices, respectively.

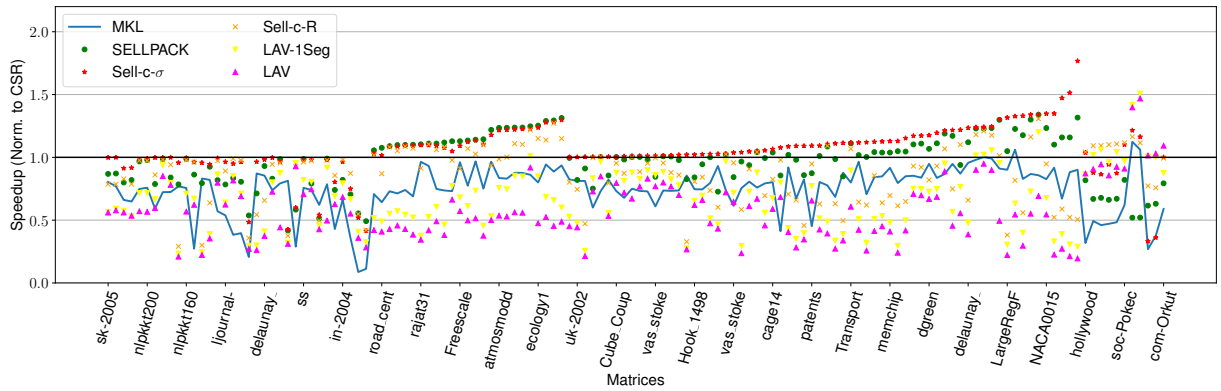


Figure 4.4: Speedup of vectorized SpMV versions and MKL over the best CSR for the SparseSuite matrices. The matrices are grouped by the fastest SpMV method.

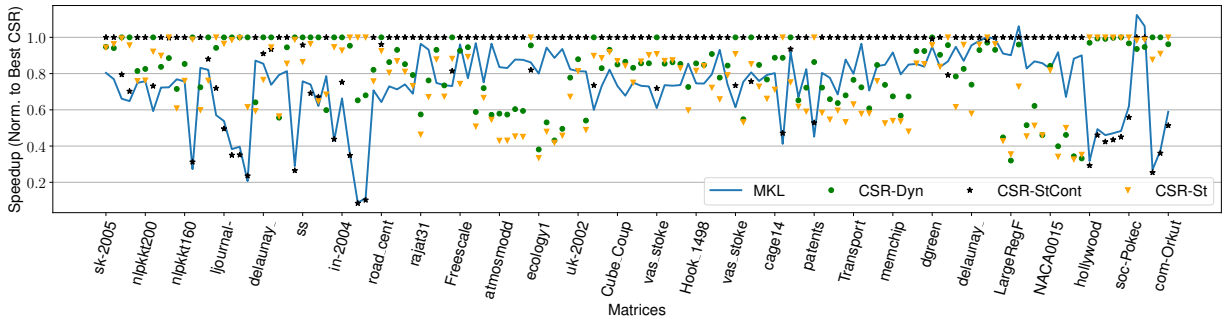


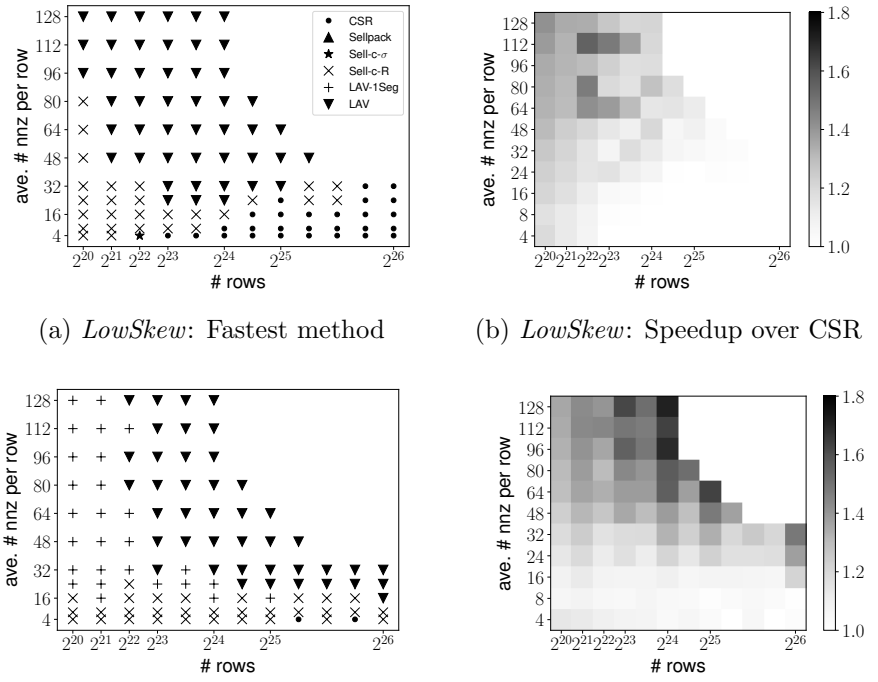
Figure 4.5: Speedup of CSR with different scheduling algorithms and MKL over the best CSR for the SparseSuite matrices.

Dyn is the fastest for web graphs and social networks, while *St* and *StCont* perform best with scientific matrices.

4. An intricate relationship between matrix size, locality, and skew determines the best method. To glimpse how the interaction between these three parameters determines the highest-speedup method, we consider two experiments with RMAT-generated matrices. They show the impact of nonzero skew and nonzero locality, respectively.

In the first experiment, we analyze the skew characteristics of sparse matrices. We consider skew in terms of the distribution of nonzeros to rows of the sparse matrix. We use high (*HighSkew* set) and low skew (*LowSkew* set) matrices. In the second experiment, we use matrices with a higher nonzero locality (i.e., most nonzeros are placed in areas close to the matrix diagonal) (*HighLoc* set) and with low locality (i.e., nonzeros are spread all over the matrix) (*LowLoc* set). Section 4.4.5 describes the RMAT parameters used. In each 102-matrix set, we vary the number of rows and the average number of nonzeros per row to model different size characteristics.

We consider the effect of skew first. We measure the execution time of each SpMV method of Section 4.2 for the *LowSkew* and *HighSkew* sets. Then, for *LowSkew*, Figures 4.6a and 4.6b show the fastest method and its speedup over the best CSR, respectively. The figures show data points for most of the matrices (for readability), where a matrix is characterized by the number of rows (X-axis) and the average number of nonzeros per row (Y-axis). Figures 4.6c and 4.6d show the same data for *HighSkew* matrices.



(a) *LowSkew*: Fastest method

(b) *LowSkew*: Speedup over CSR

(c) *HighSkew*: Fastest method

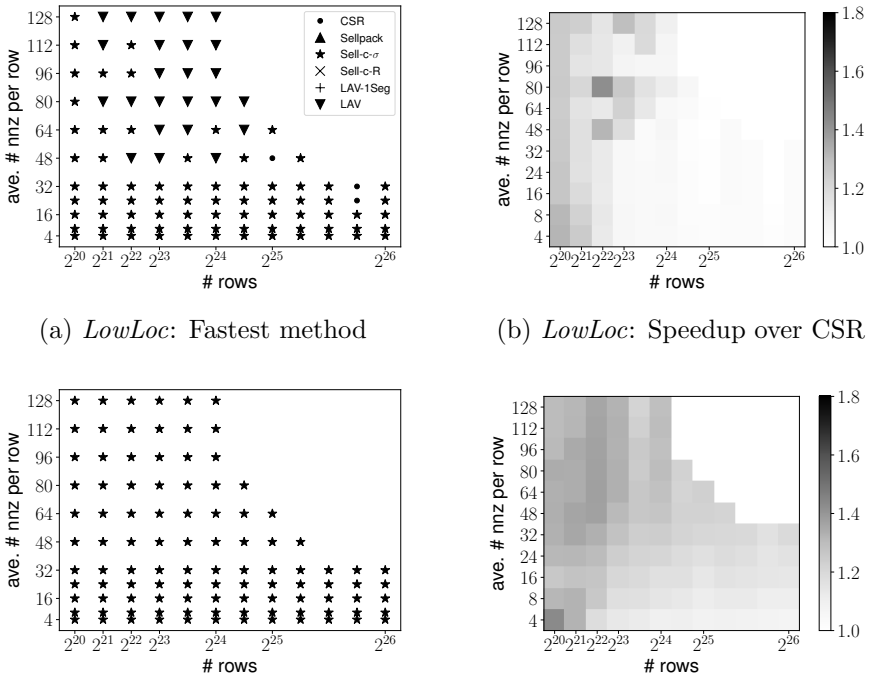
(d) *HighSkew*: Speedup over CSR

Figure 4.6: Behavior of matrices with different skew.

Looking at Figures 4.6a and 4.6c, we see that LAV, LAV-1Seg, and Sell-c-R deliver the highest speedups most of the time. LAV typically outperforms other methods when the input matrix is larger than the LLC size (i.e., number of rows $\geq 2^{22}$) and the average number of nonzeros per row is high (≥ 16). LAV's speedup depends on the skew of the matrix; it is highest for the *HighSkew* matrices (Figure 4.6d). For matrices with few rows and a high average number of nonzeros per row, LAV-1Seg often delivers the highest performance. This is because a single segment is enough. When matrices have a low average number of nonzeros per row, are small, and especially are in *LowSkew*, Sell-c-R is often the fastest method. This is because the input vector fits in the last-level cache, and there is no need for the more advanced LAV and LAV-1Seg formats. However, the speedups are limited.

Figure 4.7 shows the same data for the *LowLoc* set (Figures 4.7a and 4.7b) and the *HighLoc* set (Figures 4.7c and 4.7d). We observe that Sell-c-σ is the fastest method for matrices with

high locality (*HighLoc*). For *LowLoc*, Sell-c-σ is often the best except for large matrices with a high average number of nonzeros per row. In this case, LAV is the best because it provides segmenting. Although the skew is limited, LAV creates a segment that can fit in the LLC. For *HighLoc*, this is unnecessary because caches work efficiently without segmenting. Finally, the magnitude of the Sell-c-σ speedups is higher for *HighLoc* than for *LowLoc*.



(a) *LowLoc*: Fastest method

(b) *LowLoc*: Speedup over CSR

(c) *HighLoc*: Fastest method

(d) *HighLoc*: Speedup over CSR

Figure 4.7: Behavior of matrices with different locality.

5. SuiteSparse doesn't have many matrices with diverse behaviors. As seen in Figure 4.3, the distribution of the fastest method in SparseSuite matrices strongly favors Sell-c-σ. However, our experiments with RMAT-generated matrices showed that LAV and LAV-1Seg are faster for HighSkew matrices.

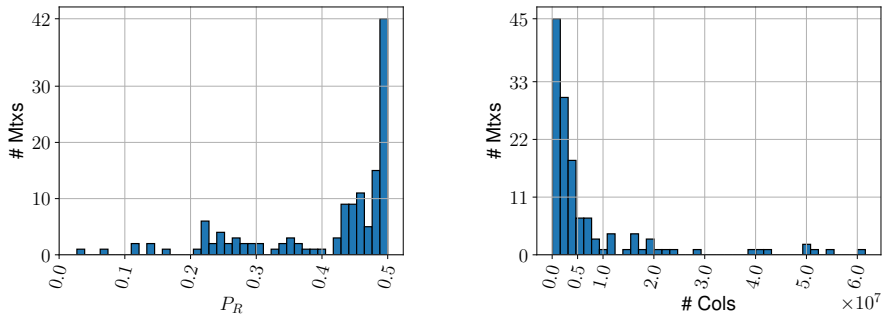
This discrepancy occurs because SuiteSparse matrices come from the scientific domain or problems with a geometric structure such as road graphs. Therefore, they are mainly non-skewed matrices.

We can see the bias in skew by plotting a histogram of the *p-ratio* of nonzeros in SuiteSparse matrices. A p-ratio of p indicates that a p fraction of rows has a $(1-p)$ fraction of the nonzeros in the matrix. Figure 4.8a shows the distribution of the nonzero p-ratio. We see that most of the SuiteSparse matrices have a p-ratio of 0.4 or more. This means that many matrices have a balanced distribution of nonzeros to rows. This fact makes SELLPACK and Sell-c-σ

methods effective.

The number of columns also significantly affects the cache behavior of SpMV computations. Figure 4.8b shows the distribution of the number of columns in our SuiteSparse set—which includes almost all of the largest matrices in SuiteSparse. We see that SuiteSparse matrices generally have a low number of columns. Most of them have $< 5\,\mathrm{M}$ columns, allowing the SpMV input vector to fit into the LLC and reducing the need for LAV.

Overall, SuiteSparse favors methods such as SELLPACK and Sell-c-σ. If we want a representative set of matrices to train our prediction model, we need to augment SuiteSparse with a broader range of matrices. For this reason, we augment our set of matrices with synthetic matrices generated by RMAT and RGG generators.



(a) P-ratio of nonzeros per row (b) Number of columns

Figure 4.8: Distribution of non-zero p-ratio and number of columns for SuiteSparse matrices.

4.4 MVPP: PICKING THE BEST SPMV METHOD

Given how challenging it is to pick the best SpMV method for a given input matrix, we propose to leverage machine learning (ML). This section presents MVPP, an ML-based framework that can select a high-performance SpMV method for a given sparse matrix. We envision that the MVPP can be integrated to the backend of GraphBLAS/BLAS frameworks such as Intel's MKL [78]. MVPP is designed to be transparent to the programmer, and integrate easily into the existing systems.

4.4.1 Overall MVPP Design

MVPP consists of a novel sparse matrix feature set, a set of performance prediction models, and a heuristic to choose the best SpMV method considering the outputs of performance prediction models.

Figure 4.9 shows the operation of MVPP. First, MVPP extracts the value of certain features from the input matrix ((1)). The feature set includes features to identify size, nonzero distribution skew, and locality characteristics of sparse matrices. In the second step, these features are passed to a set of ML performance models that predict the speedup of the methods ((2)). MVPP has a performance prediction model for each combination of method and parameter values. Next, MVPP picks the method and parameter value predicted to deliver the highest speedup with the lowest preprocessing cost ((3)). Then, it transforms the matrix layout from CSR to the layout for the selected method and parameter pair ((4)). If MVPP chooses CSR, we do not perform any transformations. Finally, we run SpMV with chosen matrix layout ((5)).

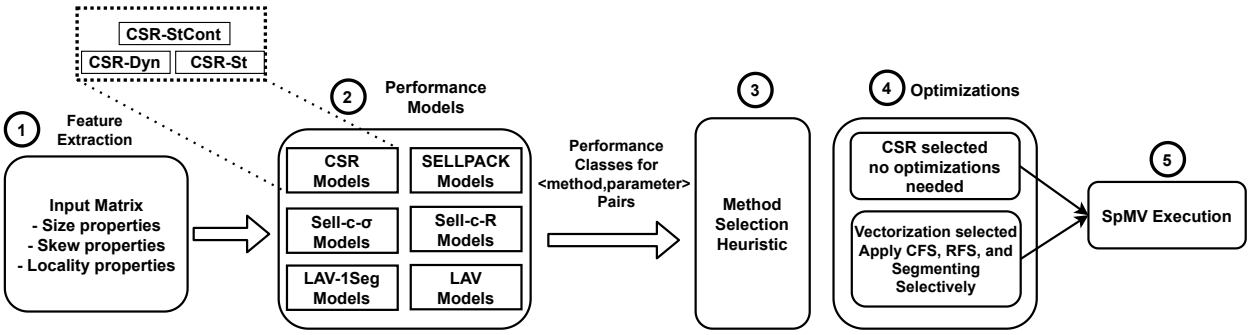


Figure 4.9: MVPP operation.

4.4.2 Sparse Matrix Features

In Section 4.3, we observed that a sparse matrix's size, nonzero skew, and nonzero locality characteristics substantially impact the performance of an SpMV method. We choose features to build our ML-based performance models based on these insights and previous analysis [103].

We summarize these characteristics by analyzing the distribution of nonzeros to rows (R), columns (C), tiles (T), row blocks (RB), and column blocks (CB) of the sparse matrix. R and C distributions give us the number of nonzeros per row and column, respectively. The T distribution is created by logically breaking the matrix into K^2 tiles (Figure 4.10). K is selected to be 2048 by considering the size of sparse matrices and L2 cache size. Moreover, each tile has $M = n_R/K$ rows and $N = n_C/K$ columns, where n_R is the number of rows and n_C is the number of columns of the sparse matrix. A row of tiles is called a *row block*, and a column of tiles is called a *column block*. The distribution of nonzeros to these row blocks and column blocks creates RB and CB distributions, respectively.

To characterize these distributions, we use the mean (μ), standard deviation (σ), variance (σ^2), *min*, *max*, number of nonzero entries (*ne*), *Gini coefficient (G)*, and *p-ratio (P)*. The Gini coefficient and p-ratio are statistical measures that calculate the imbalance (or inequality) in distribution [103]. For example, a perfectly balanced distribution has a G=0 and P=0.5. In a maximum imbalanced distribution (i.e., all nonzeros are in one row, column, or tile), G gets close to 1, and P gets close to 0. We use the following naming convention for these summary statistics: $\text{stat}_{\text{dist}}$, where stat is the name of the summary statistic, and dist is the distribution name.

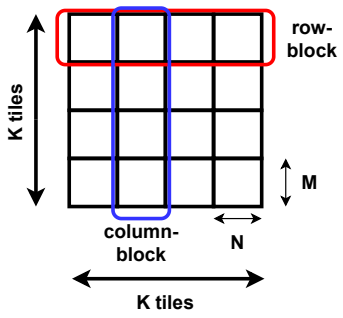


Figure 4.10: Logically tiling a matrix.

Next, we describe how we use these summary statistics described over R, C, T, RB, and CB distributions and additional statistics to create the sparse matrix features for our ML-based performance predictors. Table 4.2 shows the features MVPP extracts from the input matrix, grouped into matrix size, nonzero skew, and nonzero locality properties.

Table 4.2: Matrix features used in our ML-based performance models.

	Distribution	**Metrics**	**Explanation**
Size		n_R, n_C, n_{nnz}	Size characteristics of the matrix
Skew	*Rows(R)*	μ_R, σ_R, σ_R^2, G_R, P_R, min_R, max_R, ne_R	Models load balancing, scheduling, and vectorization
	Cols(C)	μ_C, σ_C, σ_C^2, G_C, P_C, min_C, max_C, ne_C	Models access patterns to the input vector
Locality	*Tiles(T)*	μ_T, σ_T, σ_T^2, G_T, P_T, min_T, max_T, ne_T	Models locality behavior across tiles
	RowBlocks(RB)	μ_{RB}, σ_{RB}, σ_{RB}^2, G_{RB}, P_{RB}, min_{RB}, max_{RB}, ne_{RB}	Models locality in L1 and L2 caches
	ColBlocks(CB)	μ_{CB}, σ_{CB}, σ_{CB}^2, G_{CB}, P_{CB}, min_{CB}, max_{CB}, ne_{CB}	
		uniqR, *uniqC*, *GrX_uniqR*, *GrX_uniqC*	
		potReuseR, *potReuseC*, *GrX_potReuseR*, *GrX_potReuseC*	Models locality in the last level cache (LLC)

(1) Matrix Size Properties. MVPP measures the number of rows (n_R), columns (n_C), and nonzeros (n_{nnz}). n_R and n_C give information about the size of the output and input vector, respectively, while n_{nnz} gives information about the amount of work to be done.

(2) Nonzero Skew properties. MVPP uses features that measure the skewness of the nonzero distribution across rows (R) and columns (C). We collect the mean (μ_R and μ_C), standard deviation (σ_R and σ_C), variance (σ_R^2, and σ_C^2), min (min_R and min_C), max (max_R and max_C), Gini coefficient (G_R and G_C), p-ratio (P_R and P_C), and the number of nonempty

rows (ne_R) and columns (ne_C).

The features of the R distribution determine the row scheduling and the padding for vectorization. The features of the C distribution determine the irregularity of the memory accesses. Therefore, they can indicate the effectiveness of CFS in LAV-1Seg and LAV.

(3) Nonzero Locality Properties. We use T, RB, and CB distributions' statistics to describe the locality characteristics of a given sparse matrix. With these features, MVPP estimates the nonzero locality. For example, a low p-ratio in the T distribution tells that the nonzeros are grouped together in a few tiles. Therefore, the matrix has a relatively high nonzero locality.

MVPP also collects additional information on the layout of nonzeros inside each tile. Specifically, for tile i, $uniqR_i$ and $uniqC_i$ are the number of unique rows and of unique columns that contain nonzeros, respectively. If many nonzeros are in the same row or column, the program is likely to exploit locality. Furthermore, since data is laid out in cache lines, if the nonzeros are nearby, they may share caches lines and further enhance locality. Hence, MVPP also measures GrX_uniqR_i and GrX_uniqC_i for tile i, which are unique rows and unique columns with nonzeros *grouped in groups of X adjacent elements*. For example, if X=16, 16 adjacent columns (or rows) will count as a single ID. The X values used by MVPP are $\{4, 8, 16, 32, 64\}$. Consider 64-byte cache lines as an example. In this case, a line may fit four 128-bit elements or eight 64-bit elements. Therefore, we can get a cache hit if more than one element of the cache line is accessed in close temporal succession.

MVPP computes $uniqR_i$, $uniqC_i$, GrX_uniqR_i, and GrX_uniqC_i for each tile i. Then, it sums the values across all the tiles and divides the result by the number of nonzeros in the matrix. The resulting values, which we call $uniqR$, $uniqC$, GrX_uniqR, and GrX_uniqC, are used as matrix feature values.

Finally, MVPP measures, for each row i, the number of tiles where the row has at least one nonzero ($potReuseR_i$). It also measures a similar metric for each column i ($potReuseC_i$). If these metrics are high, there is potential for data reuse in the last-level cache, as data is reused across tiles. In addition, MVPP measures, for each group of X consecutive rows i, the number of tiles where the group has at least one nonzero ($GrX\_potReuseR_i$). For each column i, it also measures a similar metric, called $GrX\_potReuseC_i$. As before, these metrics try to include the effect of cache lines. Finally, MVPP takes the average across all rows ($potReuseR$), all columns ($potReuseC$), all groups of X consecutive rows ($GrX\_potReuseR$), and all groups of X consecutive columns ($GrX\_potReuseC$). These averages are used as matrix feature values.

Note that the distributions are not the features used to train our ML-based performance

prediction models. Instead, we only use the summary statistics calculated from the R, C, T, RB, and CB distributions.

4.4.3 Performance Prediction Models

Each feature comes from a significantly different number domains. For example, the number of rows and columns can be in millions while the Gini index is a number between $[0-1]$. Thus, it is hard to choose good normalization techniques for each feature. Therefore, we use *decision trees* for the performance models of each SpMV <method, parameter>. Decision trees in this context can be seen as creating simple decision rules inferred from the data features.

MVPP's models are as follows. CSR has three models corresponding to the Dyn, St, StCont scheduling methods. For Sell-c-σ, we have as many models as possible combinations of the c values, the σ values, and the number of scheduling mechanisms (StCont and Dyn) considered. Sell-c-R and LAV-1Seg have only c parameter to tune since they only use Dyn scheduling. Finally, LAV has as many models as the combinations of c values times T values. We pick $c = \{4, 8\}$ because these are the widths of the vector instructions in the machine evaluated. We pick σ=$\{2^9, 2^{12}, 2^{14}\}$ to cover a range of behaviors: 2^9 and 2^{14} roughly correspond to the maximum values for which the input and output vectors fit amply in the L1 cache and the L2 cache, respectively; 2^{12} is a value in between. Finally, we pick T=$\{0.7, 0.8, 0.9\}$ to cover different levels of nonzero skew. At the end, we have a total of 29 decision tree models for all SpMV methods and parameters considered.

Furthermore, it is possible to run into overfitting issues with decision trees. For this reason, we apply pruning techniques. First, we limit the maximum depth of the trees to 15 to avoid creating branches that have only a few samples in them. Second, we enable minimal cost-complexity pruning, with a threshold of 0.005. The maximum depth of the tree and threshold for pruning are selected experimentally using grid search. Furthermore, our decision trees use the Gini measure for split criteria.

Speedup Classes. Each of the models predicts the execution time of the method and parameter value combination *relative* to the fastest CSR method. Specifically, a model can output one of seven classes (C0-C6), corresponding to ranges of relative execution time. These ranges are, from higher (slower) to lower (faster): C0 = [1.05 - ∞), C1 = [0.95 - 1.05), C2 = [0.85 - 0.95), C3 = [0.75 - 0.85), C4 = [0.65 - 0.75), C5 = [0.55 - 0.65), and C6 = [0 - 0.55). To create these discrete classes C1-C5, we created categories with a step size of 0.1 between speedup values $\approx$1$\times$ and $\approx$2$\times$. C0 and C6 represent all other matrices that don't fall in this range. C0 includes all matrices that have a slowdown with the given method,

and C6 represents the matrices with more than 2× speedup. We observed that there aren't many matrices that can obtain more than 2× speedup. Note that the classes are created based on normalized execution time. Thus, the range [0-1] represents speedups. Therefore, our classes focus on performance improvements rather than performance degradation.

4.4.4 Choosing the SpMV Method

After separate ML models predict the speedup class, we use a cheapest-first strategy based on preprocessing cost of each method. Given a list of speedup classes for each method and parameter pair, it chooses the method and parameter pair with the highest speedup class with the smallest preprocessing cost. We use the following order: CSR, SELLPACK, Sell-c-σ, Sell-c-R, LAV-1Seg, and LAV. If there is a tie among different methods, we choose the first one. Furthermore, we also need a mechanism to break the ties among different parameters of a method. For this reason, we also have an order for the parameters of each method. Specifically, we sort the parameters smallest to largest. For example, for LAV, the order is $T = 70\%$, $T = 80\%$, and $T = 90\%$ and all $c = 4$ (SIMD length 4) LAV methods preceed $c = 8$ (SIMD length 8) LAV methods.

4.4.5 Creating a Representative Training Set

An ML model requires patterns to perform predictions. However, our analysis in Section 4.3 shows that real-world graphs available are skewed in terms of their characteristics.

We use the RMAT [104] random graph generator with carefully selected parameters to model these aspects not well represented in SuiteSparse and obtain a representative training set. The RMAT generator has 3 parameters: (1) number of nodes, (2) average degree (i.e. number of edges), and (3) probabilities a, b, c, d for an edge to fall into each of the four quadrants ($a + b + c + d = 1$). The graph is generated recursively. A quadrant is picked according to the probabilities to place each edge. Selected quadrant is again subdivided into four partitions, and the process repeats until ending at a single matrix cell.

For obtaining skewed matrices, we start with Graph500 [86] parameters $a = .57$, $b = .19$, $c = .19$, and $d = .5$ (*HighSkew*) which generates a power-law graph with a strong community structure. Then we decrease the a parameter while increasing b, c, and d to have less skew while still showing a community structure ($a > d$). We analyze two more graphs with parameters $a = .46$, $b = .22$, $c = .22$ (*MedSkew*), and $a = .35$, $b = .25$, $c = .25$ (*LowSkew*). The p-ratio of nonzero distribution of rows is ≈0.1, ≈0.2, ≈0.3 for *HighSkew*, *MedSkew*, and *LowSkew* graphs, respectively.

For obtaining matrices with varying locality, we start with $a = b = c = d = .25$ (Erdos-Renyi), which has a more uniform distribution of nonzeros to rows, but nonzeros are uniformly distributed across all columns and rows. We obtain matrices with nonzeros gathered around the diagonal by increasing the a and d parameters equally and decreasing b and c with the same amount. With *MedLoc* ($a = d = 0.35$, $b = c = 0.15$) and *HighLoc* ($a = d = 0.45$, $b = c = 0.05$), locality increases. For all *LowLoc*, *MedLoc*, and *HighLoc* graphs, the p-ratio of rows' nonzero distribution is 0.4-0.5, showing little skew. In addition to RMAT graphs, we also include random geometric graphs (RGG) [105]. RGGs are undirected spatial graphs. They generate a random graph by placing n vertices uniformly at random in a 2-dimensional unit grid. An edge connects the vertices if their distance in the 2D grid is below a given radius r. r is determined by the average degree expected from the random graph. We include *RGG* graphs to model the behavior of matrices with spatial structure, and they fall under high locality matrices.

Table 4.3: Parameters for Random Matrices.

Behavior	**Matrix**	**Parameters**
Skew	*HighSkew* (HS)	$a = .57$, $b = .19$, $c = .19$, and $d = .5$
	MedSkew (MS)	$a = .46$, $b = .22$, $c = .22$
	LowSkew (LS)	$a = .35$, $b = .25$, $c = .25$, $d = .15$
Locality	*LowLoc* (LL)	$a = b = c = d = 0.25$
	MedLoc (ML)	$a = d = 0.35$, $b = c = 0.15$
	HighLoc (HL)	$a = d = 0.45$, $b = c = 0.05$
	RGG (rgg)	Radius (r) is set based on ave. nnzs per row

4.5 EXPERIMENTAL SETUP

Test Environment. We use an Intel Skylake shared-memory machine. It has 4- and 8-wide vector instruction support. The machine has two processors; each has 12 2.6 GHz cores, a total of 24 cores. Each core has a private 32 KB L1 data cache and a private 1 MB L2 cache. Each processor has a shared 19 MB LLC. The machine has 192 GB of main memory. Table 4.4 summarizes the details of our system.

Table 4.4: System characteristics.

Component	**Characteristics**
CPU	Intel Gold 6126 CPU @ 2.60GHz 12 cores per processor, 2 processors
Cache	Private 32 KB instruction and data L1 caches Private 1 MB L2 cache, 19 MB LLC per processor
Memory	192 GB
Vector ISA	avx512f, avx512dq, avx512cd, avx512bw, avx512vl

Matrices. To train and test MVPP, we use 136 matrices from SuiteSparse [82] and 1326 random graphs generated as described in Section 4.4.5. For SuiteSparse matrices, we limit ourselves to matrices with 2^{20}–2^{26} rows to make sure the performance of SpMV is consistent. We also use matrices with less than 2 billion nonzeros in order to be able to fit the matrices in a single machine's memory. For random matrices, we use RMAT and random geometric graph (RGG) generators. For both, we test matrices with 2^{20} to 2^{26} rows with the average number of nonzeros per row $4 - 128$. However, the matrix size increases significantly for the large matrices ($2^{24} - 2^{26}$ rows). Therefore, we don't generate matrices with the high average number of nonzeros per row for these large matrices to limit the total number of nonzeros to be less than 2 billion. We also include matrices with $2^{24.58}$, $2^{25.30}$, $2^{25.58}$, and $2^{25.80}$ rows to have a more detailed training set.

Implementations. For all our implementations, we use C++ and OpenMP for parallelism. In addition, we rely on OpenMP simd pragma for vectorization. We use Intel compiler v2021 with the *O3* and *vec* flags. During execution, all threads are pinned to physical cores, and 24 threads are used. We use *numactl -i all* to interleave memory allocation across NUMA nodes.

Model Verification. To verify our model, we use k-fold cross-validation with $k = 10$. In this case, ten separate training and test sets are formed from our initial matrix set with 1462 matrices. In each of these training and test set pairs, the matrices included are disjoint. Finally, to assess the accuracy of the method, we report the combined confusion matrices of 10-fold cross-validation.

4.6 EXPERIMENTAL RESULTS

In this section, we assess the performance of MVPP. First, we analyze the characteristics of our training set and show that it represents a large set of matrix characteristics. Second, we analyze the accuracy of predictions made by MVPP.

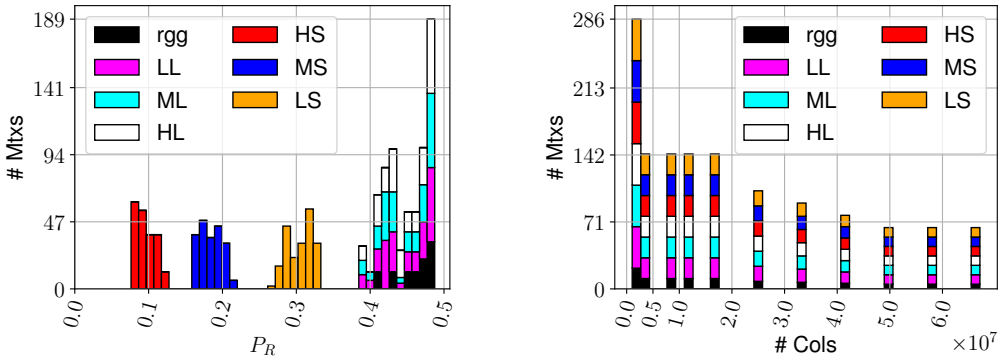
4.6.1 Characteristics of The New Matrix Set

We aim to create a more representative set of matrices covering a more comprehensive range of values for selected matrix features.

First, we consider the skew characteristics of the matrices and the number of columns. In Section 4.3, we observed that SuiteSparse matrices have a highly balanced number of nonzeros per row, and the number of columns of the matrices was limited. For these matrices,

we observe that the input vector of SpMV can fit in LLC and SELLPACK and Sell-c-σ gave the best performance for the majority of the cases. However, this set was not able to model a variety of characteristics. Figure 4.11 shows the p-ratio distribution of nonzeros to rows and the number of columns for randomly generated matrices described in Section 4.4.5.

We see that, in contrast to SuiteSparse matrices, the random matrix set covers a wide range of values for both features. For example, High-Skew (HS), Medium-Skew (MS), and Low-Skew (LS) matrices are clustered around P_R values of 0.1, 0.2, and 0.3, respectively. The wide range of p-ratios for the R distribution allows us to model matrices with highly unbalanced rows (lower p-ratio) to more balanced rows (higher p-ratio). Furthermore, the matrices chosen to model different locality characteristics (rgg, Low-Locality (LL), Medium-Locality (ML), and High-Locality (HL)) have a P_R value between $\approx 0.4 - 0.5$. Thanks to this balanced distribution of nonzeros to rows, we can model the effect of locality on SpMV performance without interference due to skew behavior.

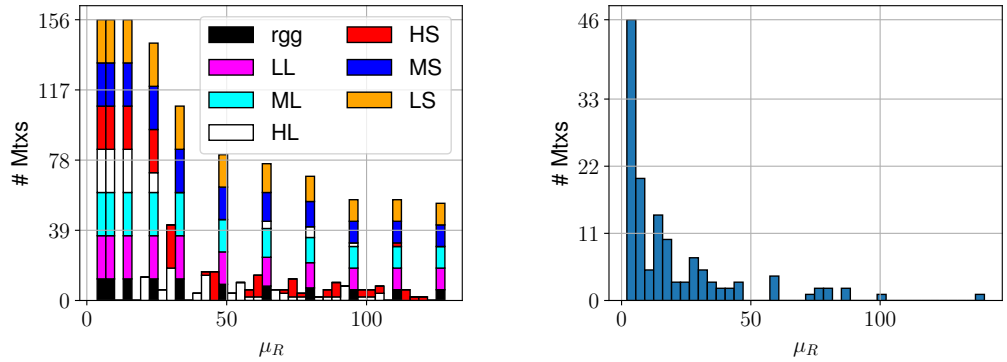


(a) Dist. of P_R for Random Matrices (b) Dist. of n_C for Random Matrices

Figure 4.11: Distribution of P_R and number of columns for random matrices.

Secondly, recall that the vectorization performance is also affected by the average number of nonzeros in rows. Figure 4.12 shows that while SuiteSparse matrices generally have 20 or fewer nonzeros per row on average, our randomly generated matrix set covers a significantly more comprehensive range. Since μ_r affect the number of elements in row chunks, a large number of elements in the row chunk means a higher utilization of vector units. The limited nature of the sample set from SuiteSparse is a limiting factor for an ML model which relies on examples to detect patterns.

Our new training set also offers more examples of locality behavior. The *uniqC* and *Gr8_uniqC* features show the potential for higher locality. A low ratio indicates high reuse. In contrast, a high ratio indicates a lower reuse. Figure 4.13 shows the distribution of *uniqC* and *Gr8_uniqC* for both randomly generated matrices and SuiteSparse matrices. Both



(a) Dist. of μ_R for Random Matrices (b) Dist. of μ_R for SuiteSparse

Figure 4.12: Comparison of average number of nonzeros per row (μ_R) for random and SuiteSparse matrices.

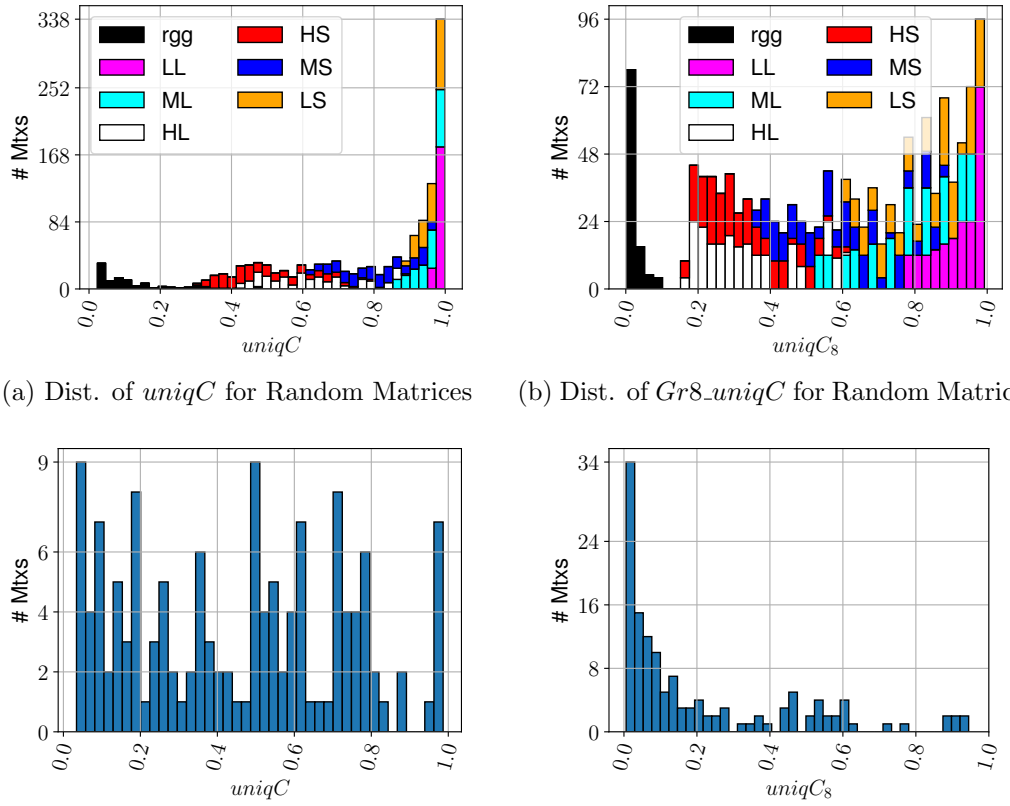
random and SuiteSparse matrices show a wide distribution for both features. However, in SuiteSparse (for the selected matrices), there are only a few examples for different values of *uniqC* and *Gr8_uniqC*. Also, note that *Gr8_uniqC*, which models the cache line size for double-precision numbers, has a skew towards lower *Gr8_uniqC*, while randomly generated matrices have a much wider spread.

4.6.2 Accuracy of MVPP

We first consider the accuracy of classification for MVPP. Figure 4.14 reports the confusion matrices for models generated for different SpMV <method, parameter> pairs. We show correct classes on the vertical axis, while predicted classes are shown on the horizontal axis. We only choose to report models generated by $c = 8$ since the $c = 4$ case behaves similarly.

An exact match of the correct and predicted class lies on the diagonal of these matrices. The accuracy of our model could be interpreted by observing the number of test cases that lie on the diagonal compared to the total number of test cases. However, such a simple classification accuracy metric is not representative because our classes are not entirely independent. Instead, they are a discretization of speedup values. For this reason, not all incorrect predictions are equal for MVPP.

As a result, we interpret our accuracy results in three dimensions. First, we consider accuracy with its traditional interpretation by reporting correct vs. incorrect predictions of MVPP. Second, we consider the distance between the predicted and the correct classes. Recall that the classes in MVPP correspond to 10% intervals of normalized execution time. Therefore, a distance of x between predicted and correct class means that estimation given by MVPP is within $x \times 0.1$ neighborhood of correct execution time. Third, we consider



(a) Dist. of *uniqC* for Random Matrices

(b) Dist. of *Gr*8_*uniqC* for Random Matrices

(c) Dist. of *uniqC* for SuiteSparse

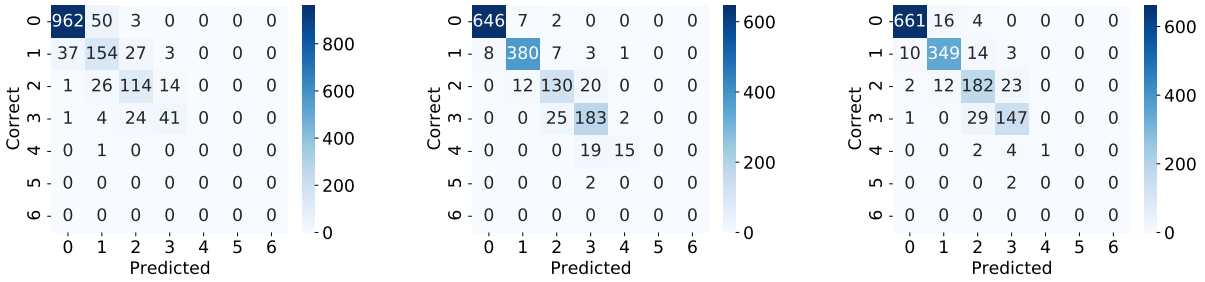
(d) Dist. of *Gr*8_*uniqC* for SuiteSparse

Figure 4.13: Distribution of *uniqC* and *Gr*8_*uniqC* for random and SuiteSparse matrices.

the possibility of overestimating the potential speedup. In this case, we consider the upper triangular part of the confusion matrix, which tells us that the predicted speedup is higher than the actual speedup that can be obtained by the SpMV method.

For SELLPACK, we observe that 15% of test matrices could see performance benefits. Among these matrices, 69% of them were classified correctly. However, the majority of predictions (28%) for misclassified matrices are classified as one-higher or one lower performance class (a distance of 1). Similarly, for Sell-c-σ, we observe an accuracy rate of 80%, 81%, and 86% for models for σ values of 2^9, 2^{12}, and 2^{14}, respectively. For all σ values, only 4-10% of mispredictions have a distance greater than 1 to the correct class.

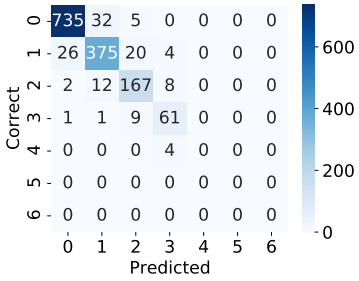
MVPP is also successful at predicting the performance of more complex SpMV methods. Figures 4.14e-4.14i show the accuracy of predictions for Sell-c-R, LAV-1Seg, and LAV methods. For LAV, we use T values of 70%, 80%, and 90%. We observe that Sell-c-R and LAV-1Seg have 82% and 63% accuracy rates, respectively. Similar to SELLPACK and Sell-c-σ, we observe that mispredictions are not far off from the correct classes. For example, although LAV-1Seg has only a 63% accuracy rate, 33% of classification have only distance



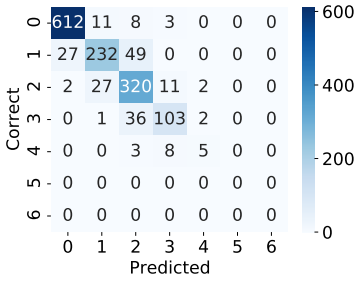
(a) SELLPACK

(b) Sell-c-σ ($\sigma = 2^9$)

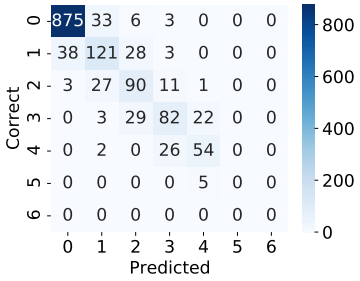
(c) Sell-c-σ ($\sigma = 2^{12}$)



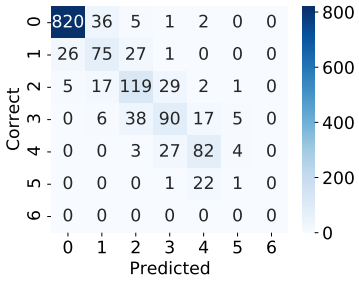
(d) Sell-c-σ ($\sigma = 2^{14}$)



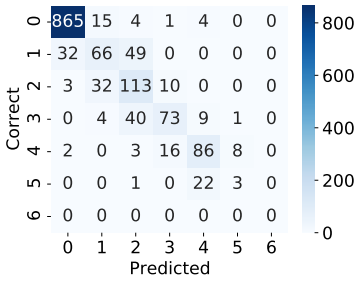
(e) Sell-c-R



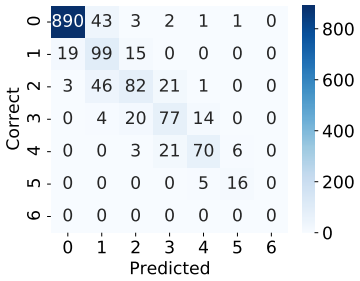
(f) LAV-1Seg



(g) LAV ($T = 70\%$)



(h) LAV ($T = 80\%$)



(i) LAV ($T = 90\%$)

Figure 4.14: Accuracy of the predictions for SELLPACK and Sell-c-σ. c parameter for all implementations are set to 8.

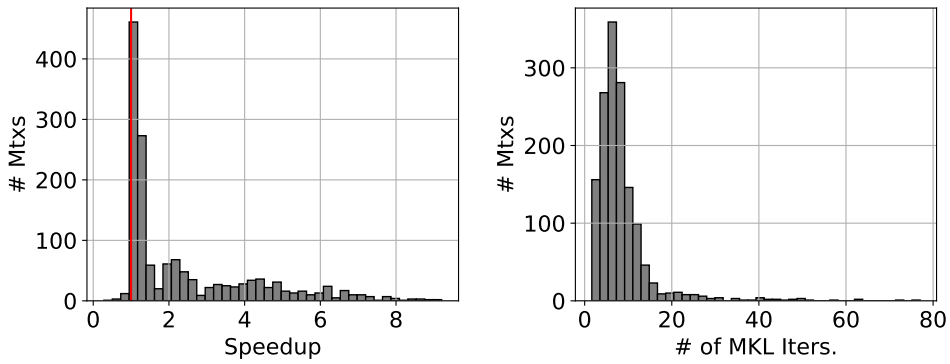
1 to the correct class.

MVPP also successfully predicts the performance of the LAV method, which is the most complex technique. In LAV, the ML model needs to consider the skew in the nonzero distribution of columns, rows, and the matrices' locality behavior. The accuracy rate of MVPP is 62, 64, 61 % for $T = 70\%$, $T = 80\%$, and $T = 90\%$ cases. Similar to LAV-1Seg, a vast majority of mispredictions (32-35%) are mispredictions with a distance of 1. Except for C5, MVPP generally predicts the performance accurately. However, for $T = 70\%$ and $T = 80\%$ cases, MVPP generally predicts C4 instead. The reason behind this is likely the limited number of C5 samples in the training sets.

4.6.3 Speedups Obtained

In this section, we consider the speedup that can be obtained by employing MVPP with respect to the MKL baseline. Figure 4.15a shows the distribution of speedups obtained by all matrices tested. In these experiments, we use a least-expensive first strategy to choose the method to use. The least expensive method with the best speedup class is selected as the SpMV strategy.

We observe that MVPP can achieve an average speedup of 2.4× compared to MKL baseline, achieving an equivalent or better execution time for almost all matrices. The harmonic mean of the speedup is 1.6×. The maximum speedup observed can be up to 9×. Furthermore, to compare against ground truth, we compare MVPP to an oracle method. Our oracle method finds the fastest method for each matrix. With the oracle, we observe that the average speedup is 2.5×. Additionally, we compare MVPP to the state-of-the-art vendor library MKL's inspector executor. We observe that the average speedup that we can obtain with inspector-executor is 2.11×. To summarize, MVPP is 1.13× faster than MKL and an oracle method can only achive 2.5× speedup (1.04× with respect to MVPP). When we consider SuiteSparse matrices alone, MVPP has an average speedup of 1.50× while MKL inspector-executor has an average speedup of 1.56×. However, as we will see next, MVPP has a lower overhead.



(a) Speedup (b) Overhead

Figure 4.15: Distribution of speedups of MVPP over MKL.

4.6.4 Preprocessing Overhead

We measure the preprocessing overhead by taking both feature calculation and format conversion into account. Figure 4.15b shows the distribution of the preprocessing overhead. We report the preprocessing in terms of number of SpMV iterations with MKL baseline.

We observe that the average overhead of MVPP is 8.33 iterations. On the other hand, the average overhead of MKL inspector-executor overhead is 17.43 iterations. Although MVPP considers many sparse matrix features and many SpMV methods, it still has less than 50% of MKL inspector-executor's preprocessing cost. Furthermore, the average overhead of MVPP for SuiteSparse matrices is only 17 iterations while MKL inspector-executor has an average overhead of 50 iterations.

4.6.5 Choosing ML Parameters

Finally, we consider the tuning of decision tree parameters. Our goal is to find the maximum depth and pruning (*ccp_alpha*) parameters to avoid overfitting. For maximum depth (D), we test values $D = \{5, 10, 15, 20\}$. For *ccp_alpha (ccp)*, we test $ccp = \{0, 0.001, 0.005, 0.01, 0.05, 0.1\}$. Increasing maximum depth also increases the potential for overfitting. In contrast, decreasing the *ccp_alpha* value increases the chance of overfitting. Table 4.5 shows the average speedups that we can obtain with different combinations of D and *ccp* parameters. We observe that MVPP can obtain 2.37× or more speedup for most combinations, especially if the ccp parameter is not too aggressive ($ccp \geq 0.05$). The D parameter doesn't have a significant impact. At the end, we choose $D = 15$ and $ccp = 0.005$ for MVPP.

Table 4.5: Average speedup obtained with decision tree model parameters. The rows of the Table show different maximum depth parameters (D), while the columns show the different *ccp_alpha* (*ccp*) parameters.

	0	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**
5	2.39	2.38	2.39	2.37	2.21	2.23
10	2.40	2.41	2.40	2.38	2.32	2.24
15	2.41	2.41	2.40	2.38	2.31	2.24
20	2.41	2.41	2.40	2.38	2.31	2.24

4.7 RELATED WORK

Auto-tuning approaches were previously evaluated [63, 94, 102]. However, MVPP is different from auto-tuning because it is an automated and extendable system. A new heuristic or mathematical model needs to be designed for every new method for model-driven auto-tuning systems. Previous work also utilizes machine learning for performance prediction [106, 107, 108, 109, 110, 111, 112]. They generally target selecting a preferable matrix format for executing SpMV more efficiently. MVPP differs from previous work in multiple aspects. First, instead of only choosing a format, MVPP makes a decision in a compre-

hensive optimization framework. Additionally, MVPP not only chooses optimizations but also predicts the best parameters to use for a given method. Other examples also exist for leveraging machine learning to predict performance for other BLAS primitives such as matrix-matrix multiplication [113]. Auto-tuning is also applied in graph processing context. These works mainly focus on algorithmic optimizations such as selecting push-pull execution modes based on active vertex lists during execution [114, 115].

There are many examples of locality optimizations for SpMV and graph processing. Cagra [88] preprocesses the graph, dividing it into smaller LLC-sized sub-graphs. Other examples of locality optimizations include binning techniques [61, 62]. For techniques such as [61, 62, 88], MVPP can be extended with new performance models. Milk [89] is a set of language extensions to improve the locality of indirect memory accesses, which can also be used for SpMV calculations. There are also compile-time and runtime techniques to accelerate programs with indirect memory accesses [90, 91]. Moreover, partitioning techniques can also be used for improving locality [92].

A large body of previous work targets relabeling vertices of graphs to provide better locality [68, 69, 70, 95]. These sophisticated techniques achieve high locality but incur large overheads. In this work, we only consider RFS and CFS reordering mechanisms. However, we think that MVPP can also be used as a first step for deciding whether to apply reordering techniques eliminating significant overhead if reordering will not improve performance.

Many different SpMV vectorization methods have been proposed [59, 60, 64, 65, 66, 67]. Their main aim is to maximize the vector unit utilization. Liu et al. [66] propose to use finite window sorting, which is similar to RFS but only considers a small block of rows. VHCC [65] devises a 2D jagged format for efficient vectorization of SpMV. MVPP's features are representative, skew and locality characteristics can effectively assess the potential of these vectorization techniques.

4.8 CONCLUSIONS & FUTURE WORK

In this work, we proposed MVPP, a multidimensional ML approach for predicting SpMV performance. MVPP combined intuition and a detailed characterization study to create a well-designed training set and a feature set to summarize matrix characteristics. MVPP is able to choose the best SpMV method and their parameters with high accuracy to find efficient techniques to execute SpMV for a wide range of matrices. We observe that MVPP can achieve an average of 2.4× speedup with respect to the MKL baseline. Furthermore, MVPP can achieve 1.13× speedup over MKL inspector-executor method with 50% less overhead.

We envision that MVPP will be used as the backend of GraphBLAS/BLAS frameworks such as Intel's MKL [78]. We observe that MVPP has lower preprocessing overhead than MKL's inspector-executor. Furthermore, overall application speedup also depends on the cost of preprocessing. Future work can extend MVPP with a preprocessing cost model to take the preprocessing cost into account while choosing the SpMV strategy.

CHAPTER 5: DENSE DYNAMIC BLOCKS: OPTIMIZING SPMM FOR PROCESSORS WITH VECTOR AND MATRIX UNITS

5.1 INTRODUCTION

Sparse Matrix Dense Matrix Multiplication (SpMM) is a fundamental building block for many complex applications such as linear solvers [5, 6], graph analytics [17], recommender systems, and machine learning [11, 12]. Most of these applications are also iterative in nature, executing SpMM many times with the same sparse input matrix. As a result, SpMM often consumes most of an application's execution time, making it an important target to optimize.

Various state-of-the-art CPU and GPU systems are now featuring matrix-multiply hardware facilities tailored for dense matrix operations. These specialized units can execute dense matrix-multiply operations on small matrices (e.g., blocks of size 4×4). Examples of these are NVIDIA's Tensor Cores [23, 24], IBM's POWER10 Matrix-Multiply Assist (MMA) facilities [116, 117], and Intel's AMX [118]. These units are successfully utilized to maximize performance for dense matrix operations [23, 24, 25].

However, many real-world matrices have sparsity. For example, in the domain of neural network training and inference, up to 90% of matrix entries can be zero due to activation function outputting zero or weights becoming zero [119, 120]. Due to this dynamic behavior and limited sparsity, current work represents the matrices as dense matrices and tries to skip unnecessary computations involving elements that are zero or become zero during the training phase [23, 121]. In contrast, in the domains of graph analytics and scientific computing, matrices are often extremely sparse. As a result, they are represented in sparse formats and result in very irregular computations. In this work, we focus on SpMM with these extremely sparse matrices.

Previous work on speeding up SpMM for extremely sparse matrices using matrix-multiply hardware facilities includes Blocked CSR (BCSR) or its variations, as tried for GPU Tensor Cores [122, 123]. BCSR can create $r \times c$ dense blocks from consecutive rows and columns of the sparse input matrix. However, such blocks are hard to find in sparse matrices, and dense blocks that contain many zeros cause under-utilization of the matrix-multiply units' throughput.

These previous approaches were designed to improve the irregular accesses in Sparse Matrix-Vector Multiplication (SpMV) [63, 94, 124, 125, 126] and, more recently, to utilize Tensor Units for SpMM on GPUs [122]. However, these techniques rely on finding consecutive rows that show a similar nonzero structure within a consecutive range of columns. Finding such rows typically requires expensive mechanisms. Fortunately, SpMM is a more

computationally intensive operation than SpMV. In SpMM, a single nonzero causes an access to a whole row of the input dense matrix, instead of just a single element from the dense vector in SpMV. Therefore, it is less crucial to form dense blocks from consecutive columns.

We propose to use a more relaxed dynamic approach for blocking: Dense Dynamic Blocks (DDB). We extract $R \times 1$ blocks from consecutive R rows of the sparse matrix. These $R \times 1$ blocks are used to create a dynamically sized dense block to execute on matrix units. This approach, called DDB-MM, can still suffer from zero padding and underutilize the floating-point throughput of the processor. To address this issue, we propose a novel approach in which computations in SpMM are synergistically executed on matrix units or on traditional vector units to maximize the floating-point throughput. We call our scheme DDB Hybrid (DDB-HYB). DDB-HYB extracts variable-sized dense blocks from the sparse input matrix. In DDB-HYB, the matrix-multiply units process those blocks with a large fraction of nonzero elements; the remaining, sparser blocks use traditional vector units to maximize performance.

For many sparse matrices, the number of dense blocks extracted will be minimal or the dense $R \times 1$ blocks will have many zeros in them. Therefore, they will not benefit from DDB-MM or DDB-HYB significantly. For this reason, we also propose a simple metric, *Average Flop Throughput* (AFT), to assess the efficacy of DDB-HYB for a given matrix before its application. AFT can classify the matrices at a high level based on their potential to get benefits from matrix-multiply units. This coarse-grain classification can limit the search space, but it does not always choose the best SpMM strategy.

For this reason, we develop *SpMM-OPT*. SpMM-OPT is a machine learning approach with a carefully crafted feature set to select the best SpMM strategy in a functional unit-rich environment. It can choose between a vector unit oriented approach (CSR), a matrix unit oriented approach (DDB-MM), and a hybrid approach (DDB-HYB). It can also decide on the cache optimizations by selecting an appropriate slicing factor for the dense matrices of SpMM.

We validate the performance of DDB-MM and DDB-HYB on a large selection of matrices from the SuiteSparse matrix collection. First, we establish an efficient baseline with CSR implementation. Then, we assess the speedup of DDB-MM and DDB-HYB compared to this baseline. We observe that DDB-MM and DDB-HYB can achieve up to 1.1 TFlops/s (corresponding to 40% of LINPACK throughput) and 2.4 TFlops/s for double- and single-precision SpMM, respectively, on a single-chip POWER10 processor. Finally, we analyze the effectiveness of SpMM-OPT to create efficient SpMM strategies. We show that SpMM-OPT is able to find effective SpMM strategies achieving high floating-point throughput with respect to the baseline CSR implementation.

5.2 BACKGROUND

In this work, we use upper case letters for matrices (e.g., $\mathbf{A}$ and $\mathbf{B}$). $\mathbf{A}[i, j]$ is the element in row i, column j of matrix $\mathbf{A}$. To represent a slice of a matrix, we use the $\mathbf{A}[a : b, x : y]$ notation, which gives elements of $\mathbf{A}$ in rows a to b and columns x to y. A slice without any boundaries gives all the elements in that dimension. For example, $\mathbf{A}[k, :]$ represents all the elements in row k of matrix $\mathbf{A}$.

An SpMM kernel multiplies a sparse $M \times T$ matrix $\mathbf{A}$ with a dense $T \times N$ matrix $\mathbf{B}$, generating a dense $M \times N$ matrix $\mathbf{C}$. A row i of matrix $\mathbf{C}$ is updated with the computation $\mathbf{C}[i, :] = \sum_j \mathbf{A}[i, j] \times \mathbf{B}[j, :]$. Since matrix $\mathbf{A}$ is sparse, a row of $\mathbf{A}$ ($\mathbf{A}[i, :]$) has very few nonzeros. These sparse matrices require a compact representation.

This section describes the Compressed Sparse Row (CSR) representation of sparse matrices and its SpMM implementation, and discusses the Matrix-Multiply Assist (MMA) instructions of the IBM POWER10, used for validating our ideas.

5.2.1 CSR Format and SpMM Implementation

Large sparse matrices require compact in-memory representation. The compressed sparse row format (CSR) (or variants) is the most popular choice for basic linear algebra, graph processing, and machine learning frameworks [16, 17, 78]. We repeat the description of CSR format alongside its SpMM implementation to make this chapter self-contained.

CSR uses three arrays to represent a sparse matrix: `vals`, `col_id`, and `row_ptr`. The `vals` array stores all of the nonzero elements in the matrix. The `col_id` array stores the column index of each nonzero in the `vals` array. The `row_ptr` array stores the starting position in the `vals` and `col_id` arrays of the first nonzero element in each row of the sparse matrix. In this work, we use CSR as our baseline.

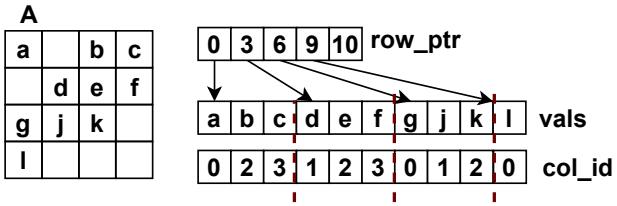


Figure 5.1: CSR format.

CSR is popular due to its ability to express sparse matrix computations effectively. Algorithm 5.1 shows the implementation of SpMM with CSR format.

Algorithm 5.1 iterates over the rows of matrix $\mathbf{A}$ in parallel (Line 1). For each row, it iterates over its nonzeros in Line 2. Finally, Line 3 iterates over the columns of the dense $\mathbf{B}$

Algorithm 5.1 Implementation of CSR SpMM (CSR-A).

```
1: for (i=0; i<M; i++) in parallel do
2:     for (e=row_ptr[i]; e<row_ptr[i+1]; e++) unroll(K) do
3:         for (j=0; j<N; j++) unroll(P) do
4:             C[i][j] = C[i][j] + (vals[e] × B[col_id[e]][j])
5:         end for
6:     end for
7: end for
```

matrix and updates the values of the **C** matrix.

Although even a straightforward implementation can use the vectorization facilities of current hardware, we can improve SpMM performance by utilizing vector registers and vector units more effectively through code transformations. Algorithm 5.1 shows two loop unrolling opportunities: (1) unrolling the loop in Line 2 by a factor of `K`, and (2) unrolling the loop in Line 3 by a factor of `P`. By unrolling the loop in Line 2, we can process multiple nonzeros of matrix **A** at the same time. It allows us to keep values of **C** in vector registers, decreasing the number of store operations for **C**. On the other hand, by unrolling the loop in Line 3, we improve the throughput of vector instructions by creating more parallelism across columns of the **B** and **C** matrices. Algorithm 5.1 focuses on reusing the values of **A**, which we name as CSR-A. Another approach is to maximize the reuse for **C**. Algorithm 5.2 shows how register blocking can be achieved for values of **C**, which we name CSR-C. In CSR-C, P values of row i of **C** are kept in registers while processing all nonzeros of row i of **A**. In the CSR-C version, **C** is kept in vector registers by sacrificing the reuse for **A**.

Algorithm 5.2 Implementation of CSR SpMM with reuse for **C** (CSR-C).

```
1: for (i=0; i<M; i++) in parallel do
2:     for (p=0; p<N; p+=P) do
3:         for (e=row_ptr[i]; e<row_ptr[i+1]; e++) unroll(K) do
4:             for (j=p; j<p+P; j++) unroll(P) do
5:                 C[i][j] = C[i][j] + (vals[e] × B[col_id[e]][j])
6:             end for
7:         end for
8:     end for
9: end for
```

5.2.2 POWER10 Matrix-Multiply Unit: MMA

Without loss of generality, we target hardware that is capable of executing $\mathbf{A}[0 : R-1, 0 : K-1] \times \mathbf{B}[0 : K-1, 0 : P-1]$ dense matrix multiplication (or $R \times K \times P$, for short), either directly or through primitives that can be combined to do so. Such hardware provides two sets of primitives: (1) instructions to move data to/from accumulators used during the matrix multiply operation, and (2) instructions that perform a matrix-multiply or a matrix-multiply and accumulate operation.

Our testbed is a POWER10 system, and we summarize how an $R \times K \times P$ matrix-multiplication can be implemented with MMA capabilities. MMA provides 8 accumulators, each holding either a 4×4 or 4×2 matrix of single- or double-precision elements, respectively. Among other operations, MMA can perform outer-products of single- and double-precision vectors, with the results added to an accumulator. Table 5.1 shows a summary of the MMA instructions.

Table 5.1: Summary of relevant MMA instructions.

Instruction	Description
xxmtacc(acc[k])	Moves values from vector registers to accumulator k
xxmfacc(acc[k])	Moves values from accumulator k to vector registers
xxsetaccz(acc[k])	Set values of accumulator k to zero
xvf32gerpp(x, y, acc[k])	Performs a 4x4 single-precision outer-product with x[0:3] (vector register) and y[0:3] (vector register), and accumulates the result on accumulator k
xvf64gerpp(x, y, acc[k])	Performs a 4x2 double-precision outer-product with x[0:3] (vector register pair) and y[0:1] (vector register), accumulating the result on accumulator k

The **xxmtacc** instruction moves data from four vector registers to an accumulator, while **xxmfacc** does the opposite, and **xxsetaccz** clears an accumulator. The single-precision **xvf32gerpp** instruction performs a 4×4 outer-product of two 4-element vectors in registers x and y and adds the result to the current value in an accumulator. The corresponding double-precision **xvf64gerpp** instruction performs the outer-product of a 4-element vector in register-pair x and a 2-element vector in register y, adding the result to an accumulator.

A (double-precision) matrix multiply is performed as follows. First, we initialize an accumulator by either transferring values from vector registers to the accumulator (**xxmtacc**) or setting all accumulator values to zero (**xxsetaccz**). Then, we perform outer products and accumulate the values of a 4×2 dense matrix by using **xvf64gerpp** instructions. Finally, we use **xxmfacc** to transfer the values from the accumulator to vector registers, which will contain the result of the matrix multiplication [116]. Thanks to their fine-grained design, MMA instructions can be used to implement small dense matrix multiplications with various sizes. As an example, Figure 5.2 shows an implementation for a $4 \times 4 \times 4$ matrix-multiplication and its code sequence.

5.3 OPTIMIZING SPMM

In this section, we describe how we can reformulate SpMM to utilize both matrix and vector units and how our Dense Dynamic Blocks approach can leverage the discrepancy in

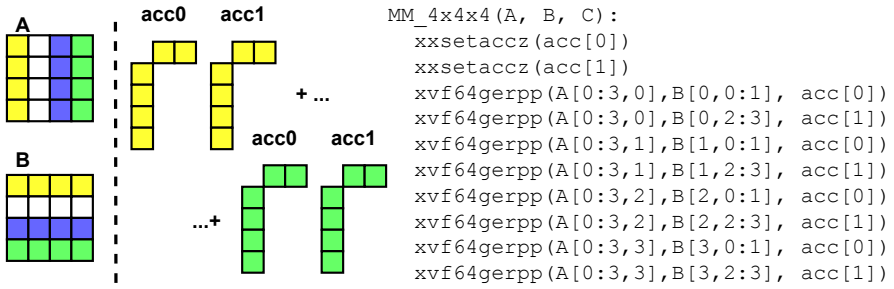


Figure 5.2: A $4 \times 4 \times 4$ matrix-multiplication of $\mathbf{A} \times \mathbf{B}$, and its code sequence using two 4x2 accumulators. Matrix $\mathbf{A}$ is stored in column-major order, while matrix $\mathbf{B}$ is in row-major order. The output will be in column-major order.

effective floating-point throughput of matrix and vector units to maximize performance.

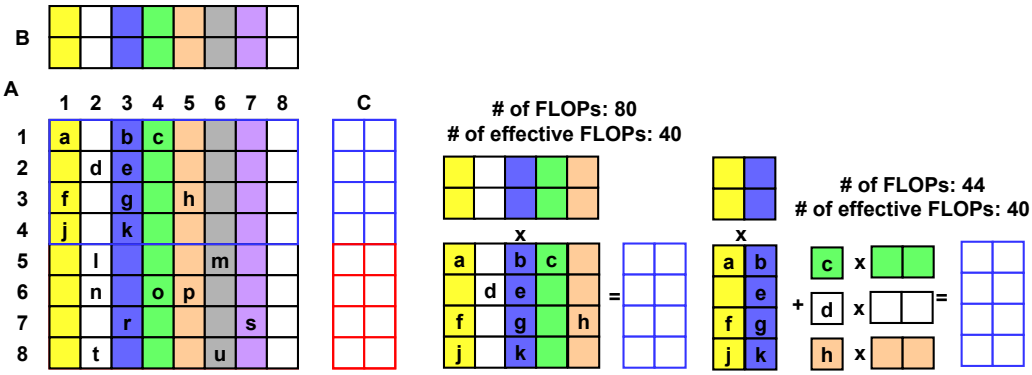
Sparse matrices vary significantly in terms of nonzero structure. While some matrices can greatly benefit from utilizing matrix units with dense $R \times 1$ blocks, others can get hurt. Therefore, choosing a good SpMM strategy is crucial. In section 5.3.3, we describe the optimization search space for SpMM in the existence of a functional unit-rich processor. Furthermore, we develop *SpMM-OPT*, a machine learning approach with a carefully crafted feature set to select the best SpMM strategy in a functional unit-rich environment. We describe SpMM-OPT in Section 5.3.4.

5.3.1 Reformulating SpMM for Matrix-Multiply Units

A matrix-multiply unit is capable of executing an $R \times K \times P$ matrix-multiply operation, where typical dimensions of R, K, and P are 2–16. Furthermore, we expect that a unit can handle variable-sized matrix-multiply operations. In the rest of this section, we use a unit that provides $4 \times 4 \times 2$ matrix-multiply operations and can also handle $4 \times 2 \times 2$, and $4 \times 1 \times 2$.

The two dynamic blocking approaches we discuss are DDB-MM and DDB-HYB. As an example, consider Figure 5.3a. It shows an 8×8 sparse matrix ($\mathbf{A}$) and two 8×2 dense matrices ($\mathbf{B}$ and $\mathbf{C}$). Note that $\mathbf{B}$ is shown transposed to emphasize the access patterns for $\mathbf{B}$. The rows of $\mathbf{B}$ and columns of $\mathbf{A}$ that will be accessed together are shown with the same colors.

First, we consider the need for finding blocks with consecutive columns. DDB-MM relaxes this requirement by identifying $R \times 1$ blocks from R consecutive rows of the sparse matrix. These $R \times 1$ blocks are then used to create a larger dense block (an $R \times n$ dense matrix from n $R \times 1$ blocks). Figure 5.3b shows this approach for the first $R = 4$ rows of the sparse matrix $\mathbf{A}$. Such an approach can utilize the high floating-point throughput of matrix



(a) Matrices for $C = A \times B$. (b) DDB-MM (c) DDB-HYB

Figure 5.3: Performing an $8 \times 8 \times 2$ SpMM with 4×1 blocks.

units. However, DDB-MM can still have drawbacks. If not all $R \times 1$ blocks are dense we will introduce zero padding, which results in underutilization of the floating-point throughput of the processor.

As an example, consider IBM POWER10. A POWER10 core has two pipelined MMA units, each capable of executing 4×2 double-precision outer-product instructions. Let us consider only the case of double-precision. If all the column elements are nonzero, the MMA units of a POWER10 core deliver an aggregate throughput of 32 effective FLOPs/cycle. For the POWER10 MMA, if the columns have 1, 2, or 3 zero elements, the effective throughput falls to 24, 16, and 8 FLOPs, respectively. However, a POWER10 core also has four pipelined vector units, each capable of computing a 1×2 (double-precision) outer-product for an aggregate throughput of 16 effective FLOPs per cycle. Note that there is no need to process zeros introduced due to padding in the vector units. Therefore, it only makes sense to process columns with 2 or more nonzero elements in the MMA where we would obtain 16 or more effective FLOPs per cycle. This suggests that we should process some columns of the sparse matrix in the MMA units and some columns in the vector units. The aforementioned observations still hold for single precision operations, where MMA units can execute 4×4 outer product instructions and vector units can execute 1×4 outer product instructions. Our hybrid approach (DDB-HYB) takes advantage of this throughput discrepancy.

Figure 5.3c shows how we can change the matrix-multiply sequence to improve throughput and reduce padding for the first four rows of input matrix **A**. We separate the processing of high and low-density columns. The columns with high density (yellow and blue) use matrix-multiply units, while the remaining columns use vector units. This approach reduces the number of superfluous FLOPs significantly. For example, 4×1 blocking shown in Figure 5.3b would perform 80 FLOPs, while DDB-HYB approach would perform 44 FLOPs for the given 4-row block, which only needs 40 FLOPs.

5.3.2 Dense Dynamic Blocks: DDB-MM and DDB-HYB

Our Dense Dynamic Blocks (DDB) technique is built on the previous observations. DDB examines the structure of a sparse matrix and finds $R \times 1$ dense blocks, which we call *dense column blocks*. In DDB-MM, each group of R consecutive rows is represented solely as $R \times 1$ dense blocks, independent of the amount of padding necessary. In DDB-HYB, we categorize these dense column blocks in terms of their floating-point throughput potential into high throughput (*htp*) and low throughput (*ltp*). For the POWER10 MMA unit, column blocks with 2 or more nonzeros are categorized as *htp*, while column blocks with a single nonzero are *ltp*. DDB-HYB then splits the sparse matrix into a *blocked* submatrix and a *compressed* submatrix. The former is composed of *htp* dense column blocks, which can leverage the MMA units; the latter is composed of the *ltp* dense column blocks and is represented in a CSR-based format to utilize vector units and maximize floating-point throughput. Finally, DDB-HYB forms large, multi-column dense blocks in the blocked submatrix during execution, based on the capabilities of the matrix-multiply units. Note that CSR is the base case for DDB in the absence of a dense portion. In the rest of this section, we explain the matrix format and SpMM implementation for only DDB-HYB. DDB-MM can be obtained by ignoring the data structures and operations required for the sparse portion of DDB-HYB.

DDB-HYB's Matrix Format. Given an $M \times T$ sparse matrix, DDB-HYB creates $R \times T$ submatrices that contain R consecutive rows each. In each submatrix, it finds the *htp* and *ltp* column blocks. An *htp* column block is represented with a dense $R \times 1$ array. In an *htp* column block, an element not present in the original sparse matrix is represented as a zero (or the annihilator of SpMM). The *htp* column blocks are then placed one next to the other. Since the column numbers of these *htp* column blocks are non-contiguous, we need to record the *ids* of the column blocks. Therefore, like in CSR, we use an `Offset` array of size $\lceil M/R \rceil$ to point to the beginning of the *htp* column blocks and column ids from each $R \times T$ submatrix. This is the representation of the *blocked* submatrix obtained from the original sparse matrix. As an example, the left-most part of Figure 5.4 shows the corresponding representation obtained from matrix **A** given in Figure 5.3a.

The *ltp* column blocks constitute the *compressed* submatrix obtained from the original sparse matrix and are represented using the CSR format. The rightmost part of Figure 5.4 shows the corresponding representation obtained from matrix **A**.

SpMM with DDB-HYB. With our technique, we can cast an SpMM into a set of $R \times K \times P$ matrix multiplications. We propose two algorithms with different objectives: MMA-A (Algorithm 5.3) and MMA-C (Algorithm 5.4). The former improves the reuse for blocks of **A**; the latter minimizes stores to **C** and transfers between accumulators and vector registers.

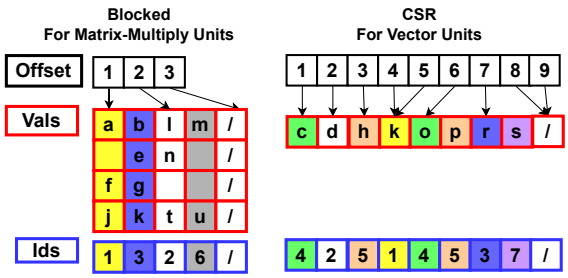


Figure 5.4: Representing the matrix in Figure 4.1a with DDB-HYB.

Algorithm 5.3 MMA-A: DDB SpMM Improving Block Reuse.

```
1: procedure SpMM(A, B, C)
2:     // A: M × T sparse matrix
3:     // B: T × N dense matrix, C: M × N dense matrix
4:   for (r=0; r<M; r+=R) in parallel do
5:       // Processing blocked portion for rows r to r+R
6:     for (k=offset[r]; k<offset[r+1]; k+=K) do
7:         cols = ids[k:k+K] // Array of column ids
8:         aRxK = vals[k*R:(k+K)*R]
9:         for (p=0; p<N; p+=P) do
10:            cRxP = C[r:r+R, p:p+P]
11:            // B matrix is sliced with column ids
12:            bKxP = B[cols, p:p+P]
13:            moveToAcc(cRxP) // Transfer to accumulator
14:            MM_RxKxP(aRxK, bKxP, cRxP)
15:            moveFromAcc(cRxP) // Transfer from accumulator
16:            C[r:r+R, p:p+P] = cRxP
17:        end for // remainder loop omitted
18:    end for  // remainder loop omitted
19:      // Processing compressed portion for rows r to r+R with CSR
20:  end for
21: end procedure
```

In both algorithms, we parallelize over row blocks ($R \times T$ submatrices) of sparse matrix **A**. We first process the dense blocks of the corresponding rows (Lines 6-18 for Algorithm 5.3 and Lines 6-18 for Algorithm 5.4). Then, in both algorithms, we process the compressed portion of the same set of rows using the CSR representation. Note that both CSR-A (Algorithm 5.1) and CSR-C (Algorithm 5.2) can be used for the compressed portion. This part is omitted for brevity.

In Algorithm 5.3, for each row block, we retrieve K consecutive dense column blocks of R rows of matrix **A**, forming an $R \times K$ dense block (Line 8). We iterate over the columns of the dense matrices, loading the corresponding dense block of **C** (Line 10) and dense dynamic block of **B** (line 12). Since the column ids in matrix **A** are not consecutive, loading the dense block from **B** is a specialized slicing operation. The block of **B** is loaded using the column ids (`cols`). Next, we move the values of **C**'s block to the accumulator and perform an $R \times K \times P$ matrix-multiply. Finally, the values are transferred back from the accumulator, and **C** is updated in memory.

Algorithm 5.4 is similar to Algorithm 5.3, except that it first iterates over the columns of

Algorithm 5.4 MMA-C: DDB SpMM Minimizing Transfers.

```
1: procedure SpMM(A, B, C)
2:     // A: M × T sparse matrix
3:     // B: T × N dense matrix, C: M × N dense matrix
4:   for (r=0; r<M; i+=R) in parallel do
5:       // Processing blocked portion for rows r to r+R
6:      for (p=0; p<N; p+=P) do
7:          cRxP = C[r:r+R, p:p+P]
8:          moveToAcc(cRxP) // Transfer to accumulator
9:          for (k=offset[r]; k<offset[r+1]; k+=K) do
10:             cols = ids[k:k+K] // Array of column ids
11:             aRxK = vals[k*R:(k+K)*R]
12:              // B matrix is sliced with column ids
13:             bKxP = B[cols, p:p+P]
14:             MM_RxKxP(aRxK, bKxP, cRxP)
15:         end for // remainder loop omitted
16:         moveFromAcc(cRxP) // Transfer from accumulator
17:         C[r:r+R, p:p+P] = cRxP
18:     end for  // remainder loop omitted
19:      // Processing compressed portion for rows r to r+R with CSR
20:   end for
21: end procedure
```

B and **C**. Therefore, it first loads the block of **C** and transfers it to the accumulator (Lines 7-8). Note that this approach increases the number of loads for blocks of **A** (Line 11) while minimizing the number of transfers to and from accumulators.

Choosing the Dimensions R, K, and P. The dimensions of the matrix-multiply operation are determined by both the hardware and the structure of the sparse matrix. R is determined by the first dimension of matrix-multiply operation. For instance, a POWER10 core has 8 4×2 (4×4) double-precision (single-precision) accumulators. Therefore, we choose $R = 4$ for our matrix-multiply kernel and for the number of consecutive rows that we can use to extract dense column blocks. We observe that $R = 4$ matches both the hardware parameters and is a good size to find dense column blocks in sparse matrices, as also noted by previous work [126]. P is mainly driven by the hardware. Since a POWER10 core has 8 accumulators, we can process up to 16 (32) columns of **C** ($P = 16$ for double and $P = 32$ for single-precision).

Finally, K is similar to an unrolling factor. It is driven by two considerations: (1) the capacity of the vector registers and (2) the number of dense column blocks in the blocked portion of the R-row sparse submatrix. Thanks to its dynamic nature, DDB-HYB can handle matrices that have any number of dense column blocks in individual R-row sparse submatrices.

5.3.3 SpMM Optimization Problem

CSR, DDB-HYB, and DDB-MM lie on a spectrum of optimizations. For example, DDB-MM addresses the needs of matrices with highly regular structures where consecutive rows

have similar nonzero structures. In contrast, CSR is needed when we have highly irregular nonzero distributions for consecutive rows of the matrix. And DDB-HYB strikes a balance between CSR and DDB-MM, maximizing floating-point throughput while minimizing zero padding.

In addition to the effective utilization of functional units available in the processor, we need to consider the reuse approach: CSR-A vs. CSR-C for compressed portion and MMA-A vs. MMA-C for blocked portion of the sparse matrices. With the addition of multiple reuse approaches, SpMM execution search space becomes even more complex.

SpMM performance is also affected by cache behavior. The nonzero structure of the sparse matrix and the scheduling of rows and row-blocks to threads affect the cache behavior. For example, if there are many common column ids among the rows or row-blocks, we can observe temporal locality for the dense input matrix. This locality is limited by the number of columns in the dense input and output matrices due to L1 and L2 cache sizes. For instance, a sparse matrix may observe high floating-point throughput when multiplied by a dense matrix with 32 columns. However, throughput can significantly decrease if the number of columns in the dense matrix is increased to 64 due to cache capacity. This is where *cache slicing becomes effective*.

By slicing the input ($\mathbf{B}$) and output ($\mathbf{C}$) dense matrices into blocks of columns, we can reduce the memory footprint for the two dense matrices during a single iteration and utilize caches more effectively. As a result, we can do multiple passes over the sparse matrix with a higher floating-point throughput, maximizing the overall performance.

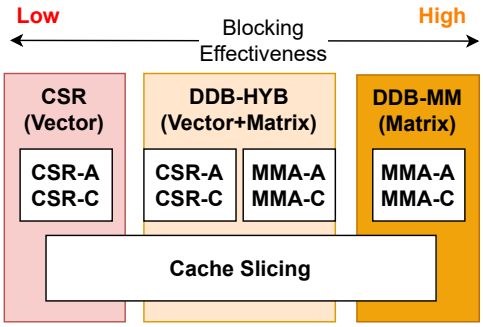


Figure 5.5: SpMM Optimization Space

Figure 5.5 shows a summary of the optimization search space for SpMM. Next, we discuss the details of SpMM-OPT that can navigate this complex search space to identify the best SpMM strategy for a given ⟨sparse matrix, dense matrix⟩ pair.

5.3.4 SpMM-OPT: An ML Approach for Method Selection

SpMM-OPT finds an SpMM execution strategy for a given pair ⟨sparse matrix, dense matrix⟩. The strategy is defined by a tuple ⟨`FU_St`, `Blocked_St`, `Compressed_St`, `SF`⟩. `FU_St` is the strategy for utilizing the functional units (either CSR, DDB-MM, or DDB-HYB). `Blocked_St` is the reuse approach to process the blocked portion of the sparse input matrix. It is only applicable for DDB-MM and DDB-HYB approaches, and it can have values in {`MMA-A`, `MMA-C`}.

`Compressed_St` is the reuse strategy for the compressed portion and is used for CSR and DDB-HYB approaches. The possible values of `Compressed_St` are in {`CSR-A`, `CSR-C`}. `SF` is the slicing factor, the number of columns in the blocks of the dense matrices.

SpMM-OPT works as follows. We detect the potential of matrices by using the Average Floating Point Throughput (AFT) metric. AFT categorizes each matrix into high AFT (HAFT) and low AFT (LAFT) categories. HAFT matrices have a higher potential to benefit from hybrid or matrix unit-oriented techniques DDB-HYB and DDB-MM. In contrast, LAFT matrices would prefer the CSR method. SpMM-OPT trains separate machine learning models for HAFT and LAFT matrices with different numbers of columns in the dense matrix. In this work, we consider that $D \in \{16, 32, 64, 128, 256\}$ can be the number of columns in the dense matrix, each corresponding to a power of two cache lines for double- and single-precision values. Since $\texttt{SF} \leq D$, each model has a different number of classes. For example, if $D = 64$ we can only have slice size $\texttt{SF} \in \{16, 32, 64\}$. The given $\langle$sparse matrix, dense matrix$\rangle$ pair is run through the corresponding machine learning model, obtaining $\langle \texttt{FU\_St}, \texttt{Blocked\_St}, \texttt{Compressed\_St}, \texttt{SF} \rangle$ tuple for an efficient SpMM strategy. Each of these machine learning models is a Support Vector Machine (SVM) which we optimized the parameters with a grid search.

Detecting Sparse Matrices with Potential. Sparse matrices show widely different characteristics in terms of the distribution of nonzeros. The AFT metric considers the distribution of a sparse matrix's nonzeros to dense column blocks with different density levels.

To compute AFT, we first create a density histogram H with R elements. H has R elements because a single column block can have 1 to R nonzeros. A single entry in the density histogram H_i gives us the percentage of nonzeros that reside in a column block with density level i, where $1 \leq i \leq R$. Using the density histogram, we calculate the AFT: $AFT = \sum_{i=1}^{R} H_i * t_i$, where t_i is the effective FLOP throughput that we can achieve when a column block has i nonzeros. For example, with POWER10 MMA units, we can have up to 4 elements in a single column block ($R = 4$). Therefore, both H and t will have 4 elements.

For double-precision floating-point values, we will use MMA units' outer-product instructions for column blocks with 2-4 nonzeros in them, which can achieve $t_2 = 16$, $t_3 = 24$, and $t_4 = 32$ FLOPs/cycle, respectively. On the other hand, we will use vector instructions for the column blocks with a single element, with throughput $t_1 = 16$ FLOPs/cycle. For single-precision, the throughput is doubled.

If the AFT is above a certain threshold, we classify matrices as HAFT. Otherwise, we classify them as LAFT. This threshold is chosen to be 20 experimentally by performing a

sensitivity analysis.

In addition to H distribution, we also calculate the distribution of dense column blocks with varying densities: T. In this case, T_i gives us the percentage of dense column blocks with i nonzeros among all possible dense column blocks.

System Features. We categorize the features of the machine learning system into three categories: (1) general, (2) blocking, and (3) locality features.

(1) General Features: These features model the size characteristics of a given sparse matrix. We use the number of rows and the number of nonzeros in the matrix.

(2) Blocking Features: These features model the blocking behavior. As described in AFT calculations (Section 5.3.4), we first find the density of each 4×1 block of the matrix by considering 4-row blocks. Each element of H and T distributions becomes a parameter in the SpMM-OPT.

(3) Locality Features: We consider two sets of locality features. The first set is $\texttt{uniq4}_X$. To calculate $\texttt{uniq4}_X$, we consider all 4 consecutive rows of the matrix as a single row and all X consecutive columns of the matrix as a single column. Then we map the nonzeros of the original sparse matrix onto this new compressed matrix by eliminating repeated nonzero entries. The ratio of the number of nonzeros observed in the compressed and the number of nonzeros of the original sparse matrix gives us the $\texttt{uniq4}_X$ value. The goal of this feature is to capture the overlap between irregular accesses of the blocking approach. The second set that we calculate is $\texttt{uniq256}_X$, in which we consider 256 consecutive rows as a single row. With $\texttt{uniq256}_X$, we can calculate the overlap between the irregular accesses due to scheduling 256 rows of the sparse matrix onto a single thread. We use $X = \{1, 4, 8, 16, 32, 64\}$ values to capture different prefetching characteristics.

In addition to $\texttt{uniq4}_X$ and $\texttt{uniq256}_X$ features, we also use the average number of nonzeros of CSR, blocked and compressed portions of DDB-HYB, and DDB-MM as indicators of cache capacity utilization. Table 5.2 summarizes the features we use and their names.

Table 5.2: Summary of Features.

Feature	**Description**
nrows,nvals	Encode the size characteristics of the matrix
H_i	The distribution of the nonzeros to dense column blocks
T_i	Density distribution of dense column blocks
$\texttt{aveBlked}_{\texttt{DDB-HYB}}$	Average number of nonzeros in the blocked portion of DDB-HYB
$\texttt{aveComped}_{\texttt{DDB-HYB}}$	Average number of nonzeros in the compressed portion of DDB-HYB
$\texttt{ave}_{\texttt{CSR}}$	Average number of nonzeros per row
$\texttt{ave}_{\texttt{DDB-MM}}$	Average number of nonzeros in DDB-MM
$\texttt{uniq4}_X$	Unique access ratio for blocks
$\texttt{uniq256}_X$	Unique access ratio for scheduling chunk

Normalization of features. Since we use SVMs as our machine learning mechanism,

we need to normalize the features to $[0, 1]$ range. Four of our features are already ratios normalized to this range: $\texttt{uniq4}_X$, $\texttt{uniq256}_X$, H_i, and T_i.

The $\texttt{aveBlked}_{\texttt{DDB-HYB}}$, $\texttt{aveComped}_{\texttt{DDB-HYB}}$, $\texttt{ave}_{\texttt{CSR}}$, and $\texttt{ave}_{\texttt{DDB-MM}}$ features can have significantly different values depending on the structure of the matrix. For these variables, we apply one-hot encoding with small changes. First we create a 9-bit one hot encoded representation of these variable by considering the ranges: $[0, 2)$, $[2, 4)$, $[4, 8)$, $[8, 16)$, $[16, 32)$, $[32, 64)$, $[64, 128)$, $[128, 256)$, $[256,)$. If the value of the feature falls under one of these ranges, the corresponding bit is set to 1. Additionally, we create a magnitude variable for each of these ranges, which is the normalized value (magnitude) in that range. For example, if the average number of nonzeros of CSR format is 6, then the 3^{rd} bit of the variable will be set. Moreover, the magnitude variable of the $[4 - 8)$ range will have a value of 0.5.

Number of rows and number of nonzeros are normalized by considering minimum and maximums observed in the training and test sets.

5.4 ESTABLISHING CSR BASELINE

We compare the performance of our DDB technique against a CSR baseline on IBM's POWER10 processor. We establish the quality of that CSR baseline by comparing it against SpMM routines in the well-known production library MKL [127] on an Intel Xeon Platinum 8268 platform [128]. We use Intel Compiler v2020 for compilation and MKL v2020. The corresponding math library for POWER10 (ESSL) does not have SpMM routines. That is why we use this indirect approach to establish the quality of our baseline.

Table 5.3: MKL and CSR comparison on Intel Processors. The values are normalized to fastest execution time observed for each matrix. Lower is better.

columns:		***16***	***32***	***64***	***128***	***256***
DP	**csrA**	1.11	1.10	1.08	1.06	1.04
	csrC	1.05	1.05	1.06	1.08	1.09
	MKL	1.21	1.19	1.23	1.25	1.30
SP	**csrA**	1.17	1.13	1.11	1.08	1.08
	csrC	1.07	1.06	1.07	1.07	1.11
	MKL	1.24	1.21	1.18	1.24	1.27

We test our CSR implementation with different unrolling parameters – as described in Algorithms 5.1 and 5.2 – and various number of columns in the dense matrices. Details of datasets and unrolling parameters are found in Section 5.5. Both MKL and CSR are tested with 24 threads (a single socket). Each thread is bound to a physical core using *numactl*, and memory is allocated in the local node. For these experiments, we use 50 matrices including matrices used in previous work [129].

Table 5.3 reports, for each precision and method, the average of execution times normalized to fastest version for each matrix. (Best possible value is 1.) For both double- and single-precision SpMM, CSR-A and CSR-C implementations have at least one unrolling parameter set that is significantly faster than MKL, on average.

5.5 EXPERIMENTAL SETUP

System Setup. Our POWER10 system has a single socket with 15 SMT8 cores, equivalent to 30 SMT4 cores [117]. Each SMT4 core has 32 KB private L1 and 1MB private L2 caches. Also associated with each SMT4 core is a 4 MB local component of the L3 cache (LLC). POWER10 supports PowerISA 3.1, which includes Vector-Scalar Extensions (VSX) and Matrix-Multiply Assist (MMA) facilities. The system memory is 485 GB. We used IBM Advance Toolchain for Linux on Power Systems version 15.0.0-alpha (GCC 11.0.0), which provides compiler built-ins for MMA and VSX instructions. Details of our system are summarized in Table 5.4.

Table 5.4: POWER10 System Setup

System	Component	Properties
POWER10	***CPU***	30 SMT4 cores; 32 KB private L1, 1 MB private L2, 4 MB local L3 per core
	Memory	485 GB
	ISA	PowerISA 3.1, MMA, VSX
	Software	Advance Toolchain 15.0.0-alpha (GCC 11.0.0), OS Kernel: 4.18.0-277.el8.ppc64le

Input Matrices. We test 440 matrices from the Sparse Suite [82] matrix collection. We select the matrices used in previous work [129]. Additionally, we test matrices with 100 thousand to 10 million rows, to make sure that matrices are large enough to give consistent performance results and small enough to fit into memory when we have a large number of columns in the dense matrices.

Experiments. We always use the best performing version of CSR as our baseline implementation on POWER10 (CSR-A and CSR-C without slicing). For DDB-MM, we use both MMA-A and MMA-C versions. By using MMA-A and MMA-C for the blocked portion, and CSR-A and CSR-C for the compressed portion of DDB-HYB, we obtain four different implementations of DDB-HYB.

All our versions are implemented in C++ using OpenMP. We have manually vectorized all implementations by using compiler built-ins for POWER10. Similarly, for MMA-A and

MMA-C we have used the built-ins provided for MMA instructions and enable *-O3* optimizations.

We test both double-precision (64-bit) and single-precision (32-bit) floating-point arithmetic. We repeat each experiment for 110 iterations and ignore the first 10 iterations as warm-up and report the average execution time of the last 100 iterations.

Choosing SVM parameters and Testing SpMM-OPT. We use Support Vector Machines (SVM) with linear kernel for our individual machine learning models. A linear SVM kernel has regularization parameter (c) that needs to be tuned. In order to tune c parameter, we use grid search approach. We test c parameters in the range $[0, 1, 10, 100, 1000, 10000]$ by increasing c exponentially. To select the best c parameter, we have considered the average speedup observed with respect to baseline CSR implementation (best of CSR-A and CSR-C without slicing). In the end, we have chosen a single $c = 1$ parameter to use in all ML models. Speedups from using SVMs generated with $c = 1$ are always in the 10% neighborhood of the maximum average speedup observed for any values in the grid search. While testing SpMM-OPT, we use 10-fold approach and report the aggregate results. The 10-fold approach is commonly used for creating training and test sets from a single data set. It creates 10 disjoint training and test set pairs that includes 90% and 10% of all samples in training and test sets, respectively.

5.6 EXPERIMENTAL RESULTS

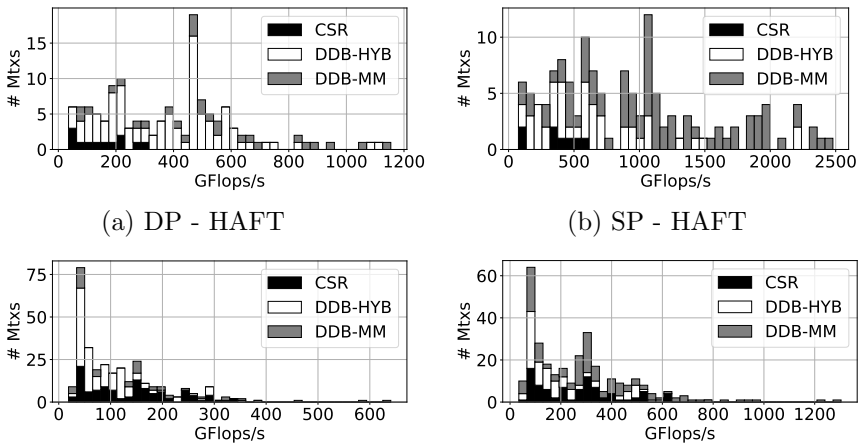
In this section, we analyze the effectiveness of various DDB techniques and the effectiveness of SpMM-OPT.

5.6.1 Performance of Blocking with MMA Facilities

First, we analyze the performance potential of DDB-MM and DDB-HYB. We test all $\langle$`method`, `slicing`$\rangle$ pairs against each sparse matrix and number of columns in the dense matrices. We measure the FLOPS/s (single- and double-precision) for each case.

Figure 5.6 shows the distribution of FLOPS/s for SpMM computations for double and single-precision, respectively. Each column is a performance bin and the y-axis is the number of matrices for which the best observed FLOPS/s falls on that bin. The columns also show the breakdown of methods achieving the highest floating-point throughput for each matrix.

We observed that the maximum FLOPS/s for our matrices are 1.15 TFLOPS/s for double-precision and 2.5 TFLOPS/s for single-precision computations. These maximums



(a) DP - HAFT (b) SP - HAFT

(c) DP - LAFT (d) SP - LAFT

Figure 5.6: Floating-point throughput distribution of HAFT and LAFT matrices.

are achieved by utilizing the MMA units with the DDB-MM method. The maximum performance achieved by the DDB-MM method (with double-precision) is approximately 40% of the 2.9 TFLOPS/s observed in the LINPACK benchmark on a POWER10 single-chip module.

Furthermore, we observed that MMA units are needed to achieve higher than 335 GFLOPS/s with double-precision values and higher than 631 GFLOPS/s with single precision values.

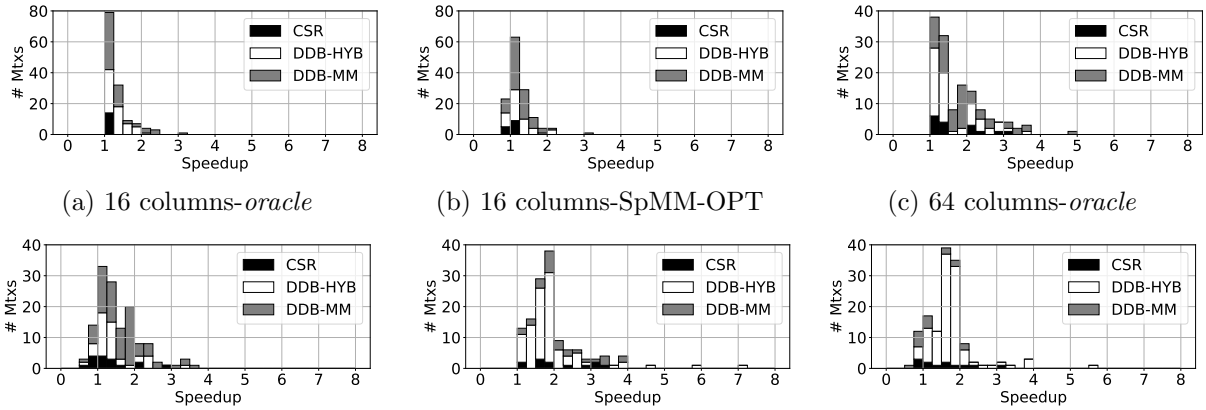
As expected, we observed the highest floating-point throughput for HAFT matrices from DDB methods, DDB-MM and DDB-HYB. For double-precision, we see that DDB-HYB is the fastest method for the majority of the matrices. For single-precision, we see that DDB-MM is more commonly the fastest, because single-precision SpMM has a higher FLOPS/byte ratio. MMA techniques are also useful for LAFT matrices. DDB-HYB falls back to CSR for such matrices, and even a slight improvement by the dense portion impacts on performance. DDB-HYB is the fastest method for 247 of the matrices with double-precision SpMM. DDB-MM is the fastest method for 211 matrices for single-precision SpMM.

Finally, we observe that slicing has a positive performance potential. For example, 147, 68, 19, and 19 of the matrices achieve the highest FLOPS/s with 16-, 32-, 64-, and 128-column dense matrices, respectively, for double-precision SpMM. For single-precision SpMM, we observed that 116, 106, 82, and 18 of 440 matrices obtain the highest throughput with 16-, 32-, 64-, and 128-column dense matrices, respectively. 256-column dense matrices achieve top throughput in the remaining cases.

5.6.2 Performance of SpMM-OPT

To analyze the effectiveness of SpMM-OPT, we compare its performance to an *oracle* method. The *oracle* method selects the best SpMM strategy by exhaustively searching all

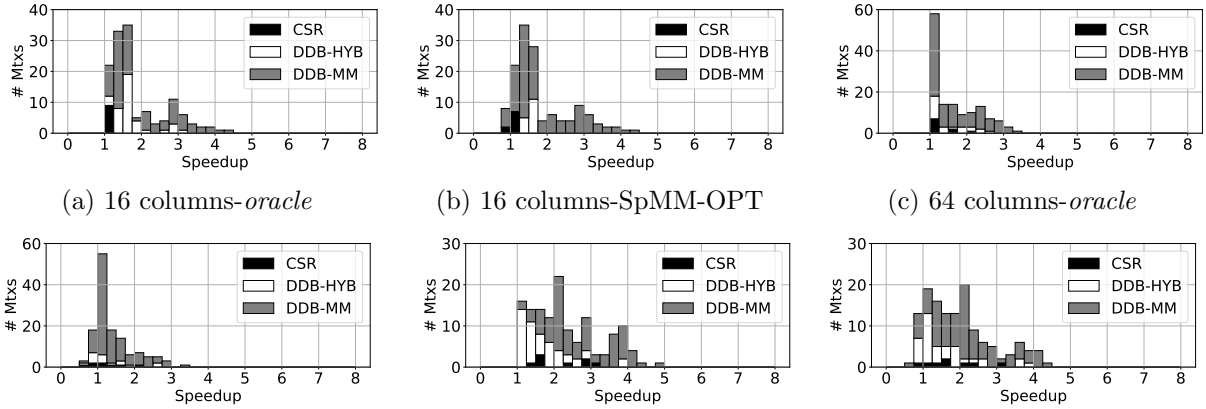
possibilities. Figures 5.7-5.10 show the distribution of the potential speedup for *oracle* and SpMM-OPT. The speedups are calculated by normalizing the floating-point throughput of the selected SpMM strategy by *oracle* or SpMM-OPT to the floating-point throughput of CSR implementation without slicing. In these figures, each bar represents a 0.25 range.



(a) 16 columns-*oracle* (b) 16 columns-SpMM-OPT (c) 64 columns-*oracle*

(d) 64 columns-SpMM-OPT (e) 256 columns-*oracle* (f) 256 columns-SpMM-OPT

Figure 5.7: Speedup for HAFT matrices (double-precision).



(a) 16 columns-*oracle* (b) 16 columns-SpMM-OPT (c) 64 columns-*oracle*

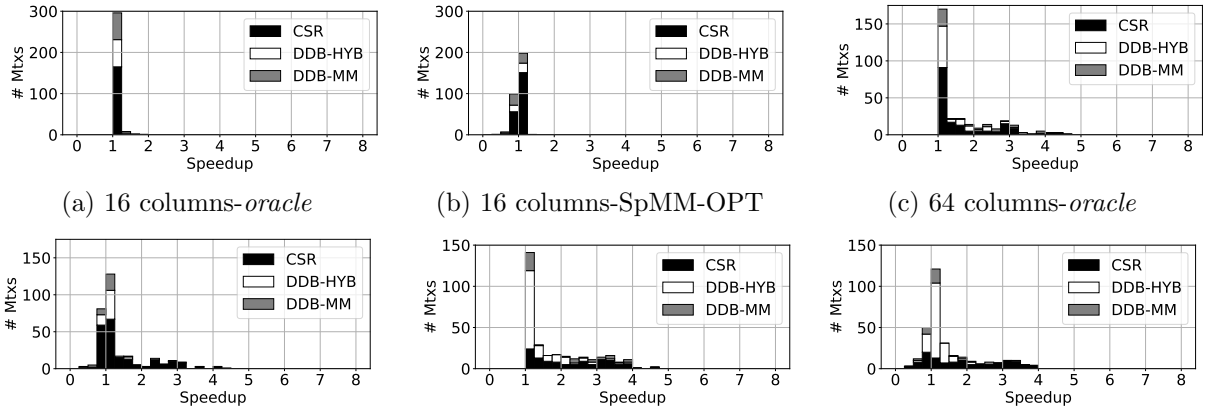
(d) 64 columns-SpMM-OPT (e) 256 columns-*oracle* (f) 256 columns-SpMM-OPT

Figure 5.8: Speedup for HAFT matrices (single-precision).

Prediction for HAFT matrices. Figure 5.7 shows the speedup distribution for HAFT matrices with double-precision SpMM. We observe that the oracle achieves an average speedup of 1.31, 1.32, 1.76, 1.93, and 2.02× for 16, 32, 64, 128, and 256 columns dense matrices, respectively. SpMM-OPT can achieve 1.25, 1.18, 1.55, 1.60, and 1.70× for 16, 32, 64, 128, and 256 columns dense matrices, respectively. For single-precision SpMM (Figure 5.8), the average speedups that we observe are higher. The average speedup of *oracle* for the same set of columns in the dense matrices are 1.85, 1.63, 1.66, 2.24, and 2.34× while SpMM-OPT can

achieve 1.79, 1.46, 1.41, 1.91, and 1.99×, respectively. As observed in the previous section, the reason behind this is the higher FLOP density per byte in single-precision cases.

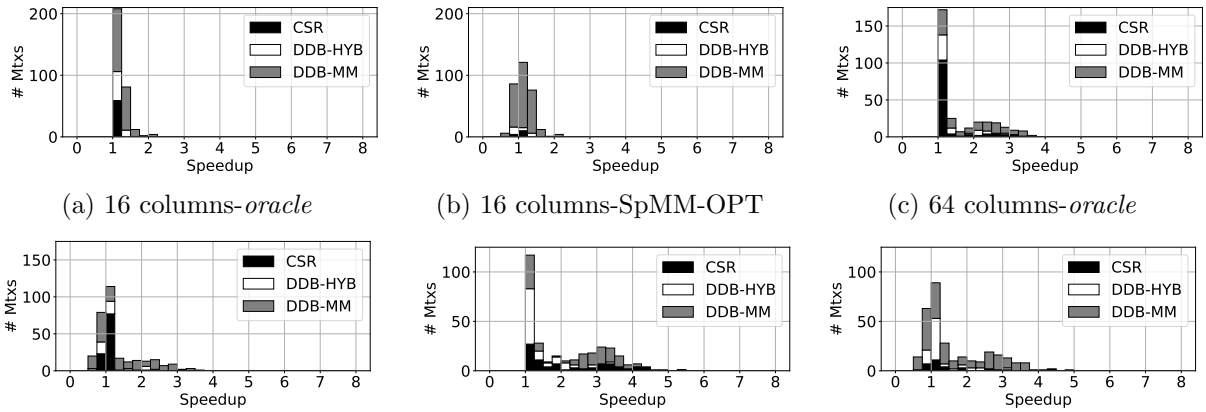
We observe that SpMM-OPT is successful at predicting the SpMM method and slicing factor. Although there are matrices in the speedup range $0.75 - 1.0$, generally, these are matrices with a speedup of $\geq 0.9\times$. Another reason behind the differences between *oracle* and SpMM-OPT is that SpMM-OPT cannot identify all extremes, such as very high speedup values found by the *oracle* method. With HAFT matrices, we observe that DDB-HYB and DDB-MM methods are frequently selected. For example, one of DDB-HYB and DDB-MM is selected for more than 80% of the matrices.



(a) 16 columns-*oracle* (b) 16 columns-SpMM-OPT (c) 64 columns-*oracle*

(d) 64 columns-SpMM-OPT (e) 256 columns-*oracle* (f) 256 columns-SpMM-OPT

Figure 5.9: Speedup for LAFT matrices (double-precision).



(a) 16 columns-*oracle* (b) 16 columns-SpMM-OPT (c) 64 columns-*oracle*

(d) 64 columns-SpMM-OPT (e) 256 columns-*oracle* (f) 256 columns-SpMM-OPT

Figure 5.10: Speedup for LAFT matrices (single-precision).

Prediction for LAFT matrices. Figures 5.9 and 5.10 show the speedup distributions for the LAFT matrices for double and single-precision SpMM, respectively. For LAFT matrices, we expected to see that CSR would be the best method for most matrices However,

we observe that DDB-HYB and DDB-MM have been selected for a significant number of matrices for LAFT matrices as well.

For LAFT matrices, we observe that many matrices fall under $0.75 - 1.00\times$ speedup range. Similar to HAFT matrices, these generally show $\geq 0.9\times$ speedup. Thus, the average performance is not significantly affected. In the worst case, 16 columns dense matrices where slicing is not applicable, we see an average speedup of $0.98\times$ for double-precision SpMM with a harmonic mean of $0.97\times$. DDB-HYB generally behaves as the CSR baseline and do not lose much performance since we do not have many dense blocks. Similarly, for the single-precision case, the average speedup is $1.14\times$ with a harmonic mean of $1.10\times$ for 16 columns dense matrices. For single-precision matrices, we also observe that DDB-MM gives us a performance boost in many cases.

Effect of Slicing. For both HAFT and LAFT matrices, cache slicing improves the performance significantly. Tables 5.5 and 5.6 show the percentage of matrices using different slicing parameters for a given number of columns in the dense matrix. The rows of the table are for the number of columns in the dense matrix, while columns show the selected slicing parameter.

In HAFT matrices, for double-precision SpMM, up to 85% of the matrices may benefit from slicing for a given number of columns in the dense matrices. For example, 95.2%, 64.4%, and 53.3% of matrices benefit from slicing for 64, 128, and 256 columns, respectively. Similarly, slicing is also improving the performance in single-precision SpMM. For example, for 128 columns dense matrices, we observe that more than 90% of HAFT matrices benefit from slicing.

Table 5.5: Slicing parameters selected by *oracle* and SpMM-OPT for HAFT matrices. Rows show the number of columns in the dense matrices, while the columns of the Table show the slicing parameter selected. All values are in percentages.

	oracle					SpMM-OPT				
DP	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
16	100	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0
32	29.6	70.4	0.0	0.0	0.0	23.0	77.7	0.0	0.0	0.0
64	29.6	55.6	14.8	0.0	0.0	31.1	64.4	4.4	0.0	0.0
128	27.4	34.1	3.0	35.6	0.0	19.3	29.6	0.7	50.4	0.0
256	24.4	22.2	2.2	4.4	46.7	15.6	21.5	0.0	3.0	60.0
SP	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
16	100	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0
32	24.4	75.6	0.0	0.0	0.0	8.1	91.9	0.0	0.0	0.0
64	21.5	34.8	43.7	0.0	0.0	14.1	51.9	34.1	0.0	0.0
128	21.5	32.6	36.3	9.6	0.0	17.0	48.1	29.6	5.2	0.0
256	20.7	28.9	27.4	2.2	20.7	17.8	43.0	25.9	0.0	13.3

Table 5.6: Slicing parameters selected by *oracle* and SpMM-OPT for LAFT matrices. Rows show the number of columns in the dense matrices, while the columns of the Table show the slicing parameter selected. All values are in percentages.

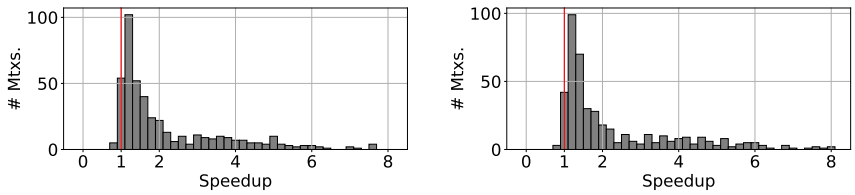
	oracle					SpMM-OPT				
DP	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
16	100	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0
32	45.6	54.4	0.0	0.0	0.0	37.6	62.4	0.0	0.0	0.0
64	44.0	17.9	38.1	0.0	0.0	51.3	8.8	39.9	0.0	0.0
128	38.4	13.7	7.2	40.7	0.0	51.0	7.2	1.6	40.2	0.0
256	36.8	12.4	5.2	4.2	41.4	43.5	5.9	1.6	2.9	46.1
SP	**16**	**32**	**64**	**128**	**256**	**16**	**32**	**64**	**128**	**256**
16	100	0.0	0.0	0.0	0.0	100	0.0	0.0	0.0	0.0
32	32.2	67.8	0.0	0.0	0.0	17.0	83.0	0.0	0.0	0.0
64	30.3	25.4	44.3	0.0	0.0	32.4	24.8	42.8	0.0	0.0
128	29.0	23.1	16.6	31.3	0.0	34.3	20.9	10.1	34.6	0.0
256	28.7	21.5	14.7	4.9	30.3	40.8	23.5	8.2	1.0	26.5

DDB-MM vs. Rectangular Blocks. We compare the DDB-MM approach to using rectangular 4×4 blocks. Figure 5.11 shows the distribution of the speedups obtained by using DDB-MM for 16 and 64 columns dense matrices with respect to using rectangular blocks with double-precision values. We observe that DDB-MM can achieve an average of 2.21× and 2.35× speedup with MMA-A and MMA-C methods with 16 columns dense matrices. The average speedups with 64 columns dense matrices are 1.45× and 1.48× for MMA-A and MMA-C methods, respectively. Figure 11 histograms cover all 440 matrices in our suite. HAFT matrices perform better with 4x4 blocks compared to LAFT matrices. For a few matrices that naturally have 4×4 blocks, DDB-MM and rectangular blocks are competitive. However, on average, DDB-MM still outperforms 4×4 by 1.49× and 1.08× for 16 and 64 columns, respectively, for HAFT matrices.

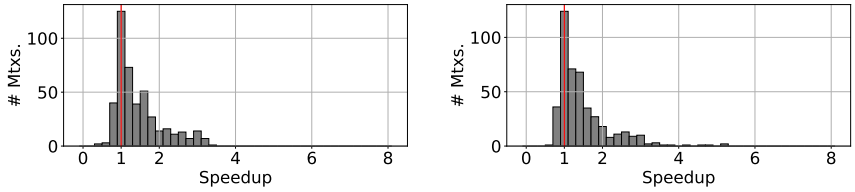
The main reason behind the success of DDB-MM is the superfluous FLOPs introduced by using rectangular blocks. Figure 5.12 shows the ratio of effective FLOPs for these HAFT matrices when DDB-MM and 4×4 rectangular blocks are used.

Using DDB-MM compared to 4×4 blocks significantly increases the effective FLOPs ratio. With 4×4 blocks, almost half of the matrices have less than 50% effective FLOPs ratio –50% or more of the FLOPs executed don't contribute to the final result– where matrix units such as MMA operate inefficiently.

On the other hand, with DDB-MM, only 30% of matrices would have less than 50% effective FLOPs ratio. Moreover, only 10% of all matrices have an effective FLOPs ratio of 40% or less. Therefore, such matrices with low effective FLOPs ratio would be executed by DDB-HYB or CSR efficiently, and SpMM-OPT can select a suitable method to use.

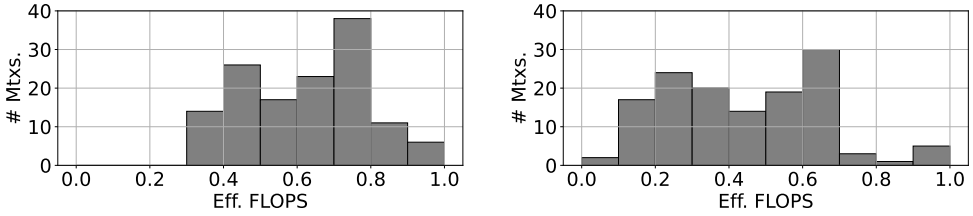


(a) 16 columns - MMA-A (b) 16 columns - MMA-C



(c) 64 columns - MMA-A (d) 64 columns - MMA-C

Figure 5.11: Speedup of DDB-MM with respect to using rectangular 4×4 blocks with MMA-A and MMA-C techniques.



(a) Effective FLOPS with DDB-MM (b) Effective FLOPS with 4×4 Blocks

Figure 5.12: Effective FLOPs ratio for 4×4 rectangular blocks and DDB-MM for HAFT matrices.

5.7 RELATED WORK

SpMM is optimized for both CPU and GPU systems. In the CPU domain, the main concern is to improve locality of memory references. For example, Compressed Sparse Blocks [6, 130] improves performance by tiling for caches, while ApST [93] augments CSR representation to improve locality. Additionally, [129] proposes a formulation to find the best tiling parameters for registers and cache level blocking. SpMM optimizations for GPUs generally target load balancing issues and efficient use of cache capacity in GPUs [131, 132]. These techniques include methods to manage warp level parallelism and to improve memory coalescing. Moreover, due to SpMM operations observed in Graph Neural Network training and inference, several other approaches were proposed for GPUs [12, 133], including improving the performance via slicing and locality aware scheduling.

Blocking for matrix-multiply units for SpMM and for multiplications of two sparse matrices were explored in the GPU domain with the BCSR approach for Tensor Cores [122, 123]. Our work is differentiated by its dynamic structure and the fine-grained capabilities of MMA

units. Blocking techniques were heavily explored for SpMV. Examples include [94, 124, 125, 126, 134, 135]. While BCSR was limited with its static block sizes, Unaligned BCSR [126] and VBR [134] improved performance by introducing dynamically sized blocks. However, our main focus is to exploit throughput discrepancy rather than optimize padding or improve irregular access patterns. Instead of finding rectangular blocks, we find column blocks that are dynamically grouped to create a block for MMA units. Previous vectorization techniques for SpMV, such as ELLPACK and Sell-c-σ also create a compact representation by grouping consecutive rows of the matrix [64, 101, 136, 137]. However, these techniques do not identify the common columns across the consecutive rows and only focus on minimizing padding. For example, the dense column blocks identified by Sell-c-σ would have multiple column ids appearing in the same dense-column block.

Auto-tuning approaches were previously evaluated [63, 94, 102]. Recently, [12] and [129] proposed to use slicing to improve SpMM performance. They use a model based mechanism to tune the slicing parameter. Our work, on the other hand, also targets the effective utilization of functional units. Previous work also utilizes machine learning for performance prediction for sparse matrix computations [106, 107, 108, 109]. They generally target selecting a preferable matrix format for executing SpMV more efficiently. SpMM-OPT differs from previous work in multiple aspects. First, instead of only choosing a format, SpMM-OPT makes a decision in a comprehensive optimization framework. Additionally, SpMM-OPT chooses optimizations and also predicts the best slicing parameters to use for a given method. Other examples also exist for leveraging machine learning to predict performance for other primitives, such as sparse matrix sparse matrix multiplication [113].

Although we don't explore reordering the matrix in this work, several studies may apply to our work to improve its performance [68, 69, 70, 95, 135, 137, 138, 139]. DDB uses consecutive rows while relaxing the order of columns. As a result, if we can find similar rows cheaply and group them together, we can futher improve performance.

5.8 CONCLUSIONS

In this work, we introduced Dense Dynamic Blocks (DDB), which coordinates the execution of SpMM between matrix-multiply units and vector units to get the maximum performance. DDB can enable a matrix unit-oriented strategy via DDB-MM or a hybrid strategy with DDB-HYB. This approach enables DDB to achieve the highest floating-point throughput for SpMM. However, we observed that not all sparse matrices are equal in terms of DDB's performance and utilizing matrix units effectively. Moreover, the cache behavior of SpMM with a given sparse matrix also affects the performance significantly. As a re-

sult, we have designed a machine learning method that can navigate this complex search space: SpMM-OPT. SpMM-OPT is a prediction mechanism for identifying the best SpMM strategy for a given sparse matrix and dense matrix pair. SpMM-OPT selects among vector unit oriented, matrix unit oriented, and the hybrid strategies for the highest floating-point throughput while also taking the cache optimizations into account.

We test DDB and SpMM-OPT with matrices from the well-known SuiteSparse matrix collection on a POWER10 system with vector and matrix-multiply units. We observe that DDB-MM and DDB-HYB can achieve up to 1.15 and 2.5 TFLOPS/s floating-point throughput for double- and single-precision SpMM, respectively. SpMM-OPT is effective in choosing high-performance SpMM methods, achieving up to 2× average speedup compared to a CSR baseline.

CHAPTER 6: CONCLUSIONS

In this thesis, we have proposed novel solutions for obtaining optimized graph analytics applications and sparse matrix primitives. We discussed solutions for priority scheduling for graph analytics, SpMV style graph analytics applications, an ML-based approach to optimize SpMV kernels, and, finally, a method to accelerate SpMM and an optimizer.

In the first part, we proposed PMOD. PMOD is a novel CPS based on the obim approach that dynamically adapts to the ranges of priorities exhibited by the application and input graph characteristics. It addressed the significant slowdowns that we observed under certain priority distributions. PMOD was able to achieve the same performance as obim without any manual and application specific tuning.

In the second part, we addressed the challenges in SpMV style computations with power-law graphs. We observed that current vectorization and locality optimizations are not effective for such skewed graphs. We designed LAV, a new SpMV approach that leverages the input's power-law structure to extract locality and enable effective vectorization. LAV splits the sparse matrix into a vectorization-friendly and locality aware dense portion and a compressed sparse portion by leveraging the power-law structure of the input matrix. We observed that LAV, on several real-world and synthetic graphs on a modern aggressive out of order processor, is faster than CSR (and prior approaches) by an average of 1.5× with only 3% memory overhead.

LAV is able to leverage power-law structure of the input matrix by using a set of preprocesing techniques such as CFS, RFS, and segmenting. These preprocessing techniques and combinations thereof encapsulate many different SpMV methods. These different combinations can yield the best performance for different classes of sparse matrices. We designed MVPP to take advantage of these SpMV methods. MVPP is an ML-based method that can integrate all of the optimizations into a single framework and choose a fast SpMV strategy for a wide range of sparse matrices. MVPP relies on a set of novel matrix features to identify sparse matrices' size, skew, and locality characteristics instead of format specific features. Its modular design makes it easily extandable. At the end, MVPP can achieve an average of 2.4× speedup with respect to the MKL baseline and 1.13× with respect to MKL inspector-executor baseline with 50% less overhead.

Finally, we focused on a more computationally intensive kernel: SpMM. We introduced Dense Dynamic Blocks (DDB), a method which coordinates the execution of SpMM between matrix-multiply units and vector units to maximize floating-point throughput. Similar to SpMV, we realized that a single SpMM method is not enough to get the best performance

for a wide range of sparse matrices. Moreover, the cache behavior of SpMM with a given sparse matrix also affects the performance significantly. We designed SpMM-OPT, which selects among vector unit oriented, matrix unit oriented, and the hybrid strategies for the highest floating-point throughput while also taking advantage of cache optimizations.

Our techniques can easily be integrated into existing graph processing and sparse basic linear algebra frameworks. For instance, PMOD is already developed within a state-of-the-art Galois graph processing framework. On the other hand, MVPP and SpMM-OPT are user-transparent. They do not require user input. Therefore, they can become part of sparse BLAS frameworks such as MKL and GraphBLAS systems seamlessly.

REFERENCES

[1] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2013, pp. 66–76.

[2] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 488–499.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1016/S0169-7552(98)00110-X

[4] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324140

[5] H. Anzt, S. Tomov, and J. J. Dongarra, "Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product." in *SpringSim (HPS)*, 2015, pp. 75–82.

[6] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 1213–1222.

[7] H. M. Aktulga, M. Afibuzzaman, S. Williams, A. Buluç, M. Shao, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "A high performance block eigensolver for nuclear configuration interaction calculations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1550–1563, 2017.

[8] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. USA: Society for Industrial and Applied Mathematics, 2003.

[9] C. T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics (SIAM), 1987.

[10] C. Yang, Y. Wang, and J. D. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the gpu," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 841–847.

[11] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019. [Online]. Available: http://arxiv.org/abs/1909.01315

[12] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang, "Featgraph: A flexible and efficient backend for graph neural network systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[13] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 438–450.

[14] N. Amenta, S. Choi, and G. Rote, "Incremental constructions con brio," in *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, ser. SCG '03. New York, NY, USA: ACM, 2003. [Online]. Available: http://doi.acm.org/10.1145/777792.777824 pp. 211–219.

[15] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun, "Internally deterministic parallel algorithms can be fast," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145840 pp. 181–192.

[16] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP'13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2517349.2522739 pp. 456–471.

[17] 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*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2442516.2442530 p. 135–146.

[18] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ser. ICDM '09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: https://doi.org/10.1109/ICDM.2009.14 pp. 229–238.

[19] A. Buluc and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516

[20] D. Buono, J. A. Gunnels, X. Que, F. Checconi, F. Petrini, T. Tuan, and C. Long, "Optimizing sparse linear algebra for large-scale graph analytics," *Computer*, vol. 48, no. 8, pp. 26–34, Aug 2015.

[21] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[22] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," 2016.

[23] "NVIDIA Tesla V100 GPU Architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017, [Online; accessed 02-April-2021].

[24] "NVIDIA A100 Tensor Core GPU Architecture," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020, [Online; accessed 02-April-2021].

[25] J. E. Moreira, K. Barton, S. Battle, P. Bergner, R. Bertran, P. Bhat, P. Caldeira, D. Edelsohn, G. Fossum, B. Frey, N. Ivanovic, C. Kerchner, V. Lim, S. Kapoor, T. M. Filho, S. M. Mueller, B. Olsson, S. Sadasivam, B. Saleil, B. Schmidt, R. Srinivasaraghavan, S. Srivatsan, B. Thompto, A. Wagner, and N. Wu, "A matrix math facility for power isa(tm) processors," 2021.

[26] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688507 pp. 183–193.

[27] L. Dhulipala, G. Blelloch, and J. Shun, "Julienne: A framework for parallel graph algorithms using work-efficient bucketing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3087556.3087580 pp. 293–304.

[28] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos, "Graphgrind: Addressing load imbalance of graph partitioning," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079079.3079097 pp. 16:1–16:10.

[29] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "Graphlab: A new framework for parallel machine learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, ser. UAI'10. Arlington, Virginia, United States: AUAI Press, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=3023549.3023589 pp. 340–349.

[30] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali, "Dsmr: A parallel algorithm for single-source shortest path problem," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926287 pp. 32:1–32:14.

[31] B. V. Cherkassy and A. V. Goldberg, "On implementing push-relabel method for the maximum flow problem," in *Proceedings of the 4th International IPCO Conference on Integer Programming and Combinatorial Optimization*. London, UK, UK: Springer-Verlag, 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=645586.659457 pp. 157–171.

[32] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 209–221.

[33] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688523 pp. 11–20.

[34] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2755573.2755616 pp. 80–82.

[35] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688547 pp. 277–278.

[36] M. Wimmer, F. Versaci, J. L. Träff, D. Cederman, and P. Tsigas, "Data structures for task-based priority scheduling," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555278 pp. 379–380.

[37] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959. [Online]. Available: http://dx.doi.org/10.1007/BF01386390

[38] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms," *SIGPLAN Not.*, vol. 46, no. 8, pp. 3–12, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/2038037.1941557

[39] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 228–241.

[40] J. Lindén and B. Jonsson, "A skiplist-based concurrent priority queue with minimal memory contention," in *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304*, ser. OPODIS 2013. Berlin, Heidelberg: Springer-Verlag, 2013. [Online]. Available: https://doi.org/10.1007/978-3-319-03850-6_15 pp. 206–220.

[41] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 609–627, May 2005. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2004.12.005

[42] N. Shavit and I. Lotan, "Skiplist-based concurrent priority queues," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, May 2000, pp. 263–268.

[43] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, Computer Laboratory, February 2004.

[44] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, June 1990. [Online]. Available: http://doi.acm.org/10.1145/78973.78977

[45] D. Alistarh, J. Kopinsky, J. Li, and G. Nadiradze, "The power of choice in priority scheduling," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3087801.3087810 pp. 283–292.

[46] D. Nguyen and K. Pingali, "Synthesizing concurrent schedulers for irregular algorithms," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950404 pp. 333–344.

[47] DIMACS, "9th DIMACS implementation challenge," 2006. [Online]. Available: http://www.dis.uniroma1.it/challenge9/download.shtml

[48] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772751 pp. 591–600.

[49] M. W. Mahoney, A. Dasgupta, K. J. Lang, and J. Leskovec, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[50] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1150402.1150412 pp. 44–54.

[51] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[52] U. Meyer and P. Sanders, "Δ-stepping: A parallelizable shortest path algorithm," *J. Algorithms*, vol. 49, no. 1, pp. 114–152, Oct. 2003. [Online]. Available: http://dx.doi.org/10.1016/S0196-6774(03)00076-2

[53] I. Calciu, H. Mendes, and M. Herlihy, "The adaptive priority queue with elimination and combining," in *Distributed Computing*, F. Kuhn, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 406–420.

[54] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1810479.1810540 pp. 355–364.

[55] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit, "Using elimination to implement scalable and lock-free fifo queues," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1073970.1074013 pp. 253–262.

[56] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389013 pp. 12:1–12:10.

[57] S. Kang and D. A. Bader, "An efficient transactional memory algorithm for computing minimum spanning forest of sparse graphs," *SIGPLAN Not.*, vol. 44, no. 4, pp. 15–24, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504182

[58] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, July 2015. [Online]. Available: https://doi.org/10.14778/2809974.2809983

[59] 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*, ser. ICS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751209 pp. 339–350.

[60] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: Efficient Vectorization of SpMV on x86 Processors," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3168818 pp. 149–162.

[61] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2925426.2926278 pp. 37:1–37:12.

[62] S. Beamer, K. Asanovic, and D. Patterson, "Reducing Pagerank Communication via Propagation Blocking," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 820–831.

[63] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004. [Online]. Available: https://doi.org/10.1177/1094342004041296

[64] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for modern processors with wide SIMD units," *CoRR*, vol. abs/1307.6209, 2013. [Online]. Available: http://arxiv.org/abs/1307.6209

[65] W. T. Tang, R. Zhao, M. Lu, Y. Liang, H. P. Huyng, X. Li, and R. S. M. Goh, "Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2015, pp. 136–145.

[66] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465013 pp. 273–282.

[67] L. Chen, P. Jiang, and G. Agrawal, "Exploiting Recent SIMD Architectural Advances for Irregular Applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854046 pp. 47–58.

[68] 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*, ser. SIGMOD '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915220 pp. 1813–1828.

[69] 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*, ser. WWW '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1963405.1963488 pp. 587–596.

[70] 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)*, May 2016, pp. 22–31.

[71] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3319535.3354252 p. 753–768.

[72] P. Boldi and S. Vigna, "The Webgraph Framework I: Compression techniques," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/988672.988752 pp. 595–602.

[73] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph structure in the web — revisited: A trick of the heavy tail," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14 Companion. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2567948.2576928 pp. 427–432.

[74] A. Addisie, H. Kassa, O. Matthews, and V. Bertacco, "Heterogeneous memory subsystem for natural graph analytics," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2018. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IISWC.2018.8573480 pp. 134–145.

[75] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 203–214, 2018.

[76] H. Wang, W. Liu, K. Hou, and W.-c. Feng, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS'16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926291

[77] "Performance monitoring and benchmarking suite," https://github.com/RRZE-HPC/likwid/, accessed: 2019-04-30.

[78] "Intel Math Kernel Library Inspector-executor Sparse BLAS Routines," https://software.intel.com/en-us/articles/intel-math-kernel-library-inspector-executor-sparse-blas-routines, Mar. 2015. [Online]. Available: https://software.intel.com/en-us/articles/intel-math-kernel-library-inspector-executor-sparse-blas-routines

[79] W. Liu and B. Vinter, "CSR5-based SpMV on CPUs, GPUs and Xeon Phi," https://github.com/bhSPARSE/Benchmark_SpMV_using_CSR5, accessed: 2020-01-30.

[80] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Parallelized and vectorized SpMV on Intel Xeon Phi (Knights Landing, AVX512, KNL)," https://github.com/puckbee/CVR, accessed: 2020-01-30.

[81] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[82] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[83] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna, "Web data commons - hyperlink graphs," http://webdatacommons.org/hyperlinkgraph/, accessed: 2020-01-30.

[84] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proc. the 19th Intl. Conf. on World Wide Web*. New York, NY, USA: ACM, 2010, pp. 591–600.

[85] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[86] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[87] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[88] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia, "Making caches work for graph analytics," in *2017 IEEE International Conference on Big Data (Big Data)*, Dec 2017, pp. 293–302.

[89] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing Indirect Memory References with Milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967948 pp. 299–312.

[90] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, "Non-affine extensions to polyhedral code generation," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO'14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2544137.2544141 pp. 185–194.

[91] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky, "An approach for code generation in the sparse polyhedral framework," *Parallel Comput.*, vol. 53, no. C, pp. 32–57, Apr. 2016. [Online]. Available: https://doi.org/10.1016/j.parco.2016.02.004

[92] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, July 1999.

[93] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3293883.3295712 pp. 300–314.

[94] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, Nov 2002, pp. 26–26.

[95] R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A high performance algorithm using pre-processing for the sparse matrix-vector multiplication," in *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992, pp. 32–41.

[96] M. Besta, F. Marending, E. Solomonik, and T. Hoefler, "SlimSell: A vectorizable graph representation for breadth-first search," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 32–41.

[97] S. Grossman, H. Litz, and C. Kozyrakis, "Making pull-based graph processing performant," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3178487.3178506 p. 246–260.

[98] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 643–652.

[99] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *High Performance Embedded Architectures and Compilers*, Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–125.

[100] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, p. C401–C423, Jan 2014. [Online]. Available: http://dx.doi.org/10.1137/130930352

[101] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Speeding Up SpMV for Power-Law Graph Analytics by Enhancing Locality & Vectorization," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00090 pp. 1–15.

[102] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone, "Performance optimization and modeling of blocked sparse kernels," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 4, p. 467–484, Nov. 2007. [Online]. Available: https://doi.org/10.1177/1094342007083801

[103] J. Kunegis and J. Preusse, "Fairness on the web: Alternatives to the power law," in *Proceedings of the 4th Annual ACM Web Science Conference*, ser. WebSci '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2380718.2380741 p. 175–184.

[104] D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A Recursive Model for Graph Mining*. Society for Industrial and Applied Mathematics (SIAM), 2004, pp. 442–446. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43

[105] D. Funke, S. Lamm, U. Meyer, P. Sanders, M. Penschuck, C. Schulz, D. Strash, and M. von Looz, "Communication-free massively distributed graph generation," 2019.

[106] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2491956.2462181 p. 117–126.

[107] B. Yilmaz, B. Aktemur, M. J. Garzarán, S. Kamin, and F. Kiraç, "Autotuning runtime specialization for sparse matrix-vector multiplication," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, Mar. 2016. [Online]. Available: https://doi.org/10.1145/2851500

[108] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2751205.2751244 p. 99–108.

[109] W. Abu-Sufah and A. Abdel Karim, "Auto-tuning of sparse matrix-vector multiplication on graphics processors," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 151–164.

[110] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for spmv-based applications," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 950–959.

[111] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3178487.3178495 p. 94–108.

[112] W. Zhou, Y. Zhao, X. Shen, and W. Chen, "Enabling runtime spmv format selection through an overhead conscious method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 80–93, 2020.

[113] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3330345.3330354 p. 94–105.

[114] 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*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3293883.3295716 p. 201–213.

[115] 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*, ser. PPoPP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3293883.3295733 p. 38–52.

[116] P. Bhat, J. Moreira, and S. K. Sadasivam, "Matrix-Multiply Assist (MMA) Best Practices Guide," https://www.redbooks.ibm.com/redpieces/pdfs/redp5612.pdf, 2021, [Online; accessed 02-April-2021].

[117] W. J. Starke, B. W. Thompto, J. A. Stuecheli, and J. E. Moreira, "Ibm's power10 processor," *IEEE Micro*, vol. 41, no. 2, pp. 7–14, 2021.

[118] "Intel Architecture Instruction Set Extensions Programming Reference ," https://software.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html, 2021, [Online; accessed 02-April-2021].

[119] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. S. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40, 2017.

[120] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 78–91, 2018.

[121] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "Sparsetrain: Leveraging dynamic sparsity in software for training dnns on general-purpose simd processors," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3410463.3414655 p. 279–292.

[122] T. Yamaguchi and F. Busato, "Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores," https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/, 2021, [Online; accessed 02-April-2021].

[123] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix–matrix multiplication with gpu tensor cores," *Computers & Electrical Engineering*, vol. 88, p. 106848, Dec 2020. [Online]. Available: http://dx.doi.org/10.1016/j.compeleceng.2020.106848

[124] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *2009 International Conference on Computational Science and Engineering*, vol. 1, 2009, pp. 247–256.

[125] V. Karakasis, G. Goumas, and N. Koziris, "Perfomance models for blocked sparse matrix-vector multiplication kernels," in *2009 International Conference on Parallel Processing*, 2009, pp. 356–364.

[126] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *Proceedings of the First International Conference on High Performance Computing and Communications*, ser. HPCC'05. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: https://doi.org/10.1007/11557654_91 p. 807–816.

[127] "Intel oneAPI Math Kernel Library," Online, 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/comp-onents/onemkl

[128] "Intel Xeon Platinum 8268 Processor," https://ark.intel.com/content/www/us/en/ark/products/192481/intel-xeon-platinum-8268-processor-35-75m-cache-2-90-ghz.html, [Online; accessed 02-April-2021].

[129] S. E. Kurt, A. Sukumaran-Rajam, F. Rastello, and P. Sadayyapan, "Efficient tiled sparse matrix multiplication through matrix signatures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[130] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1583991.1584053 p. 233–244.

[131] C. Yang, A. Buluç, and J. D. Owens, "Design principles for sparse matrix multiplication on the gpu," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 672–687.

[132] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[133] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen, "Understanding and bridging the gaps in current gnn performance optimizations," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437801.3441585 p. 119–132.

[134] Y. Saad, "SPARSEKIT: A Basic Toolkint for Sparse Matrix Computations," https://www-users.cs.umn.edu/~saad/PDF/RIACS-90-20.pdf, 2021, [Online; accessed 02-April-2021].

[135] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. IEEE, 1999, pp. 30–30.

[136] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on gpus," *ACM sigplan notices*, vol. 45, no. 5, pp. 115–126, 2010.

[137] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 66–79.

[138] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus," in *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, R. Gupta and X. Shen, Eds. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3332466.3374546 pp. 376–388.

[139] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.