UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

BRIDGING THE GAP BETWEEN SPARSE MATRIX COMPUTATION AND GRAPH MODELS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR of PHILOSOPHY

By

Khaled Abdelaal

Norman, Oklahoma

2024

BRIDGING THE GAP BETWEEN SPARSE MATRIX COMPUTATION AND GRAPH MODELS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Richard Veras, Chair

Dr. John Antonio

Dr. Qi Cheng

Dr. Javier Jo

Acknowledgements

I extend my sincere thanks to my family and friends who provided me with unwavering support and encouragement throughout my years of study and through the process of researching and writing this dissertation. This accomplishment would not have been possible without them. Thank you.

I would like to express my deepest appreciation to all those who provided me the possibility to complete this dissertation. Special gratitude goes to my advisor, Dr. Richard Veras, for his constant support and guidance throughout my Ph.D. journey. Additionally, I am also grateful to my dissertation committee for their invaluable support.

Table of Contents

List of Tables

List of Figures

Abstract

Sparsity manifests itself in a multitude of modern High-Performance Computing applications including graph neural networks, network analytics, and scientific computing. In sparse matrices, the majority of values are zeros. Traditional methods of storing and processing dense data are unsuitable for the new nature of sparse data, as they end up wasting storage and compute on zeros. Hence, a variety of sparse data formats that store only the non-zero elements were proposed in literature to provide a compact representation of sparse data.

Performance of operations on sparse data mainly depends on the sparse data format used for storing the data, as the algorithm needs to closely match the sparse data format. However, choosing the optimal sparse data format for the input sparse matrix is non-trivial, as the optimal format depends on the sparsity pattern of the input sparse matrix. For example, in sparse matrix-vector multiplication (SpMV), for the same input sparse matrix, using different sparse data formats can yield highly variant performance. The best format being the one that closely matches how the non-zeros are arranged within the matrix. Additionally, performance prediction for operations involving sparse matrices is not as straightforward as it used to be for the dense case. For dense computations, dimensions and strides suffice for performance predictions as they provide a sense of the number of floating point operations (FLOPs) to be performed, and how this number compares to the architecture properties (peak FLOPs, number of processing elements, etc.). On the other hand, sparse matrix dimensions do not directly convey useful information about the total number of operations to be performed, since the majority of elements are zeros and do not contribute to the total number of FLOPs. Moreover, existing work on sparse operations optimizations mainly depends on a discrete set of matrices, limiting the ability to generalize observations.

To address these challenges, we identify the sparsity pattern as the main driving factor for

performance. First, we propose an extensible classifier framework to automatically identify the sparsity pattern of the input sparse matrix. This framework uses graph neural networks (GNNs) by representing the input sparse matrix as a graph, and then learning the structural relationship between nodes in the matrix (graph). Our framework achieves up to 98% classification accuracy on full graphs, the same accuracy for scrambled matrices, and 92% accuracy for small random subsamples taken out of original graphs. Second, we use graph models as a proxy to generate large-scale synthetic sparse matrices. We propose another modular framework to study the correlation between the graph model parameters, and the structure of the resulting graph and its tolerance to noise during the generation step. Third, we also use graph models for a performance evaluation framework that can assist in finding the best sparse format for a given graph model on a given architecture, utilizing the graph model parameters as a representative set of features to predict performance, and providing a more robust way of visualizing sparse matrix operations performance. This framework also takes into consideration noise in the matrix generation step, and evaluates the extent of noise to which performance can still be predictable based on the graph model parameters.

Our results show that sparse computations need richer models to categorize sparse matrices in terms of structure, study the sensitivity of such models even for one structure, and tie performance to a more descriptive set of parameters.

Chapter 1

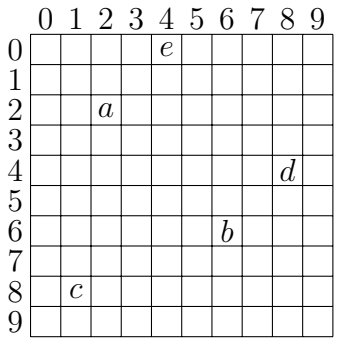
Introduction

Handling sparse data is challenge that poses itself in modern high-performance computing (HPC) workloads. Sparsity either naturally exists or enforced to reduce memory and compute requirements. Deep learning workloads are one example of such applications: input data (for training/inference) can be sparse due to the nature of data (social network relationship [62] , user-product interaction data [99], etc.), or the model inherent sparsity [80], or enforcing sparsity through pruning [39], and quantization [22], [100], [95], [31]. Either ways, efficiently exploiting sparsity leads to reducing memory requirements for such giant models. This is only one example, several other workloads manifest sparsity such as graph analytics, finite element analysis [33], [60, 21].

In Sparse Matrices, the majority of values are zeros. For a given sparse matrix A, with dimensions $n \times m$, the total number of non-zeros nnz is much smaller than $n \times m$. Hence, traditional methods of storing and processing dense matrices are unfit when dealing with sparse matrices. This is mainly due to the unnecessary storage of zeros, which take up memory resources and exhibit poor performance for computations involving sparse matrices.

Figure 1.1a shows an example of a 10×10 sparse matrix, with only five non-zeros (marked). The remaining 95 elements are zeros. Storing this matrix as a dense matrix, requires the storage of all the zeros regardless of sparsity, resulting in a matrix size of

$10 \times 10 \times 4$bytes= 400bytes (assuming 32-bit single floating point precision). Additionally, computing on this matrix as a dense matrix introduces unnecessary operations by computing on zeros. Figure 1.1a shows an example of a dense representation of a 10×10 sparse matrix, with 95% sparsity. Figure 1.1b shows the a code example for a matrix-vector multiplication operation $y = A * x$ where y is a dense vector, A is a sparse matrix in a dense representation, and x is a dense vector. The code needs to iterate over all matrix elements, including zeros, to do the multiplications, even though the zeros do not affect the result stored in the dense vector y.



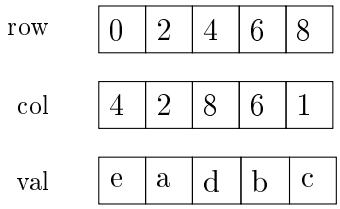
(a) Example sparse matrix.

```
1    for (int i = 0; i < n; ++i) {
2      y[i] = 0.0f;
3      for (int j = 0 ; j < m; ++j) {
4        y[i] += A[i][j] * x[j];
5      }
6    }
```

(b) Dense matrix-vector multiplication code

Figure 1.1: An Example Dense Representation of a sparse (mostly zeros) matrix, and the algorithm to perform a Dense Matrix-Vector Multiplication using the dense representation.

Due to the inefficiency of dense representation and algorithm for dealing with sparse matrices, various sparse storage formats were developed with the goal of storing only non-zeros in a sparse matrix. With the emergence of new sparse data formats, new algorithms are developed to iterate over these formats. The algorithm is tightly coupled with the sparse storage format used. Figure 1.2a shows the same example matrix represented using the COOrdinate (COO) sparse storage format. Each entry in the matrix is represented using a triplet: row index, column index, and the value. For this example, the storage requirement is 5 entries $\times$ 3 arrays $\times$ 4 bytes = 60 bytes, assuming 32-bit indices and 32-bit floating point values, which is 6.7x less storage required compared to the dense representation. Additionally, Figure 1.2b shows the code for Sparse Matrix-Vector Multiplication (SpMV)



(a) COO representation

```
for (int i = 0; i < n; ++i) {
  y[i] = 0.0f;
}
for (int i = 0 ; i < nnz; ++i) {
  y[row[i]] += val[i] * x[col[i]];
}
```

(b) Sparse Matrix-Vector multiplication code using COO

Figure 1.2: Representing the matrix in Fig. 1.1a as a sparse matrix in COO format, and the algorithm used to calculate Sparse Matrix-Vector Multiplication where tha operand sparse matrix is in COO format

where the operand sparse matrix is represented in COO format. One thing to notice is that the runtime complexity now is dependent on the number of non-zeros in the matrix (nnz) instead of the matrix dimensions in the dense case. For sparse matrices, nnz is significantly less than the matrix dimensions ($n \times m$), making sparse implementations more space and runtime efficient that dense implementations for sparse matrices.

A huge body of research introduced various different sparse storage formats [67, 38, 6, 72, 43, 104, 7, 45, 27, 84, 53, 11, 70, 20, 63, 30, 90]. For a given input sparse matrix, the performance of a sparse operation (e.g. SpMV) varies depending on many factors. One important factor is the used sparse storage format. Figure 1.3 shows SpMV performance in GFLOPs (vertical axis) for different matrices from the SuiteSparse Matrix Collection [21] (horizontal axis), using three different sparse storage formats: COO, Compressed Sparse Row (CSR), and ELLPACK (ELL) on AMD Radeon VII GPU. Two main points are clear from this experiment: (a) For the same input matrix, performance varies across different sparse storage formats, and (b) there is no single format that universally performs best across all matrices.

To this end, a lot of research work focused on using machine learning and auto-tuning techniques to find the optimal sparse format for an input sparse matrix [64, 88, 103, 81, 13, 17]. However, they fall short due to the following reasons:

1. They represent matrices based on a limited set of features, extracted through a pre-

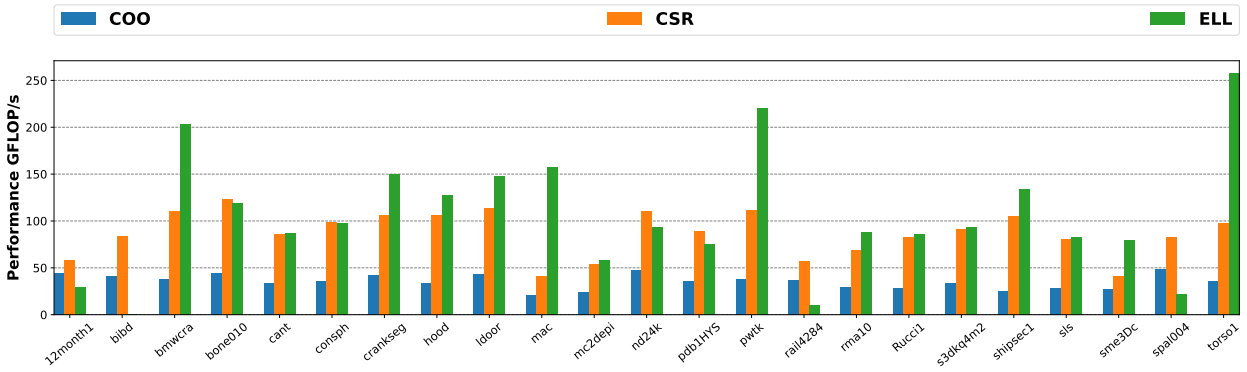


Figure 1.3: Performance (in GFLOP/s) comparison of rocSPARSE SpMV kernel on AMD Radeon VII GPU with 23 different sparse matrices from the SparseSuite Matrix Collection in COO, CSR, and ELL representations

processing step, or based on a fixed-size representation of the matrix, losing important information in the process.

2. They exclusively use a discrete set of matrices for training and inference, making the generalization of their conclusions challenging.

3. The proposed solutions are not extensible: the choice of the optimal format is based on a limited number of available sparse storage formats.

This dissertation addresses the above challenges through a tri-fold approach. The contributions of this dissertation are:

1. Proposing an extensible classifier framework based on Graph Neural Networks (GNNs). The framework takes as input a sparse matrix and produces as output the predicted class of shape (sparsity pattern) of the input matrix.

2. Proposing the usage of graph models to generate large-scale sparse matrices and study their structure. A modular framework is developed to study the sensitivity of sparse matrices and graphs structures to the variation of the input parameters of graph generative models. The framework takes an input a graph generative model and produces as output a sensitivity analysis of the structure to the input parameters, and to noise.

3. The introduction of a framework for performance analysis, prediction, and visualization of sparse matrix operations. The framework uses graph models as a parameterized way of studying performance. The goals of the framework are: (a) to find a reperenstative set of features in the sparse case to relate to performance (other than matrix dimensions), (b) to derive decision on the optimal sparse storage format for a given graph model with a set of parameters and target architecture, (c) to study the effect of noise in sparse data on sparse operation performance, (d) to study the end-to-end performance breakdown across the sparse application workflow, and (e) to provide appropriate visualizations of sparse performance using identified features.

4. Evaluation of high-level optimizations to accelerate sparse computations based on the sparse data characteristics such as the graph model used to generate the data and its parameters alongside with the used architecture and sparse operation.

The ultimate goal of this dissertation is to:

1. Automatically and quickly identify the non-zero and sparsity pattern of sparse data, with minimal memory and compute requirements.

2. Provide the tools for a comprehensive understanding of the structure of sparse matrices and graphs, based on how the data was generated.

3. Drive optimization decisions for sparse applications.

4. Layout the ground for future work that takes an input sparse data and architecture characteristics, and provides as output the ideal sparse storage format and a set of high-level optimizations and their configuration to achieve the best possible performance.

The rest of this dissertation is organized as follows:

- Chapter 2 provides the fundamental background needed for the rest of the dissertation.

- Chapter 3 discusses in detail the development of the GNN-based classifier framework for identifying sparse matrix structures.
- Chapter 4 describes the proposed framework for analyzing the robustness of graph models.
- Chapter 5 introduces the proposed framework for performance analysis and visualization for sparse computations.
- Chapter 6 explores high-level optimizations for sparse computations.
- Chapter 7 concludes the dissertation by providing a summary and future directions.

Chapter 2

Background and Related Work

In the realm of computational sciences and engineering, the abstraction of complex systems and relationships through graphs and sparse matrices has become a cornerstone of algorithmic development and data analysis. This chapter delves into the foundational aspects of graph theory and sparse matrix representation, elucidating their pivotal roles in modeling intricate network structures and facilitating efficient computational operations. By exploring various graph types, their properties, and the myriad ways in which sparse matrices can be stored and manipulated, we embark on a journey through the theoretical underpinnings and practical applications that underscore their significance. The evolution of storage formats and optimization techniques reveals a landscape marked by a relentless pursuit of efficiency, underscoring the ingenuity and innovation that characterize this domain. This chapter also sheds light on some of the significant work in literature dealing with sparse matrices and graph in terms of tuning important kernels (such as SpMV), introducing sparse data formats, architecture-aware auto-tuning and selection the optimal sparse format, and automatic generation of sparse kernels.

2.1 Graphs

In computer science, a *graph* is a data structure that consists of a set of *vertices* (or nodes) and a set of *edges* connecting these vertices. Graphs are widely used to model relationships between different entities. In graph theory, a graph G is formally defined as an ordered pair $G = (V, E)$, where:

V is a non-empty set of vertices (or nodes),

E is a set of edges, where each edge is a 2-element subset of V.

The elements of E represent connections between the vertices in V. If G is an undirected graph, the order of the elements in the subset is not considered, meaning $\{u, v\} = \{v, u\}$. If G is a directed graph, the order of the elements in the subset matters, and the edge is directed from the first element to the second.

Additionally, in the case of a weighted graph, each edge is associated with a numerical value or weight.

So, in mathematical terms:

$$G = (V, E) \quad \text{and} \quad E \subseteq \{\{u, v\} \mid u, v \in V\}.$$

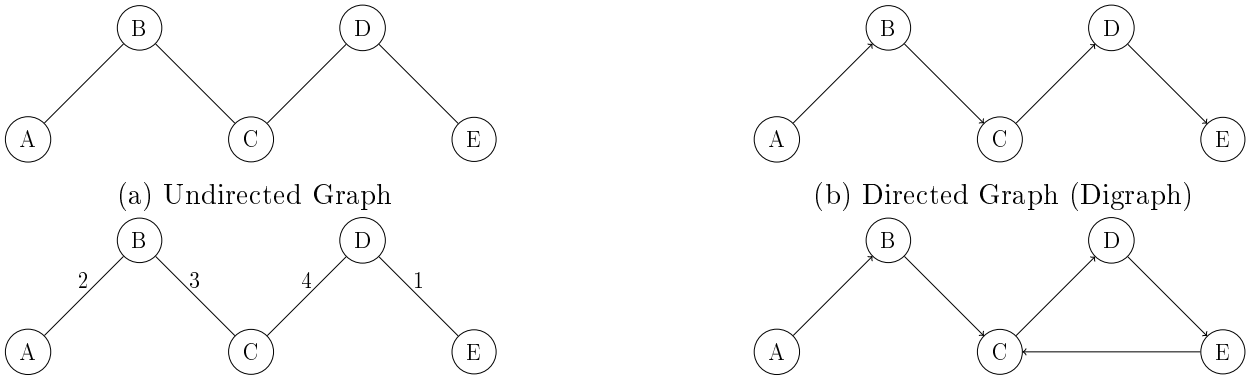
This formal definition captures the fundamental components of a graph and the relationships between its vertices and edges.

2.1.1 Types of Graphs

Graphs can be categorized into multiple categories. Examples (not an exhaustive list) of such categories are:

- **Undirected Graphs:** Figure 2.1a shows an example, where edges are bi-directional.

- **Directed Graphs (Digraphs):** Edges have a specific direction, that goes from one node (vertex) to another. Figure 2.1b shows an example directed graph.
- **Weighted Graphs:** Edges have weights assigned to them. Weighted graphs can be directed or undirected. Figure 2.1c shows an example of a weighted undirected graph.
- **Cyclic Graphs:** If a directed graph contains at least one cycle (loop), it is considered a cyclic graph as shown in Figure 2.1d.



(a) Undirected Graph (b) Directed Graph (Digraph)

(c) Weighted Graph (d) Cyclic Graph

Figure 2.1: Examples of different categories of graphs

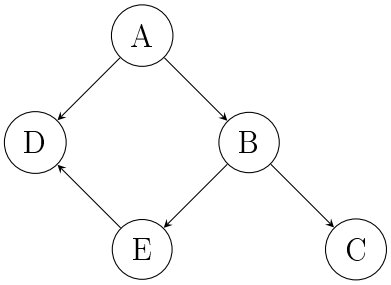
2.1.2 Representations of Graphs

Graphs can be represented in different ways, commonly using adjacency lists or adjacency matrices.

Adjacency List

In an adjacency list representation, each vertex maintains a list of its adjacent vertices (neighbors). Figure 2.2a shows an example directed graph of five nodes (vertices). To represent this graph using adjacency list, a list of neighboring nodes (vertices) is maintained for each of the five nodes. Figure 2.2b shows the adjacency list representation of this example graph. For undirected graph, if node A is connected to node B, then node B is also connected

to node A as the edges of the graph are bidirectional. Figure 2.3a shows the same graph as 2.2a, but now all edges are undirected. To accommodate for this change, Figure 2.3b shows the adjacency list representation of the undirected graph.



(a) Directed Graph

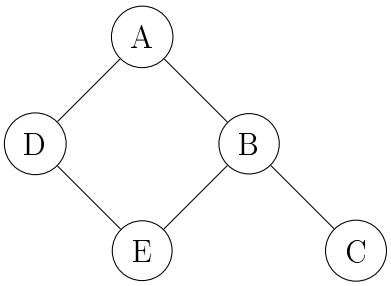
Node	Adjacency List
A	[B, D]
B	[C, E]
C	[]
D	[]
E	[D]

(b) Adjacency List

	A	B	C	D	E
A	0	1	0	1	0
B	0	0	1	0	1
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	1	0

(c) Adjacency Matrix

Figure 2.2: (a)Directed Graph and two Representations of it using (b) Adjacency List, and (c) Adjacency Matrix



(a) Undirected Graph

Node	Adjacency List
A	[B, D]
B	[A, C, E]
C	[B]
D	[A, E]
E	[B, D]

(b) Adjacency List

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	1
C	0	1	0	0	0
D	1	0	0	0	1
E	0	1	0	1	0

(c) Adjacency Matrix

Figure 2.3: (a)Undirected Graph and two Representations of it using (b) Adjacency List, and (c) Adjacency Matrix

Adjacency Matrix

Another way to represent graphs is using an adjacency matrix. In an adjacency matrix representation, a 2D array is used to represent the connections between vertices. The dimensions of the matrix are $|V| \times |V|$ where $|V|$ is the number of nodes (vertices) of the graph. If there exists an edge between v_1 and v_5, then the entry at index (v_1, v_5) will be set (1 in unweighted graph, or w_{15} in a weighted graph, where w_{15} is the weight on the edge between v_1 and v_5). Figure 2.2c shows the adjacency matrix representation for a directed graph, and Figure 2.3c illustrates the same representation for an undirected graph.

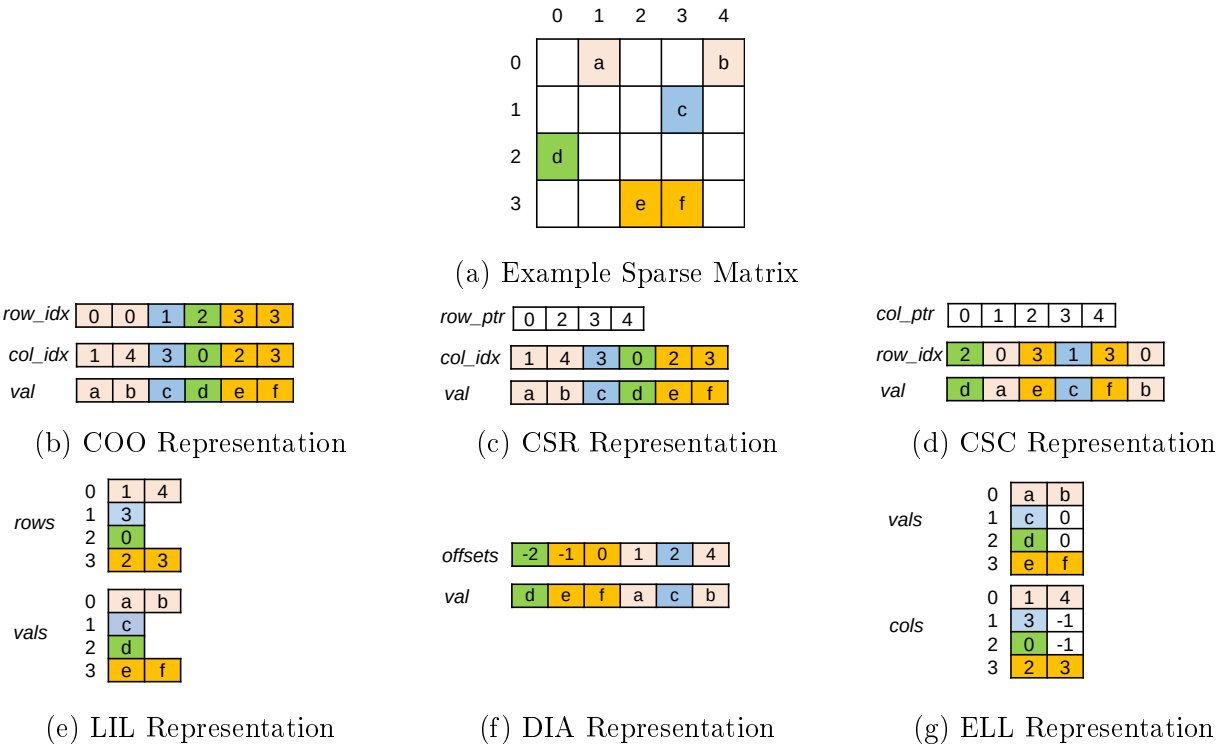
2.2 Sparse Matrices

Sparse matrices are data structures with a majority of elements being zero. They arise naturally in many scientific and engineering applications, such as the representation of adjacency matrices in graph theory, solutions to partial differential equations, and data encoding in machine learning algorithms. The sparse nature of these matrices typically stems from the systems they model; many real-world systems are comprised of elements with limited interaction, leading to a significant number of zero coefficients.

Working with sparse matrices requires algorithms that are fundamentally different from those used for dense matrices. The usual matrix operations such as addition, multiplication, and inversion need to be adapted to ignore the zero elements, which significantly improves computational efficiency. One of the main advantages of exploiting sparsity is the potential for reducing computational complexity. Operations that are quadratic or cubic in the dimension of the matrix for dense matrices can often be performed in linear or near-linear time for sparse matrices. This efficiency can be critical in applications such as solving systems of linear equations, eigenvalue problems, and performing various matrix decompositions. The utility of sparse matrices is underscored by their ability to handle large datasets and models that would otherwise be infeasible to manage with dense matrix representations.

Sparse matrices are pivotal in various applications, such as:

- Network analysis, where the connectivity can be represented by adjacency matrices that are typically sparse.
- Finite element methods in engineering simulations, which produce system matrices that are sparse due to locality of element interactions.
- Graph algorithms and data mining techniques that involve large, sparse adjacency matrices.



(a) Example Sparse Matrix

(b) COO Representation

(c) CSR Representation

(d) CSC Representation

(e) LIL Representation

(f) DIA Representation

(g) ELL Representation

Figure 2.4: Sparse Matrix Example (a) and its representation using (b) COO, (c) CSR, (d) CSC, (e) LIL, (f) DIA, and (g) ELL formats.

2.3 Sparse Matrix Storage Formats

Efficient storage and operation on sparse matrices are crucial for the performance of many algorithms and applications. To efficiently utilize the memory and computational advantages of sparse matrices, several storage schemes have been devised. Each format presents different advantages depending on the operations to be performed. Figure 2.4 shows an example sparse matrix and its representation using different common sparse storage formats. These include:

2.3.1 COOrdinate Representation (COO)

In this representation, three arrays are maintained as shown in Figure 2.4b: `val` linearly stores the non-zero values in the matrix, `row_idx` and `col_idx` store the corresponding row indices and column indices respectively for these non-zero values.

2.3.2 Compressed Sparse Row Representation (CSR)

CSR [78] stores the sparse matrix using 3 arrays as shown in Figure 2.4c: `val` stores the non-zero values, `col_idx` stores the corresponding column indices for these values, and `row_ptr` keeps track of elements that belong to each row. For example, in Figure 2.4c `row_ptr[0]=0` means that non-zero elements for row 0 of the sparse matrix, start at index 0 in both `val` and `col_idx`. The last non-zero element in row 0 is identified by looking at `row_ptr[1]` which in this example is 2. This means that non-zero elements of row 1 of the sparse matrix start at index 2 in `val` and `col_idx`, suggesting that index 1 is the last index for row 0 elements.

2.3.3 Compressed Sparse Column Representation (CSC)

This representation is similar to CSR, except that it records non-zero elements column-wise (instead of row-wise in CSR). As shown in Figure 2.4d-d, it also keeps track of three arrays: `val` for the nonzero values, `row_idx` to store the corresponding row indices for the non-zero values, and `col_ptr` stores pointers to the start indices of non-zero elements in `val` and `row_idx` for the corresponding column. In this example, `col_ptr[0] = 0` means that non-zero elements in column 0 of the sparse matrix, start at index 0 in both `val` and `row_idx` arrays.

2.3.4 List of Lists Representation (LIL)

LIL keeps maintains two structures: `rows` and `vals`. Each of them contains an array for each row in the sparse matrix that has at least one non-zero element. Arrays in `rows` store the column indices of each non-zero element in that row. Non-zero values for each row are stored in `vals` arrays. Figure 2.4e shows an example of LIL representation. Each row of the example sparse matrix contains at least one non-zero element, so LIL contains 4 arrays for `rows` and `vals`: one for each sparse row.

2.3.5 DIAgonal Representation (DIA)

Diagonal representation is the most efficient representation for matrices where all or most of the non-zero lie on or around the main diagonal. It maintains two arrays: `offsets` and `val`. `offsets` stores sorted offsets of non-zeros from the main diagonal. In Figure 2.4f, value d is at (2,0) in the original sparse matrix. For row 2 in the matrix, a diagonal element would be at index (2,2). This means that d is at offset 0-2= -2 columns from the main diagonal element. Offsets for all non-zero elements from the main diagonal are obtained and are sorted in an increasing order in both `offsets` and `val`.

2.3.6 ELLPACK Representation (ELL)

ELL representation utilizes two 2D arrays to store non-zero values, and non-zero column indices as shown in Figure 2.4g. The number of rows for each of the arrays is equal to the number of rows in the original sparse matrix. The number of columns in these arrays is equal to the maximum number of non-zeros per row. If a row contains fewer non-zeros than the maximum number of non-zeros per row, the corresponding entries in the values and column indices arrays are padded with 0s and -1s respectively. One advantage of ELL representation is the regular access (as compared to CSR for example), which enables further loop optimizations and parallelization techniques. However, a major drawback rises when the variation of the number of non-zeros per row is high, since the memory storage required for the values and column indices can be very high, depending on the maximum number of non-zeros per row.

2.3.7 Doubly-Compressed Sparse Column Representation (DCSC)

DCSC [14] evolved as an improvement over CSC for hypersparse matrices, in which the number of non-zeros is much lower than the dimensions of the matrix. The problem in CSC arises when many columns do not contain any non-zeros, which means that the *col_ptr*

array of CSC will store the same index multiple times (`|col_ptr| = n` where n is the number of columns). DCSC only keeps track of columns with at least one non-zero value. Hence, `|col_ptr| = nzc` where *nzc* is the number of columns with at least one non-zero. However, one more level of indirection is needed to map the new column pointers to the original column pointers.

2.4 Optimizing Sparse Operations

2.4.1 Linear Algebra Libraries

Intel Math Kernel Library (MKL) [91] is a general purpose sparse linear algebra library, written in C/Fortran. It contains optimized kernels for Intel CPUs for a variety sparse matrix computations. cuSparse [74] is NVIDIA's library for sparse matrix computations on GPUs, offering both row-major and column-major access patterns. AMD's GPU counterpart library is rocSPARSE [5]. Eigen [36] is a header-only, C++ template-based library that provides high performance linear algebra operations and is used in Google's Tensorflow.

2.4.2 Architecture-Specific Optimizations

Merge-based CsrMV [73] introduced a parallel, load-balancing method to perform SpMV on CPUs and GPUs using the CSR format. The method partitions the input matrix equally among threads, so that all threads have the same workload, regardless of any irregularities in the sparsity structure of the matrix (rows with too many or too few non-zeros for example). Results show predictable performance with minimum correlation to the non-zero distribution across the matrix.

Su and Keutzer [86] proposed clSpMV, a runtime auto-tuning framework built on top of OpenCL [85] that analyzes and chooses the best sparse matrix representation for the target platform. clSpMV also introduces a new sparse format called the "Cocktail Format". Results show that clSpMV performs between 63.3% and 83% better than NVIDIA's cuSparse SpMV

implementation.

YaSPMV [97] addresses the same problem of load imbalance in SpMV kernels, in addition to memory bottleneck. In order to tackle these problems, this work introduced the blocked compressed common coordinate sparse format (BCCOO) to enable blocked SpMV execution on GPUs. This format split the matrix into vertical segments for improved cache locality. This work also proposes an efficient SpMV scan algorithm to mitigate load imbalance, and an auto-tuning framework to optimize SpMV parameters based on the input matrix features. Results show performance gains over cuSparse v 5.0 between 42% to 65% depending on the GPU architecture, and between 40% and 70% over clSpMV [86].

Choi et. al [18] presented a model-driven auto-tuning framework for SpMV on GPU using blocked formats. The model includes GPU hardware characteristics into consideration when optimizing execution parameters. They introduced hand-crafted SpMV kernel based on the block ELLPACK (BELLPACK) format. The auto-tuner has an offline and online phase. The online phase is required to determine block size parameters. Results show that the framework can achieve within 15% of the best global performance obtained by exhaustive space search.

Ortega et. al [75] introduced a fast library implementation of SpMM on GPUs that attempts to hide the memory access latency with a high operational intensity ratio, exploiting ELLPACK-R format, and using streaming accesses to minimize the CPU-GPU communication latency effect.

Liu and Vinter [69] studied SpGEMM kernels that involve irregular matrices as operands on GPUs. Their work focuses on mitigating two major problems: memory pre-allocation of the result matrix, since the number of non-zeros is unknown before the actual computation, and the expensive insertion of values at random indices due to lack to locality. For the result matrix allocation, they introduced a parallel algorithm to calculate the upper bound of non-zeros for each row and assigning each row to a bin of 38 possible bins according to the number of non-zeros per row and allocate a temporary result matrix. For insertions, they

use a parallel GPU merge algorithm. Results show speedup over cuSparse and Intel MKL implementations.

CSX [52] targets the optimization of SpMV on shared-memory systems where the performance of the kernel does not scale well with increasing the number of processors. It targets reducing the volume of metadata communicated through the shared memory system by introducing a new compressed format for the sparse matrix metadata. The newly introduced format depends mainly on the fact that sparse matrices represent real-life physical structures, and they include repeated patterns or regularities. By exploiting these regular substructures horizontally, vertically, and diagonally, CSX can represent sparse matrices in a more compressed form (than CSR for example). The preprocessing cost can be partially hidden by either doing it offline, or perform sampling while executing online preprocessing.

Adaptive Sparse Tiling (ASpT) [40] focuses on optimizing the performance of two specific sparse matrix kernel: sparse-dense matrix multiplication (SpMM) and Sampled Dense Dense Matrix Multiplication (SDDMM). The key motivation of this work is the complexity of applying tiling to sparse matrix formats. This is because the irregular matrix access patterns when sparse matrices are involved. The high-level idea of ASpT uses a hybrid of two techniques: 2D tiling, and no tiling (using regular row-wise access) according to the number of non-zeros in a column segments. First, ASpT splits the matrix into a number of row panels, each panel consists of a fixed number of consecutive rows which depends on the cache size. Within each panel, columns are sorted according to the number of non-zero elements in each column. ASpT defines a target-specific threshold that differentiates heavy column segments from sparse column segments. Assuming that this threshold is T_h, column segments with $nnz >= T_h$ are considered heavy, while column segments with $nnz < T_h$ are considered light. Now, each row panel is divided into two tiles: a dense tile with all the heavy segments, and a sparse tile with all the light segments. The dense tile is further divided into 2D tiles in which the tile width is determined by cache capacity. Column segments are then ordered according to the number of non-zero elements: column segments with the highest number of non-zeros

comes first. Heavy column segments are processed using 2D tiling, while light segments are processed using traditional CSR row-wise execution. The paper reports significant speedups between 7.26x to 30.15x compared to other state-of-the-art solutions.

2.4.3 Machine Learning Techniques

Selection of Sparse Matrix Representation on GPU [81] is one work that addresses the issue of choosing an appropriate sparse matrix representation with the goal of optimizing performance. That work focuses mainly on optimizing SpMV operation on GPUs. They introduced a machine learning model using decision trees to select the best sparse matrix representation for a given matrix on a given target to achieve the highest performance possible. For the dataset, they used 682 matrices from 114 in the University of Florida sparse matrix collection [21]. The feature set used in the training and testing is mainly about the sparsity structure of the matrix such as: number of rows, number of columns, number of non-zeros, fraction of non-zeros, non-zeros per row (minimum, maximum, average, and standard deviation), non-zero blocks per row (minimum, maximum, average, and standard deviation), and size of non-zero blocks per row (minimum, maximum, average, and standard deviation).

SMATER [88] extends SMAT by supporting more storage formats and including a flexible backend containing both Intel MKL implementation and their hand-tuned implementation. The backend is intended to be flexible to include any BLAS library for multiple targets (multi/many core CPUs and GPUs). SMATER training set is also the University of Florida sparse matrix collection. The feature set is similar to SMAT, with the addition of two features to represent the estimated dense sub-block row and column size (to enable support for blocked sparse formats). Results show a 2.5× performance improvement on average when compared with Intel MKL.

Benatia et. al [13] proposed a similar machine learning based approach for SpMV on GPUs. Their results show that the prediction model achieves 98% of the performance of the optimum format selection. They consider COO, CSR, ELL, and HYB as candidate formats.

Sparse matrix features are extracted and fed to a support vector machine (SVM) classifier to be trained on them. The extracted feature set represents the sparsity structure of the matrix, similar to the previous two works.

Lehnert et. al [55] introduced a Linear Regression and k-Nearest Neighbors (KNN) models to predict the performance of SpMV on GPUs using different sparse matrix formats. In addition to the regular matrix sparsity structure features, they added CUDA block size as a feature to take GPU thread mapping a factor into the decision process.

2.4.4 Sparse Matrix Formats

HiCOO (Hierarchical COOrdinate) [63] is another hierarchical format for sparse tensors. The motivation behind HiCOO was the mode-dependent nature of most sparse representations and their hierarchical variants. Mode here is referring to the dimension. For example, CSR is mode-dependent because the access pattern (iteration) is dependent on the row (mode 1) structure. In CSR, it is efficient to randomly access any row. However, in order to find element at index (i, j), we cannot find the value for j directly. We need to search the structure (array) storing non-zero values for row i, and then get element (i, j). So, we say that CSR is mode-1 dependent. One motivation to introducing HiCOO is to be mode independent. The HiCOO format represents tensors as blocks in a sparse pattern. For every non-zero value, HiCOO stores a pointer to the containing block (start address of the block), sparse block indices, and non-zero indices within the block. This provides a compact storage, compared to COO, if the number of non-zeros per block is greater than 2. The paper also uses HiCOO format for the Matricized Tensor Times Khatri-Rao Product (MTTKRP) operation. A parallel variant of the algorithm based on HiCOO is also introduced. Results show a significant speedup (up to 3.5× compared to the COO format and 4.3× compared to CSF format).

CSR5 [70] was proposed as an improvement over the traditional CSR format, for SpMV on CPU and GPU targets. One major strength of CSR5 is its independence of the matrix

sparsity structure. It provides high throughput SpMV regardless of regularity of the input matrix. CSR5 was evaluated against multiple existing formats on different architectures, using a benchmark suite of 24 matrices: 10 irregular and 14 regular ones. Results show that CSR5 achieves similar or better performance for the regular matrices, while achieving an average performance improvement between 17.6% and 293.0% depending on the target architecture.

Hierarchically Tiled Arrays (HTAs) [30] introduce a class to split matrices into multiple levels of representation. The first level partitions the matrix into tiles. These tiles can be further partitioned into tiles (second level) or remain untiled. The outer level (first level) can be distributed among different processors to achieve computation parallelism. Inner tiles can be processed locally to exploit locality. Tiles can be referenced as a whole. For example, if the original matrix M is partitioned into 4 square tiles of size n, we can access the second tile using the notation M1,1. We can access the first element in this tile using the expression M1,1(1,1), or alternatively the tile can be flattened and the same element can be accessed using M(1, n+1). In order to carry on an operation over two HTAs, they need to be conformable, meaning that they must have the same number of levels and shape.

Recursive Matrix and Vector (RMV) [90] applies a similar idea to scale-free graph. The idea is based on the observation that these graphs contain a few vertices with many edges, and a large number of vertices with few edges. If we zoom in these vertices with few edges, we can see that the same pattern repeats. Based on this recursive pattern, RMV introduced a hierarchical storage representation of graphs as matrices. The lowest level of hierarchy consists of the actual matrix values. Going up, each level represents a coarser view of the bottom-most level. Upper levels structures are called containers. Each structure consists of multiple fields: whether this is a container or a value, the number of columns and rows which can be accessed by this structure, and a bit matrix describing the pattern of non-zero elements. The main objective for RMV is to preserve the locality in both graphs and the cache, which was not supported by other sparse formats for this type of graphs.

2.4.5 Domain Specific Languages and Compiler-based Approaches

The Tensor Algebra Compiler (TACO) [50] is the first compiler that generates entire kernels for both sparse and dense tensors. TACO takes two inputs: a tensor expression that consists of an arbitrary number of operands and operators. The second input is the data format for each dimension of the involved operands. Each operand dimension can be either dense or sparse. For sparse dimensions, a format can be specified such as CSR, CSC, COO, and DIA. The output is a single kernel code generated to evaluate the specified expression with the specified tensor formats. TACO generates kernel code by introducing an intermediate representation called iteration graph, when then uses "Merge Lattices" to merge elements at matching indices, depending on the operation. Iteration graphs are graph representations that describe how to iterate over the indices of sparse dimensions, while merge lattices describe how to merge indices that access dimensions of more than one tensor. For example, for a SpMV, merge lattices operate at the intersection of the indices of the sparse matrix and the dense vector. On the other hand, for dis-junction operation such as vector addition, lattices operate at the union of the indices of both vectors. Code is generated lattice point by lattice point, introducing the appropriate loop boundaries and necessary code for merge operations. TACO results show competitive performance when compared to other popular hand-crafted libraries.

Tiramisu [8] is another compiler that uses the polyhedral model to generate efficient code for different domains such as: image processing, deep learning, and linear algebra (including sparse). It generates code for CPU, GPU, and distributed multi-node systems. Tiramisu leverages a 4-level intermediate representation, which enables the decoupling of the algorithm, the schedule, computation processing unit location, and the data representation and layout. Results show comparable performance to hand-tuned library implementations.

Taichi [41] is a domain-specific language for 3D visual computing that exploits spatial sparsity (regions of interest are smaller than the containing bounding volume). It addresses the issue of the complexity of sparse data structures, specifically the time spent handling

sparse structures due to indirection and parallelization inefficiencies. The approach uses is Taichi depends mainly on separating the computational kernel from the data structures. It provides a separate hierarchical data structure description language, which is, along with imperative computation language, is then transformed into an intermediate representation and data structure access optimization by the compiler. Then, the runtime system performs auto parallelization and memory management. Finally, code is generated for a choice of CPU or GPU backends. Results show that code generated by Taichi is 7× to 13× shorter and 1.2× to 13× faster than state-of-the-art counterparts.

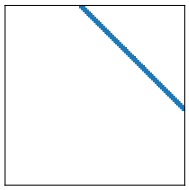
2.5 Summary

This chapter has traversed the basic background needed for the rest of the dissertation, especially on graphs and sparse matrices, shedding light on their profound impact on modern applications and workloads across different domains. The chapter explored some of the common graph types and their representations and showed the difference between the dense and sparse representation of data in terms of storage and needed algorithms. The examination of sparse matrix storage formats and optimization strategies has further demonstrated the relentless quest for computational efficiency, highlighting significant advancements and emerging trends. As we reflect on the contributions of pioneering works and contemporary studies, it is evident that the journey of innovation in graph theory and sparse matrix manipulation is far from over. The ongoing exploration of new storage schemes, optimization techniques, and applications promises to unlock even greater potentials, propelling us towards novel discoveries and advancements in computational science and beyond.

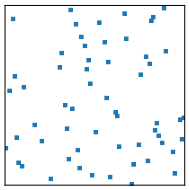
Chapter 3

Using GNNs to Identify Sparse Matrix Structure

3.1 Introduction



(a)



(b)

	Diagonal	Random	Rand+Diag	Kronecker
Diagonal	**500**	0	0	0
Random	0	**490**	10	0
Rand+Diag	0	40	**460**	0
Kronecker	0	20	0	**480**

(c)

Figure 3.1: Efficacy of our classifier framework at determining structure when the data is permuted. (a) is an off-diagonal matrix, (b) is a re-labelled variant of (a), and (c) is the confusion matrix for the classifier framework on re-labelled matrices similar to (b). By using GCNs our approach is invariant to node labelling and achieves around 97% accuracy.

Sparse matrices represent a fundamental building block used throughout the field of scientific computing in applications, such as graph analytics, machine learning, fluid mechanics, and finite element analysis [60, 21]. Such matrices appear as operands in numerous fundamental computational kernels such as sparse matrix-vector multiplication (SpMV), Cholesky factorization, LU factorization, sparse matrix -dense matrix multiplication, and matricized tensor times Khatri-Rao product (MTTKRP) among others. Building efficient algorithms

for this class of kernels mainly depends on the storage format used for the sparse matrix as observed in different studies [17, 12]. A variety of such formats are proposed in literature [29, 54]. Hence, it is crucial to identify the structure of the matrix to choose the ideal sparse format, and eventually tailor the algorithm to that format to optimize the workload performance. However, identifying the structure of the matrix is not always trivial. Figure 3.1a shows a spy plot of an off-diagonal sparse matrix, and Figure 3.1b shows the same matrix, with some of the original row indices and column indices re-labelled. It is less obvious for the latter figure to provide an insight of the original structure of the non-zeros within the matrix. Additionally, in case of huge sparse matrices, we might only have access to samples of the matrix. This could be because of computational or storage restrictions, or missing data. In these two cases (re-labelling and sub-sampling), we need efficient techniques to recognize the shape of the input matrix.

To tackle mentioned issues, we propose a framework to identify sparse matrices structures, using graph neural networks. Figure 3.1c shows the confusion matrix for the proposed framework using four sample classes on re-labelled variants. The framework design is modular, allowing users to easily augment it with new structures generators or feature sets. The main contributions of this chapter are as follows:

- Proposing a novel, modular Graph Neural Network framework to accurately predict the shapes of sparse matrices, including partial samples, and re-labelled variants of original matrices.
- Presenting a new balanced synthetic dataset for structured sparse matrices.
- Providing a performance analysis of graph-level classification on sparse matrices, using different feature sets.
- Introducing two new compact and efficient feature sets for matrices as graphs, namely: Linear and Exponential Binned One-Hot Degree Encoding.

The rest of this chapter is organized as follows: Section 3.2 introduces the necessary background and related work, Section 3.3 details the design of the proposed framework, Section 3.4 discusses the evaluation and results of the framework. Finally, Section 3.5 summarizes the findings of the chapter.

3.2 Background and Related Work

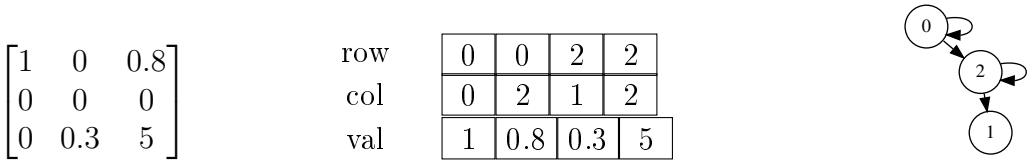
3.2.1 Graphs as Matrices

A Graph is a collection of nodes or vertices, and edges or links that connect pairs of nodes. Graphs are used to represent and analyze complex relationships and connections between entities such as social networks, transportation systems, electrical circuits, chemical molecules, etc. There are many types of graphs, including directed and undirected graphs, weighted and unweighted graphs, bipartite graphs, and more. Graph theory is the branch of mathematics that studies graphs and their properties.

Representing graphs as matrices has several benefits. Matrices provide a convenient way to perform various computations on graphs. For instance, matrix multiplication can be used to compute the number of paths between two nodes in a graph. Moreover, graph traversal algorithms such as depth-first search and breadth-first search can be efficiently implemented using linear algebra operations [47]. Additionally, matrices can be used to represent graphs in machine learning applications, allowing the use of standard machine learning techniques such as deep learning and clustering. Matrices also provide a way to visualize graphs using tools such as heat maps, which can help in identifying patterns and structures in the graph. Finally, storing graphs as matrices can be more efficient than other representations in terms of storage and access. This is because matrices can be stored and accessed using standard techniques for numerical data.

Overall, representing graphs as matrices provides a powerful and flexible way to analyze, visualize, and solve problems on graphs, making it a popular choice for many applications.

3.2.2 Matrices as Graphs



(a) Dense (b) COO (c) Graph

Figure 3.2: The graph/matrix duality allows us to represent graphs and matrices using the same data formats, and more importantly allows us to use Graph Convolutional Networks on the graphical representation of sparse matrices to recover the structure from local (node) observations.

Matrices can be represented as graphs as shown in Figure 3.2. An example of a dense representation of a matrix is shown in Figure 3.2a, as it consists of different rows and columns and the intersection of a row and a column stores a value. A more compact representation for a sparse matrix is the COOrdinate format, as shown in Figure 3.2b where three arrays are maintained: the row indices, the column indices, and the non-zero values. Only entries for non-zero values are stored. Additionally, by looking at the COO representation of the matrix, one can also represent the matrix as a graph where nodes (vertices) are row/column indices, and the existence of an edge indicates the existence of a non-zero value. The weight of an edge can be used to represent the actual non-zero value if needed. Figure 3.2c shows an example of such representation. For the first entry of the matrix (0,0,1) we have an edge (non-zero) between row 0 and column 0. Since they both are of the same value (0), we can represent them as one node in the graph with a self-loop. We can optionally add a weight to that edge with the value 1 indicating the actual non-zero value stored at that location.

Graph representation of matrices is specifically beneficial in different ways: (1) Graphs can provide a compact representation of the matrix, reducing storage and memory requirements, (2) It allows for the application of graph theory techniques to the analysis of matrices. For example, properties such as degree, diameter, clustering coefficient, and centrality measures can be computed for the corresponding graph representation of the matrix. These properties can provide insights into the structure and behavior of the matrix that might

not be immediately apparent from its matrix form (whether it is dense or sparse). Another benefit is that it enables the use of graph algorithms and techniques for matrix operations. For example, graph traversal algorithms can be used to compute matrix products, which can be useful for large and sparse matrices. Additionally, graph-based algorithms such as PageRank can be applied to matrices to matrices to perform tasks such as ranking, clustering, and dimensionality reduction. Finally, representing matrices as graphs enables the use of a powerful class of machine learning models on them to predict graph-level, node-level, or edge-level parameters. This class of graph enabled machine learning models is Graph Neural Networks.

3.2.3 Graph Neural Networks

Graph neural networks (GNNs) [79] are a class of deep learning models that operate on graphs or networks, which are commonly used to represent complex structured data such as molecules, social networks, electronic circuits, road networks, and many more. Unlike traditional neural networks that operate on structured data such as images or sequences, GNNs can handle arbitrary graph structures with varying node and edge attributes, enabling them to learn powerful representations of graph-structured data.

The key idea behind GNNs is to iteratively update node embeddings by aggregating information from the embeddings of their neighbors, which is based on the graph convolution operation. By stacking multiple layers of graph convolution and non-linear activation functions, GNNs can learn hierarchical representations of the graph that capture both local and global information.

GNNs have shown remarkable success in a variety of applications such as node classification, link prediction, and graph generation. For example, GNNs have been used to predict protein-ligand binding affinity, recommend items in e-commerce platforms, and detect fake news in social media. GNNs have also been extended to handle dynamic graphs that evolve over time, and to incorporate additional information such as node features, edge attributes,

and attention mechanisms.

3.2.4 Structured Matrices

Several common structures are observed in sparse matrices, such as:

Diagonal all non-zeros are located on the main or a secondary diagonal. This structure represents a 1D mesh and commonly appears in various scientific and engineering applications.

Random the non-zero elements are randomly distributed across the matrix, with variable density. Such matrices have no specific identifiable structure.

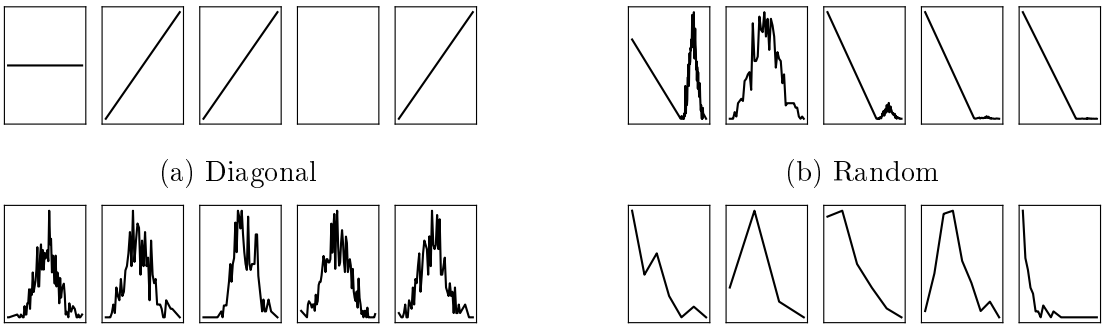
Kronecker Graphs [56] are a class of synthetic graphs that have been widely used to model real-world networks such as social networks, biological networks, and communication networks. These graphs are generated by recursively applying the Kronecker product of a small base graph with itself, typically of size 2×2 or 3×3. Let A and B be two matrices. Then, their Kronecker product $A \otimes B$ is given by

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix} \tag{3.1}$$

where a_{ij} are the entries of A. The resulting graph has a power-law degree distribution and exhibits a hierarchical structure that captures both the local and global connectivity patterns of the underlying real-world network.

We use these three classes of structures, and a combination of them, as a representative set that can be combined to form more complex relationships [76, 89]. Our framework is not limited to only these structures, and they serve as an example to evaluate its performance.

Degree as a representative node feature Figure 3.3 illustrates that one can accurately distinguish between the different classes based on the degree distribution of the representative graph. For example, for Diagonal matrices (Figure 3.3a), the degree for all nodes is low, and is either constant or linear across all nodes. Kronecker graphs follow a power-law degree



(a) Diagonal (b) Random

(c) Random+Diagonal (d) Kronecker Graph

Figure 3.3: Global Degree Distribution for samples in each matrix (graph) class studied in this chapter. In our approach we classify the shape based on local views from sampled data.

distribution, with only a few nodes having many connections (high degree) and most of the nodes having relatively few connections (low degree). However, only the local per-node degree view may be immediately available, and not the global graph view. An example of such a case is only having a sample of the graph and not the entire graph due to storage or computational limitations. The power of GNNs can be leveraged to carry out the required task: the prediction of the sample matrix structure.

3.2.5 Prediction on Sparse Matrices

Several studies investigated the use of machine learning to predict the optimal sparse format for SpMV on CPU and GPU [103, 88, 86, 66, 65, 13]. Our framework does not directly predict the best sparse format, instead, we only predict the structure of the input matrix. This allows de-coupling the sparsity pattern from the sparse format, following the argument adopted by AlphaSparse [25] since our framework also allows the seamless integration of new classes. Existing techniques collect a set of features from each matrix such as: the number of diagonals, the ratio of true diagonals to total diagonals, the (maximum) number of non-zeros per row, the variation of the number of nonzeros per row, the ratio of nonzeros in DIA and ELL data structures, and a factor or power-law distribution. We only need to calculate one feature per node: its degree. Also, [103] uses a CNN approach to treat matrices

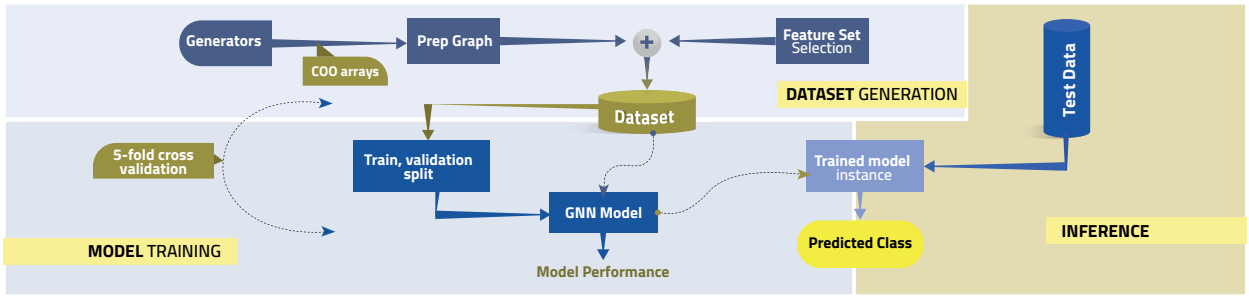


Figure 3.4: High-Level overview of the framework. It consists of three main phases: **dataset generation**, where the synthetic sparse matrices are generated, prepared as graphs, and have feature set attached. Then, the GNN **model training** using 5-fold cross validation to capture the model performance, and then generate a trained model instances, that is used later in the **inference** phase.

as images, and in order to fix the size of the matrix, they normalize input matrices into fixed size blocks, losing partial matrix information in the process. In contrast, our approach handles arbitrary sizes of matrices, without losing precision, leveraging the power of Graph Neural Networks. We can optionally sample large matrices and maintain high prediction accuracy. An additional benefit to our framework is that it is order agnostic, since matrices are represented as graphs.

3.2.6 Graph Representation for Learning

Representing non-attribute graphs is an open problem [19]. Common approaches employ graph properties such as node degree, more specifically a one-hot encoding of the degree [96]. One-hot encoding suffers from numerous limitations (Section 3.3.3). LDP [15] provides a compact representation for graph using five features per node. Although the computation of such feature vector is efficient, using LDP results in unreliable model performance for our task (Section 3.4.2). Both our representations (LBOH and EBOH) outperform one-hot encoding and LDP while addressing their shortcomings.

3.3 Framework Description

The goal of the proposed framework is to predict the structure of the input sparse matrix through its classification into one of the configured target classes. We use diagonal, random, diagonal+random, and Kronecker graph as examples of these classes to evaluate the performance of the framework. New structures can be seamlessly integrated. Figure 3.4 shows a high-level overview of the proposed framework. It consists of three main stages: Dataset generation, Model Training, and Inference. A synthetic dataset is generated using different generators for different shapes of matrices, which are then represented as graphs. In the training phase, we use GNN with 5-fold cross validation to evaluate the model performance. Finally, the trained model instance is used for later inference.

Procedure for using the trained model is as follows: the input sparse matrix is a matrix market file (`.mtx`) in triplets (COO) format, which is then used to build a graph representation of the matrix, as discussed in section 3.2.2. The input coordinate representation can be thought of as the adjacency list of the graph: at each index i of the row array and col array, there exists an edge (a non-zero) in the generated graph. The union of the unique sets of the row and col arrays is the set of vertices (nodes) in the graph. The actual non-zero values are not recorded, since the shape of the matrix does not depend on them. The existence of a non-zero (an edge) is enough information for our approach to decide the class the input matrix belongs to.

The output is a prediction (label) that specifies the class with highest probability that the input matrix belongs to, according to the model. In the rest of this section, we discuss in more detail the different building blocks in our proposed framework.

3.3.1 Dataset Generation

We generate a balanced dataset of 40K synthetic sparse matrices, covering the four sample classes through individual generators: : Diagonal, Random, Diagonal + Random, and Kro-

necker Graphs. Each of the generators returns a Coordinate (COO) representation for the matrix, excluding the actual non-zero values which provides an additional benefit of reduced storage for the dataset needed to train the model. The COO representation is then used as the adjacency list to build the graph representation.

In this section, we discuss the process of generating each.

Diagonal Generator

generates diagonal sparse matrices that can be either main diagonal, upper diagonal, or lower diagonal matrices. The generator takes as input two parameters: the size of each dimension, and the diagonal position.

Random Generator

generates random sparse matrices. The density of the matrix (ratio of the number of non-zeros to the total number of elements in the matrix) is adjustable in the range of 0% (fully sparse, no non-zeros) and 100% (fully dense, all elements are non-zeros). The default density used for the generated matrices is 20%.

Random+Diagonal Generator

generates random square sparse matrices with a main diagonal. The density of the generated matrix is also adjustable using the same method of generating Random matrices.

Kronecker Graphs Generator

generates Kronecker Graphs, starting from a 2×2 initiator matrix $K1$. The initiator matrix $K1$ is fully dense (all 4 values are non-zeros). The generator takes the number of times the Kronecker product will be applied as a parameter. For example, if $times = 2$, it generates $K3 = K1 \otimes K1 \otimes K1$ with dimensions 8×8. Then, a uniform random probability matrix

is generated with the same size as the resulting Kronecker product, and used to mask the product to generate the final Kronecker graph.

3.3.2 Dataset Preparation

For the development of the graph neural network model, we use PyTorch Geometric [28].In order for the generated matrices to be compatible with PyTorch Geometric, they are expected to be converted into the library's `data.Data` objects. In order to construct such objects, the following attributes are needed:

- `edge_index`: a 2D tensor representing the edges of the graph. Follows the same format as COO arrays. The generators described in section 3.3.1 already return the row and column indices arrays separately. Simply stacking them using `torch.stack` generates the required tensor for `edge_index`. A requirement for the edge index array is that the maximum row or column index should be less than the total number of nodes in the graph. To satisfy this requirement, the generated sparse matrices pass first through an encoder we implemented to re-label the indices. Also, a separate function is implemented to explicitly calculate the total number of nodes in the graph.

- `x`: a 2D tensor representing the nodes feature matrix. Each node needs a vector of dimension d representing the features of this node. For this chapter, different representations of degree are evaluated as node features. The shape of this tensor is `num_nodes` $\times$ `d`. A more detailed analysis of the chosen features for input graphs is provided in Section 3.3.3.

- `y`: a tensor representing the labels. In this work, the learning task is a graph-level classification task, so labels are one per graph. A numerical representation for the different labels (classes) is used as follows: 0 for diagonal, 1 for random, 2 for random+diagonal, and 3 for Kronecker.

3.3.3 Feature Set Selection

A per-node feature vector is necessary for the graph neural network to classify matrices. Node degree can be calculated for rows/columns in input matrices from their graph representation.

One-Hot Degree Encoding Since node degree appears to be an efficient feature to identify the graph structure, we use one-hot encoding of the node degree as a feature vector. For one-hot encoding, the number of features (length of the feature vector) is equal to the maximum degree in the dataset + 1. However, this kind of representation suffers from the following shortcomings:

1. To build the feature vector for each node in each graph, the knowledge of the maximum degree across the entire dataset is needed in advance. In order to overcome this limitation, for each graph, we store only the labels (`y`) and the adjacency list (`edge_index`) as discussed in Section 3.3.2. After we have iterated over all the matrices, we record the maximum degree seen across all graphs into a file. Every time we load a graph for training or testing, we attach the one-hot degree encoding to the graph object, since the maximum degree across the entire dataset is known at this point.

2. In our synthetic dataset, the maximum degree recorded across all graphs is 7710, meaning that for each node in each graph, the feature vector of node i : $x[i]$ needs to be of length 7711, even if the actual degree for that node is 0. This adds unnecessary storage requirements and slows down the training process, since graphs will be huge and batch sizes need to be close to or equal to 1 to fit in GPU memory.

3. The node feature vector length is not fixed across different training datasets and depends on the maximum degree found in the training dataset. This requires the knowledge of the maximum degree found in the training phase even in the inference phase, which is not ideal for model deployment, since a fixed feature dimension that is independent of the training data is preferred.

4. Inference on graphs with a maximum degree greater than the maximum degree that

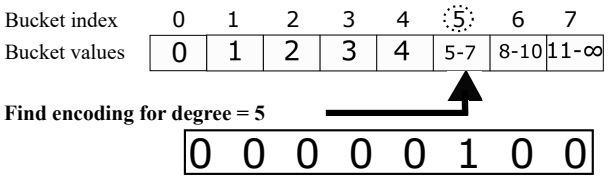


Figure 3.5: An example of finding the linear binned one-hot degree encoding for a node with degree = 5, where the parameters for the encoding scheme are $\alpha = 5$, $\beta = 3$, and $k = 2$. Degree 5 is mapped to its associated bucket (5 to 7), then the bucket index (5) is represented using one-hot encoding (1 at the position where the value 5 exists, 0 otherwise).

the model was trained on is not straightforward. For example, if the model is trained on our synthetic dataset with maximum degree = 7710 (feature vector dimension = 7711), and we try to infer the class of a graph with a maximum degree of 7711, building the feature matrix of the input graph fails (since it assumes the same feature vector dimension for input data). This problem can be indirectly solved by sub-sampling the graph and inferring on a smaller sample graph where its maximum degree is less than or equal to 7710.

Local Degree Profile (LDP) [15] captures the local structural information of nodes in their immediate neighborhood. LDP is calculated for each node as a five-feature vector: the node degree, the minimum degree of its neighbors, the maximum degree of its neighbors, the mean degree for its neighbors, and the standard deviation of the degrees of neighbors. LDP features are easy to compute for any given graph. Additionally, the number of features per node is fixed, regardless of the used training data. LDP incurs low storage overhead.

Linear Binned One-Hot Degree Encoding (LBOH) We implement a modified version of one-hot encoding, to address its limitations. LBOH works by having a fixed number of buckets for representing one-hot degrees. Buckets ranges are designed as follows: a set individual sequential buckets from 0 t α (inclusive) where α is a small integer (less than 10). Then, we add a set of buckets with fixed step β from α: $(\alpha+\beta)$, $(\alpha+2\beta)$, ..., $(\alpha+k\beta)$ where $(\alpha+k\beta)$ is the maximum degree threshold. Any degree greater than $(\alpha+k\beta)$ is mapped to the final bucket.

Figure 3.5 shows an example of LBOH encoding. Assuming $\alpha = 4$, $\beta = 3$, and $k = 2$, the

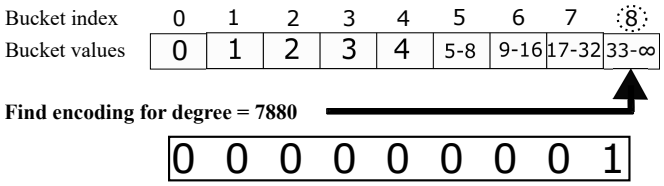


Figure 3.6: An example of finding the exponential binned one-hot degree encoding for a node with degree = 7880, where the parameters for the encoding scheme are $\alpha = 2$ and $k = 3$. Degree 7880 is mapped to its associated bucket (33 to ∞), then the bucket index (8) is represented using one-hot encoding (1 at the position where the value 8 exists, 0 otherwise).

following degree mapping is implemented: buckets 0 through 4 provide a one-to-one mapping of degrees 0 through 4. Degrees 5 through 4+3 = 7 map to bucket 5, and degrees 7 through 4+6=10 map to bucket 6. To find the encoding for a given input degree, the bucket index where this degree maps to needs to be found first. For example, if the input degree is 6, then the bucket index is 5. Finally, we represent that index using one hot encoding as 0000010.

This new representation addresses the shortcomings of the vanilla one-hot degree encoding as it provides a fixed number of features (buckets) regardless of the maximum degree in the training dataset. At the inference stage, only the values of α, β, and k are needed in order to prepare any graphs. However, one potential area of improvement is further reducing the number of features (buckets) in order to reduce the storage requirements and accelerate training. As opposed to One-Hot Encoding, LBOH provides a fixed number of features regardless of the maximum degree in the training dataset. At the inference stage, only the values of α, β, and k are needed.

Exponential Binned One-Hot Degree Encoding (EBOH) The main difference between EBOH and LBOH is the kind of step between buckets ranges. Instead of a linear step in LBOH, EBOH uses an exponential step to cover more degree values with a small number of features. First, the value of α is chosen such that $1 \leq \alpha \leq 3$. Then for the buckets, a sequential one-to-one mapping is performed for values 0 through 2^{α}. For the following buckets, the upper bound (inclusive) is $2^{\alpha+i}$ where $i \in [1, k]$ and $k \in \mathbb{N}^*$.

Figure 3.6 shows an example of EBOH encoding: given $\alpha = 2$, and $k = 3$, the bucket upper bounds will be as follows: $0, 1, 2, 3, 4, 8, 16$, and 32. To find the exponential binned

one-hot encoding for an input degree of 7880, first the bucket for the value 7880 is found (33 to ∞), and then the index of this bucket is found (8), then a one-hot encoding for this value is represented as 000000001. The main benefit of this encoding scheme over the scheme discussed in Section 3.3.3 is that it can represent a wider range of degree values, with a smaller number of buckets (features). Exponential binned one-hot degree encoding still provides the benefit of having the number of features independent of the maximum degree in the input training set. We use this scheme as the main approach to represent node features in our framework.

3.3.4 The Graph Neural Network Architecture

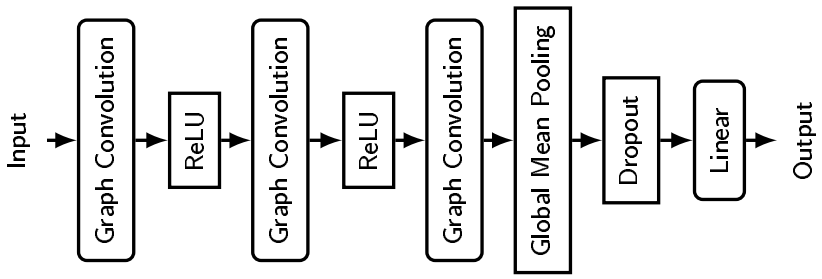


Figure 3.7: Graph Neural Network architecture.

To identify the structure of the input matrix, the matrix is viewed as the adjacency list of a graph, enabling the use of machine learning methods designed for graph data. GNNs provide additional benefits such as allowing the use of matrices (graphs) of arbitrary sizes as input, Also, GNNs are agnostic to node ordering. This powerful property enables re-labelling or permuting nodes in a graph representing a sparse matrix, while maintaining accurate predictions. The machine learning task of interest is graph-level prediction since a single label (class) is needed for an entire graph (matrix). The GNN architecture is shown in Figure 3.7. The hidden layers are three graph convolution layers [49] and one linear (output) layer. Graph convolution is an operation where node embeddings are iteratively generated as the aggregations from the node neighborhoods. This operation is used to capture complex features of the graph. The first convolution layer aggregates information from the local

neighborhood of each node. This operation is repeated in subsequent convolution layers in order to propagate information to increasingly larger neighborhoods. By the end of three convolution layers, the model has learned a hierarchical representation of the graph, where the features at each layer capture increasingly complex structural patterns. The learned representation so far is "node embeddings". Then, learned node embeddings are reduced into a single graph embedding using a global mean pooling operation (called readout layer). Samples are randomly dropped out to reduce overfitting. Finally, a linear classifier is applied to the graph embedding.

```
def generateDiagRandom(size, threshold=2):
    """ A function to generate a Diag+Random square matrix """
    tuples = [(x,y) for x in range(size) for y in range(size) if (
        random.randint(0,10) <= threshold or x == y)]
    # seperate tuples into two lists: the row array and the column
        array
    coo_rep = list(map(list, zip(*tuples)))
    return coo_rep[0], coo_rep[1], [size, size]
```

```
def process(self):
    catMap = [...., {
          # Number of instances to generate for this class
          'num_iter':10000,
          # Name of the generator function
          'generator':generateDiagRandom,
          # A string list of required generator function param
          'gen_params':['random.randint(MIN_DIM_SIZE,MAX_DIM_SIZE)'] }]
```

Figure 3.8: The two steps needed to add a new class to the classifier framework. First (top), create a new generator function in the generators file, and second (bottom), add a dictionary entry to `catMap` list in the `process` method of the dataset class.

Modularity New shapes of matrices can be easily integrated in our framework. To achieve this, two steps are needed as shown in Figure 3.8: (1) write a generator for that new shape, and (2) add an entry to the categories (shapes) map in the dataset class for this shape, containing the number of dataset instances to generate, the name of the generator function, and the different required parameters. The generator is required to return the COO representation excluding values, and the matrix dimensions. After generating the new dataset instances for this class, one does not need to re-train the entire model again. Transfer learning [92] can be used to replace the last layer of the trained model with a new layer that has

the appropriate number of outputs, after introducing the new shape(s). Then, the weights of all previous layers are frozen and only the new layer is trained. Another aspect of modularity in our framework is the ability to seamlessly attach different feature sets. Feature sets are only computed when the graph is queried. To implement a new feature set, a modification to the `get` method of the dataset is needed. This method first reads in the graph file from disk, calculates the new feature set, and attaches it to the graph.

3.4 Analysis

We run a set of experiments to evaluate the accuracy of our approach in detecting shapes, and compare it against the numbers reported by similar existing approaches. We also evaluate the impact of different feature representation discussed in Section 3.3.3 on accuracy.

3.4.1 Evaluation

Experimental Setup Table 3.1 shows the experimental setup and learning parameters used in the experiments. We use PyTorch Geometric [28] for the GNN.

Table 3.1: Experimental setup and Training parameters used in the experiments. * Batch size used for traditional one-hot encoding is 1.

Component	Specification	Parameter	Value
GPU	NVIDIA RTX A6000	Optimizer	Adam
GPU Memory	48 GB GDDR6	Learning Rate	0.01
CUDA Version	11.8	Error Criterion	Cross Entropy
Main Memory	64 GB DDR4	Batch Size	256*
Operating System	Ubuntu 22.04	Cross Validation Folds	5

Evaluation Metrics Prediction accuracy of the classifier is the main evaluation metric for our approach; since the goal of this work is to implement a classifier that can accurately detect the shapes of input sparse matrices. For accuracy, we use four derived metrics: accuracy, precision, recall, and F1 score. Accuracy simply represents the ratio of the number of correctly predicted instanced to the total instances tested. Precision is the ratio of the

number of correctly predicted instances for that shape (True Positives) to the sum of the number of correctly predicted instances and incorrectly classified instances as that shape (True Positives + False Positives); that is $precision = \frac{TP}{TP+FP}$. Recall is the ratio of the number of correctly predicted instances for that shape (True Positives) to the sum of the number of correctly predicted instances and incorrectly misclassified instances (True Positives + False Negatives); that is $recall = \frac{TP}{TP+FN}$. F1 score is the harmonic mean of precision and recall; F1 score= $\frac{2 \times precision \times recall}{precision+recall}$. We calculate all four metrics for the overall classifier, and also for each of the classes to determine how well the model performs for each individually. We report accuracy and F1 score numbers, since F1 score summarizes both precision and recall in one number. Accuracy gives an overall view of how well the classifier is performing, while the F1-score provides insights into its ability to correctly classify positive instances.

Dataset

We generate a dataset of 40K synthetic matrices for each of the sample classes: diagonal, random, random+diagonal, and Kronecker graphs using the generators discussed in Section 3.3.1. The dataset is balanced; the number of instances in each class is the same (10K matrices). The number of non-zeros in each of the generated matrices ranges between 1 and 21386821. Figure 3.9 shows the distribution of the number of non-zero values in our synthetic dataset.

Cross Validation

We use k-fold cross validation to train our model, with $k = 5$, in order to estimate the performance of our model.

as shown in Figure 3.10. We split the dataset into two subsets: a training and a validation set. It is split into five equally-sized subsets, called "folds". The model is then trained and evaluated five times, with each of the folds used once as the validation set, and the remaining four folds used as the training set. This process is repeated until each fold has been used as

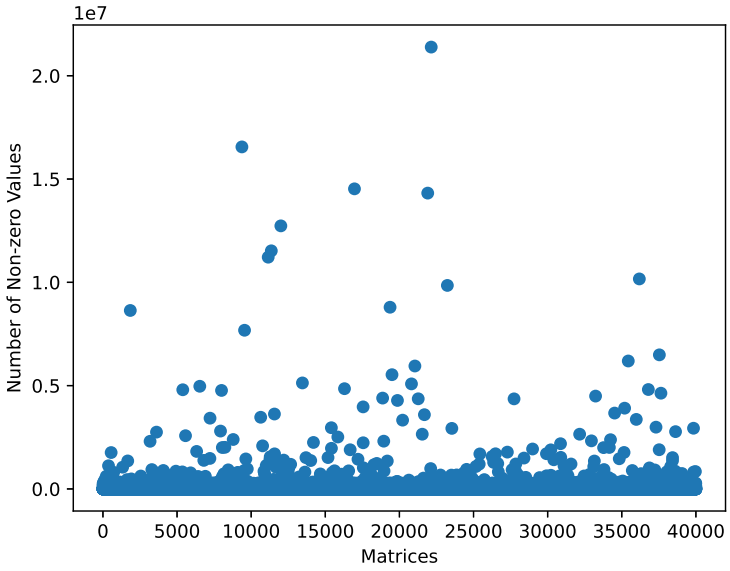


Figure 3.9: Distribution of number of non-zero values (vertical axis) across the 40000 matrices (horizontal axis) in our synthetic dataset.

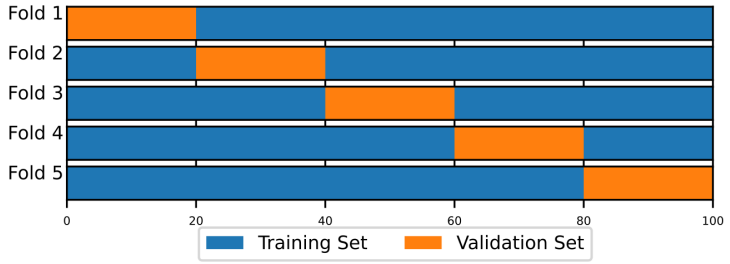


Figure 3.10: 5-fold Cross Validation used in estimating the accuracy of the model. For each fold, the dataset is split into two subsets: training and validation. Different partitions of the dataset are assigned to each of them.

the validation set once. Before splitting using k-fold cross validation, we randomly shuffle the indices of the dataset, to make sure that data is randomly distributed across folds and to prevent any bias that might be introduced if the data is sorted by label (class).

K-fold cross validation allows for a more accurate estimation of the model's performance than a single train/test split. By training and testing he model on different subsets of the data, one can get a better understanding of how the model will perform on new, unseen data. After the process, the evaluation metrics (discussed in Section 3.4.1) are averaged across the different folds to provide an estimate of the model's performance.

Training Method and Parameters

We use k-fold cross validation to train our model, with $k = 5$, in order to accurately estimate the performance of our model. For the optimizer, we use Adam [48] with a learning rate of 0.01. We used cross entropy loss as the error criterion. The model is trained on batches of input data, where a batch contains multiple graphs. We used batch sizes of 256 for LDP, LBOH, and EBOH, and a batch size of 1 for One-Hot Encoding since it was the only possible batch size that fits in GPU memory using this encoding. Batch sizes vary by experiment based on the degree encoding used for that specific experiment. The encoding scheme used affects the size of the graph, and therefore affects the number of graphs per batch that can fit at once in the GPU memory for training. In our experiments, batch sizes varied from 1 in case of using traditional one-hot degree encoding, to 256 when using LDP and EBOH.

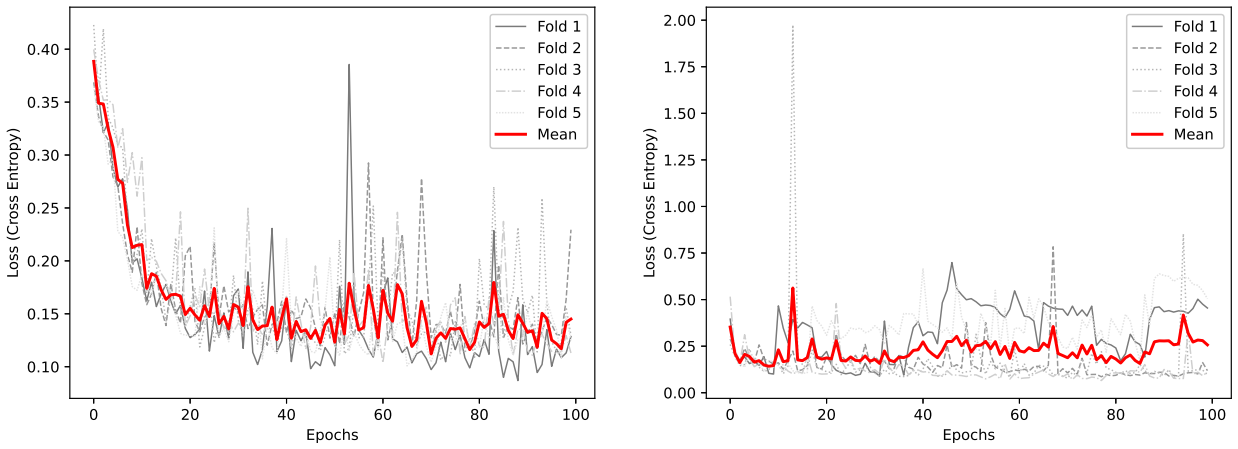
3.4.2 Results

Table 3.2: Performance of the classifier for different degree representations

	Degree Representation							
	One-Hot Encoding		LDP		LBOH		EBOH	
	Accuracy	F1 Score	Accuracy	F1 Score	Accuracy	F1 Score	Accuracy	F1 Score
Diagonal	1.0	0.90	1.0	0.97	1.0	1.0	1.0	1.0
Random	0.90	0.91	0.64	0.76	0.92	0.95	0.95	0.96
Random+Diagonal	0.86	0.99	0.98	0.83	0.96	0.94	0.97	0.96
Kronecker	0.90	0.94	0.90	0.93	0.98	0.97	0.96	0.98
Overall	0.90	0.90	0.88	0.88	0.97	0.97	0.97	0.98

Classification Performance Table 3.2 shows the accuracy and F1 score for the classifier using the different degree representations discussed in Section 3.3.3. Performance results show that both LBOH, and EBOH provide high prediction accuracy of around 97% and a F1 score of around 98%. On the other hand, traditional one-hot encoding exhibits a lower accuracy of around 90%. One-Hot Encoding requires a significantly large number of features per node (7711), limiting the training batch size on the A6000 GPU to only one graph. This forces the optimizer to adjust the neural network weights very frequently, hence

hurting the overall accuracy. Using LDP as a feature set exhibits variant model performance across folds depending on the validation set being used. In some folds, LDP provides high accuracy of around 97% to 98% similar to EBOH. In other folds, LDP fails to converge to an acceptable loss value, and ends up with an accuracy of around 74% on the last few epochs. This performance variance across folds deems LDP unfit for the purposes of our application. It significantly fails in two classes: Random and Kronecker. It predicts Random matrices as Random+Diagonal for more than 32.5% of the instances. This is likely due to the prevalence of the local degree neighbor summary features (the last four LDP features) instead of focusing on the node degree. This eventually results in failing to discover the global hierarchical structures in the matrix. LDP still shows perfect accuracy in case of diagonal matrices since almost all nodes in the matrix's graph have the same degree. LDP prediction quality for Kronecker graphs is also lower than other evaluated feature sets (around 81% in some folds) for the same reasons.



(a) (b)

Figure 3.11: Cross Entropy Loss across different folds in 5-fold cross validation training using (a) EBOH, and (b) LDP feature set.

Figure 3.11a demonstrates the validation loss across the 5 different folds for EBOH. It shows almost no variance in the loss across the different folds, indicating the stability of the model's performance across folds. On the other hand, Figure 3.11b shows the validation loss for LDP and illustrates that the loss does not converge in 2 out of 5 folds.

Training Time

While training is a one-time overhead, we capture the training time spent for our model using different feature sets. For EBOH, training the model on the entire dataset took around 10 hours and 12 minutes. For LBOH, the training time was around 11 hours. LDP trains on the entire dataset in around 10 hours and 10 minutes. Traditional One-Hot Encoding on the other hand took around 37 hours to train.

The justification of the slight variance (or almost invariance) of training time even with different batch sizes being used for LDP, LBOH, and EBOH, is the way the feature set is being attached to graph nodes. Graphs are not stored on disk with node features. Instead, node features are calculated every time the graph is fetched during the training process. This incurs additional memory and computation overhead, however it also provides the flexibility needed to easily experiment with different feature sets. Since training overhead is a one-time cost, we decided to keep the lazy feature calculation method as the flexibility benefit over-weighs the single time memory and compute overhead.

Classifying Sub-samples and Re-labelled Subgraphs The choice of graph neural networks for our framework accounts to the attractive properties of such networks, such as being agnostic to both the size and the arrangement of input data. To test the efficacy of GNN on both aspects, we generate 200 new matrices: 50 for each of the four classes. For each of them, we generate 10 subgraphs and 10 re-labelled variants. To generate the subgraphs, we use uniform random node sampling (URNS) [57]. URNS works as follows: for a given graph $G = \{V, E\}$ where $V = \{v1, v2, .., vn\}$ is the set of nodes, and $E = \{e1, e2, ..., en\}$ is the set of edges, an URNS subgraph G' consists of nodes that are randomly selected with uniform probability, as well as the edges connecting the selected nodes. Re-labelling of a graph G simply renames (re-orders) the nodes V of the graph, and produces a new graph G' with the same size and degree distribution of the original graph G. Figure 3.12 shows an example of both URNS and random re-labelling.

Table 3.3 shows the model's performance on subgraphs and re-labelled variants as com-

Table 3.3: Accuracy comparison for node sampling, node re-labelling, and original graphs using EBOH feature set.

Class	Node Sampling	Node Re-labelling	Original Graphs
Diagonal	1	1	1
Random	0.83	0.98	0.98
Random+Diagonal	0.92	0.92	0.92
Kronecker	0.94	0.96	0.96
Overall	**0.92**	**0.97**	**0.97**

pared to original full graphs. The table shows that re-labelling node has no impact on the classification accuracy; it shows the same overall accuracy of around 97% which is observed for the original graphs. This is expected because the arrangement of nodes in a graph is irrelevant, since the graph has the same degree distribution. For subgraphs generated using URNS of larger graphs, the overall accuracy drops to around 92%. The reason being that random node sampling can alter the degree distribution of the graph. The random choice of nodes can result in either isolated nodes (no edges) or much lower degree nodes as compared to the original graph. This affects the accuracy specially for complex shapes such as random and Kronecker graphs. One way to reduce the accuracy loss for samples is to use a more sophisticated graph sampling technique rather than randomly selecting nodes or edges.

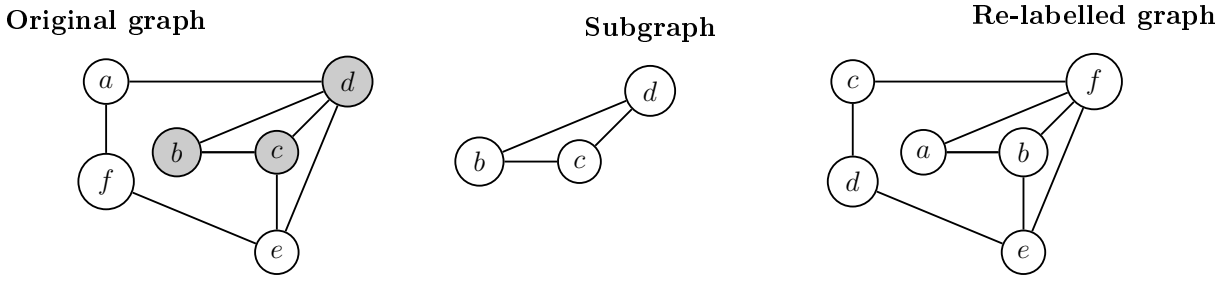


Figure 3.12: Example of generating a random sub-sample and a re-labelled variant of a graph. The original graph (left) contains six nodes. Using URNS, a random subgraph (middle) of three nodes is generated. A re-labelled variant (right) is generated using a random 1:1 mapping between the original and new node labels.

3.5 Summary

In this chapter, we proposed a GNN based framework to classify structured sparse matrices. We introduced two novel non-attribute graph representations based on node degrees: LBOH, and EBOH. We evaluated the efficacy of our framework on a synthetic, balanced dataset of matrices that we generated containing random matrices from four sample classes: diagonal, random, random+diagonal, and Kronecker graphs. Performance results demonstrate a high classification accuracy of 97% for the framework when using our feature sets: LBOH and EBOH. They also show high accuracy of 92% and 97% on random node subsamples and relabelled variants respectively. Our framework is modular, allowing the inclusion of additional classes with minimal user effort. Future endeavors target the automatic generation of the optimal sparse data format and algorithm for sparse matrix kernels, using the obtained prediction results from the current framework.

Chapter 4

A Framework for Analyzing the Robustness of Graph Models

4.1 Introduction

Graph models are incredibly important for generating large scale synthetic data when real data is not accessible, and when graph operations need to be evaluated for performance. Thus, generated data gives developers a proxy dataset to refine their code. The closer the model approximates the real data, the more likely developers are to produce code that is efficient for those real datasets. One question is how much of a predictive understanding of the global structures of the graph can these models provide, and are these structures robust enough, i.e. not highly sensitive to noise, that a developer can optimize for them?

To this end, we propose a novel framework to evaluate graph descriptors. Our framework takes an input vector of graph generation parameters P and produces a set of graphs $G_s = \{G_1, G_2, G_3, ..., G_n\}$. It then evaluates the effect of varying P values on the structure of the output graph set G_s through observing the change in different properties distributions such as: degree, in-degree, out-degree, in-betweenness centrality, clustering coefficient, etc. To ensure the robustness of the parameter set P in describing the graph structure, our

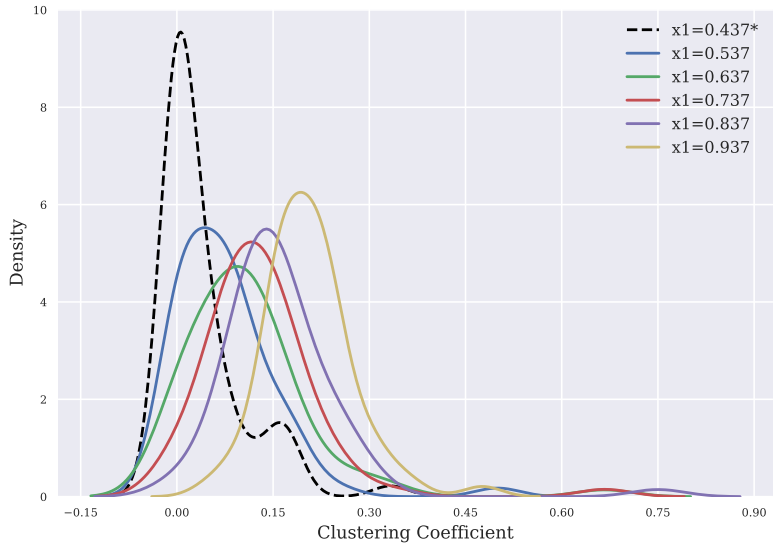


Figure 4.1: An example of an output plot produced by our graph analysis framework. The plot shows the effect of changing a graph generation parameter x_1 on the node clustering coefficient distribution of the graph. The dashed line indicates the distribution for the initial graph G_0

framework also evaluates the effect of adding gradual noise to the parameter set and observes the noise threshold *alpha* at which the structure of the graph is dominated by the induced noise.

Figure 4.1 shows an example output plot from our framework. This kernel density estimation (KDE) plot evaluates the effect of varying a graph generation parameter $x_1 \in P$ with an incremental step $\alpha = 0.1$ on the graph clustering coefficient distribution, as one of the graph structural properties. The original graph distribution is represented by the dashed black line. As discussed later, x_1 is an initiator matrix value of a Kronecker graph.

The main contributions of this work are:

1. Proposing a novel framework to evaluate graph descriptors and how sensitive graph structures are to them.

2. Evaluating the tolerance of the graph descriptor to random noise.

3. Demonstrating the usage of the proposed framework through a case study on the Kronecker Graph model.

4.2 Background and Related Work

4.2.1 Generative Graph Models

Various works presented different models to generate synthetic graphs [26, 4, 35, 37, 87, 68, 101], that are similar in terms of graph properties [16] to real world graph. The motivation behind the introduction of such models is, but not limited to: (1)understanding complex structures of large graph using smaller, well-formulated synthetic models, (2)tackling the privacy restrictions associated with accessing and studying real graphs, (3)predicting the evolution of large scale graphs, and (4) benchmarking graph neural networks (GNNs) [79]. In this chapter, we focus on the Kronecker graph model as one example of such generative models.

4.2.2 Kronecker Graphs

Kronecker graphs [56] are a class of synthetic graphs that have been widely used to model real-world networks and are generated by recursively applying the Kronecker product of a small base graph with itself. Let A and B be two matrices. Then, their Kronecker product $A \otimes B$ is given by

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix} \tag{4.1}$$

where a_{ij} are the entries of A. The resulting graph has a power-law degree distribution and exhibits a hierarchical structure that captures both the local and global connectivity patterns of the underlying real-world network.

To generate Kronecker graphs, an initiator matrix (typically of size 2×2 or 3×3) is chosen, and the Kronecker product is applied to this matrix by itself K times, where K is the Kronecker power. Then, a randomly generated probability matrix is used to mask out

random values in the Kronecker matrix (remove edges from the Kronecker graph). Other compute-efficient methods can be used to generate Kronecker graphs such as ball dropping and grass hopping [77].

In [56, 58, 46], the authors present and analyze the Kronecker graph model as an approximation of scale-free graphs. Their analysis served as a template for ours, however in their work they compared their model against the real graphs that they were approximating, whereas as we analyze varying synthetic graphs against each other. In addition to Kronecker Graphs, many other graph models have been studied and surveyed [102, 24, 34].

In [83] the authors demonstrate that the Kronecker Graph model does not produce power law graphs, but that through the addition of noise the resulting model becomes a much stronger representation of power law graphs.

4.2.3 Graph Learning and Structure Prediction

In [3], we present an extensible framework for classifying sparse matrices structures using graph neural networks. The framework classifies an input matrix (graph), into one of a pre-defined classes of structure such as mesh network, random matrix, Kronecker graph, and combination of them. Additional classes can be integrated in the framework. Our proposed framework can be used in tandem with the classifier framework by evaluating more graph models (classes of structure), and then augmenting the classifier framework with additional structures that show a clear correlation between the input generation parameter set P and the output graph structure, and robustness to injected noise.

In addition, our framework can be used to evaluate graph generation parameters that can be then used in the feature set selection step of training graph neural networks for non-attributed graphs [15], given that these parameters prove to be robust representing the graph structure.

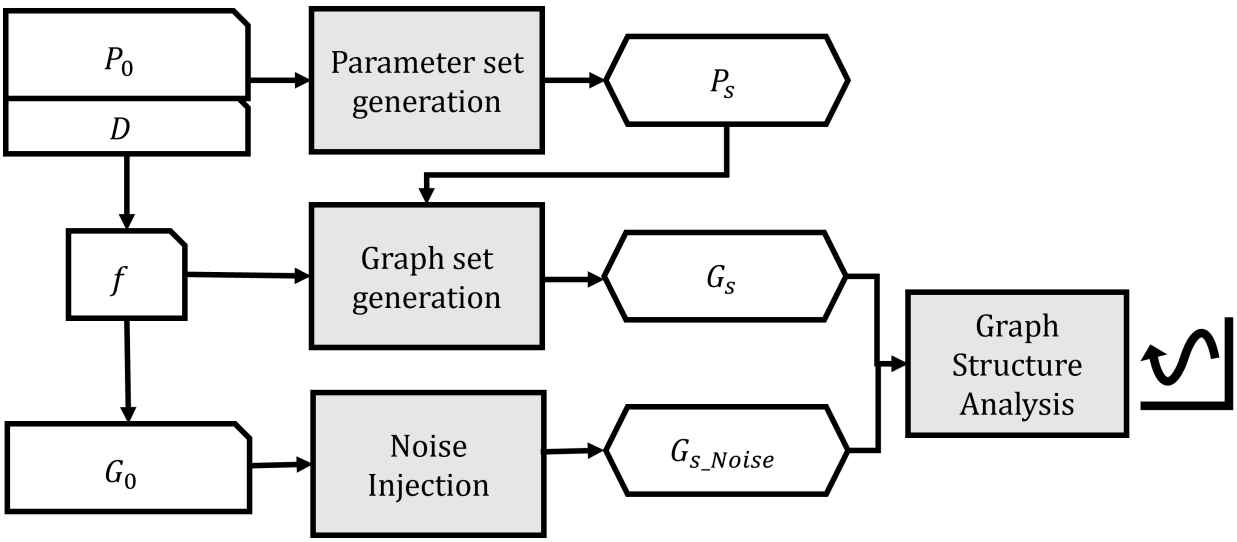


Figure 4.2: High-level overview of the proposed framework. It takes initial generation parameters P_0 alongside with the domain of legal values D and the generation function f to generate a new parameters set P_s which is used to generate a new graph set G_s. Noise is also injected after generating the initial graph G_0 to synthesize the noisy graph set G_{s_Noise}. Finally, an analysis is performed on the output graph sets structure.

4.3 Framework Overview

Our proposed framework can be visualized as shown in Figure 4.2. It takes as input a graph model (M) generator f and a set of initial parameters vector $P_0 = \{p_{00}, p_{01}, ..., p_{0m-1}\}$ such that one can generate a graph G_0 of model M as follows: $G_0 = f(P_0)$. In addition, it expects the domain of legal values for each of the parameters $D = \{d_0, d_1, ..., d_{m-1}\}$. Then, the framework generates a set of graphs G_s by varying each of the parameters in P_0 individually within D for n times, while fixing the rest of the parameters, generating a new set of parameters $P_s = \{P_1, P_2, P_3, ..., P_n\}$. With the new set of parameters, a new set of graphs G_s is generated using f such that $G_i = f(P_i)$ for $1 \leq i \leq n$.

For each of the generated graphs, the framework calculates and plots multiple graph structural properties such as degree centrality, in-degree, out-degree, betweenness centrality, closeness centrality, Laplacian centrality, clustering coefficient, and singular values. The output vectors and figures enable the evaluation of the effect of changing the parameter set P_s on the structure of the original graph G_0.

In addition, the framework evaluates how graph structure reacts to injecting random noise, and the extent of noise to which it maintains its original underlying structure. This is done by adding random edges, represented as a random sparse matrix (adjacency matrix) with density *alpha*. By varying *alpha*, different noise matrices are generated and are added to the original graph's (G_0) adjacency matrix. The same graph properties are calculated and plotted as output.

Such analysis enables the evaluation of the graph model M and the parameter set P. One can determine whether this model M is a representative model of a set of classes that share specific structural properties, that can be later exploited to design the algorithms and data structures for this class of graph, or this model does not introduce any exploitable characteristics (i.e. random).

Moreover, the analysis can predict whether the structure of the generated graphs is sensitive to the input parameter set. Hence, a decision can be made to use the input parameter set as a representative feature set for this class of graph. The choice of adequate representative set for graph is critical in many applications such as compact graph representation for large graphs (compressing large graphs by only storing a small set of parameters that can be used to generate such graph on demand), and learning on graphs: choosing a feature set for (non-attributed) graphs in a graph neural network setting.

Noise analysis is crucial as well, since most of the real data (graphs) are expected to include a certain level of entropy. Gradual noise injection and observing the resulting graph structure permits a good estimate of a noise tolerance, where the original graph model maintains its structure before, and loses its special characteristics beyond. It is also useful in evaluating the graph generation parameter set: if adding noise with higher values does not affect the graph structure, this means that the initial graph structure was already random and the used parameter set and/or graph model cannot be effectively used to represent a meaningful unique class of graphs.

4.4 Case Study: Analyzing Kronecker Graphs

In order to evaluate our framework, we analyze the stochastic Kronecker Graph Model. The generator f is the Kronecker graph generator. It takes the following parameters P : the Kronecker initiator matrix X and the Kronecker power K. For simplicity, we only consider X in this discussion.

X is assumed to be a 2×2 matrix and consists of the following values:

$$\begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix}$$

Hence, each parameter vector can be represented as $P_i = \{x_{i0}, x_{i1}, x_{i2}, x_{i3}\}$. The domain D for the each $x_{ij} \in P_i$ is floating point values between 0 and 1.

To evaluate the correlation between each of the initiator matrix values and the resulting Kronecker graph structure, we vary each of them within D, while fixing the other three values. We observe the effect of this variation on different graph and node-level attributes such as: degree, in-degree, out-degree, betweenness centrality, etc. The initial parameter vector P_0 is based on Kronfit's [58] estimated Kronecker initiator values [56] of the High Energy Physics - Phenomenology Collaboration (CA-HEP-PH) Graph [59]. Starting with P_0, we generate graphs of the 6th Kronecker Power ($K = 6$) for each of initiator matrices we evaluate $P_s = \{P_1, P_2, ..., P_n\}$.

For example, to evaluate the effect of changing x_0, we fix the values of x_1, x_2, and x_3. We then vary the value of x_0 for m times (5 in our experiments), by subtracting α (0.1) each time. So, for the first time, the new initiator matrix values will be as follows:

$$\begin{bmatrix} x_0 - \alpha & x_1 \\ x_2 & x_3 \end{bmatrix}$$

From this new initiator matrix, we generate a Kronecker Graph of Kronecker power 6. Then,

we analyze the different graph and node-level attributes detailed in the following subsections.

For distributions, our framework generates Kernel Density Estimation plots, where the horizontal axis represents the property distribution (degree, betweenness centrality, etc.), and the vertical axis represents the density of the distribution at each point. A black dashed line the Figures 4.3, 4.4, 4.5 represents the corresponding distribution for the initial parameter set P_0, which is the starting point from which the framework begins to vary the parameter set values.

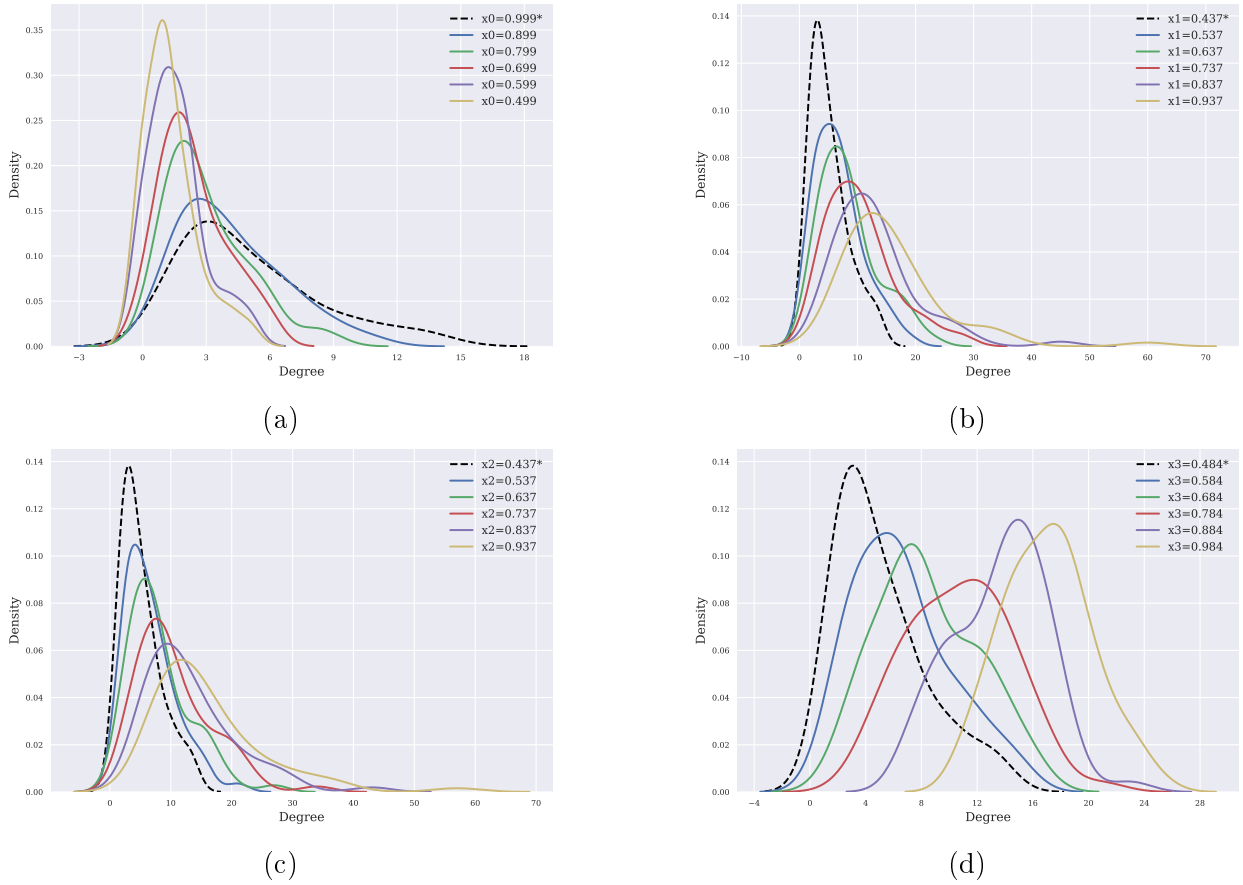


Figure 4.3: The effect of changing (a)x_0 , (b)x_1 , (c)x_2, and (d)x_3 on the degree distribution of the resulting $K6$ Kronecker Graph. Each of the sub-figure is a KDE plot where the degree distribution are on the horizontal axis, and the density is on the vertical axis.

Figure 4.3 shows the effect of varying the different initiator matrix values on the degree distribution of the resulting Kronecker graph. To further investigate the structural effects of changing the different initiator matrix values, we additionally study the in-degree and

out-degree behavior. Figure 4.4 shows a similar analysis for in-degree distribution. Figure 4.5 shows the analysis for out-degree distribution.

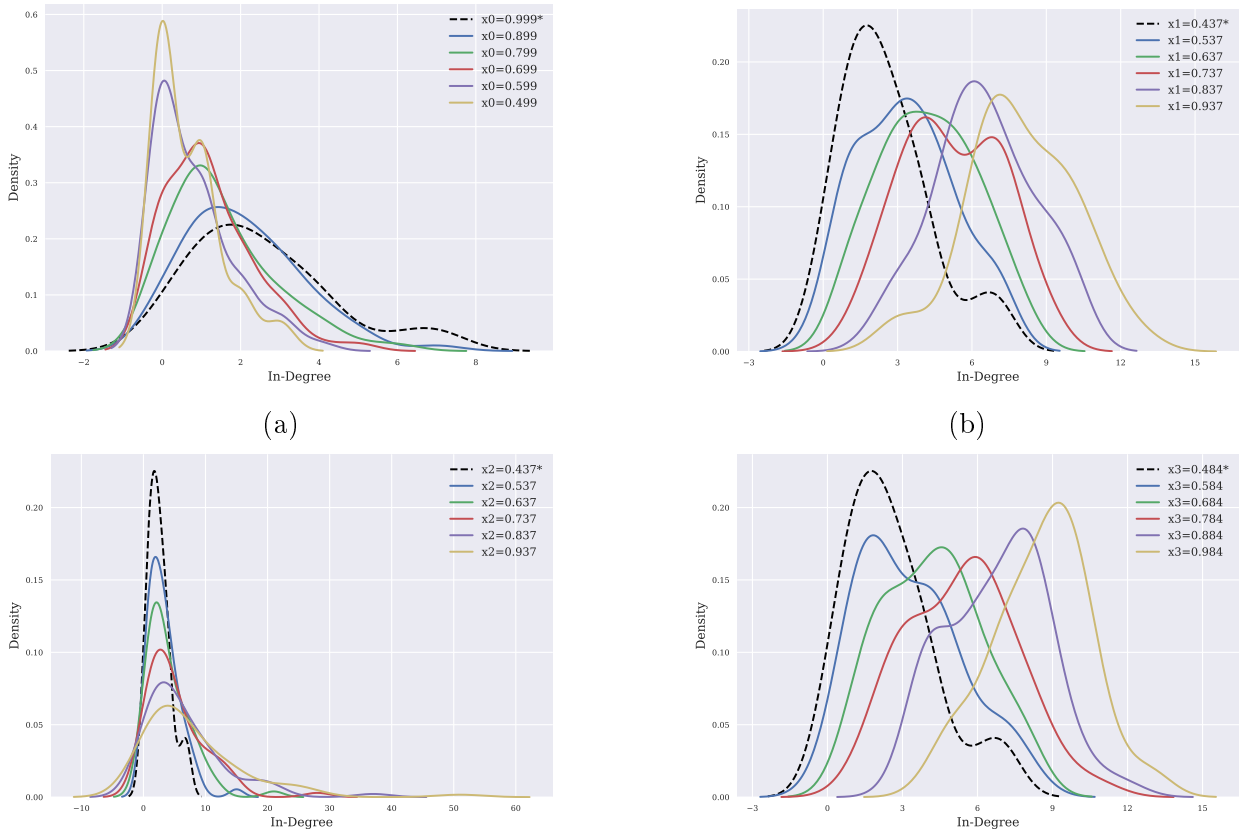


Figure 4.4: The effect of changing (a)x_0 , (b)x_1 , (c)x_2, and (d)x_3 on the in-degree distribution of the resulting $K6$ Kronecker Graph. Each of the sub-figure is a KDE plot where the degree distribution are on the horizontal axis, and the density is on the vertical axis.

For noise analysis in Figure 4.6, we evaluate the effect of varying random noise (varying the random noise sparse matrix density: alpha) on: (a) Degree, (b) In-Degree, (c) Out Degree, (d) Betweenness Centrality, (e) Closeness Centrality, (f) Laplacian Centrality, (g) Clustering Coefficient, and (h) Scree Plot (singular values). The dashed black lines represent the distributions for the original initial graph G_0 with no noise injected.

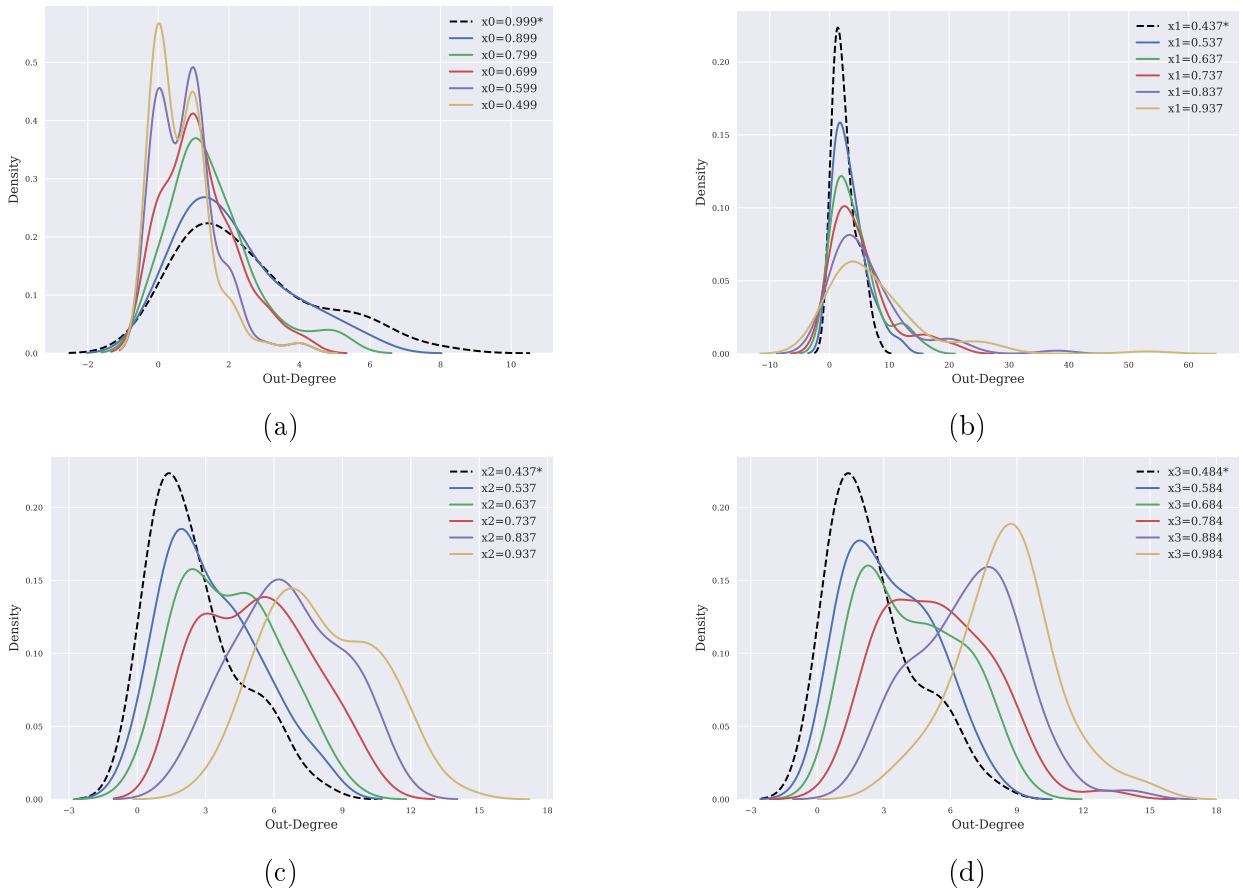


Figure 4.5: The effect of changing (a)x_0 , (b)x_1 , (c)x_2, and (d)x_3 on the out-degree distribution of the resulting $K6$ Kronecker Graph. Each of the sub-figure is a KDE plot where the degree distribution are on the horizontal axis, and the density is on the vertical axis.

4.4.1 Effect of Varying Kronecker Initiator Values

varying x_0

For the degree distribution, Figure 4.3a shows that increasing x_0 value results in increasing the maximum degree across nodes, and decreasing it decreases the maximum degree. It is also seen that the minimum degree is not affected due to x_0 variation. This means that increasing the value of x_0 flattens the degree distribution of the graph, where the peak density (or frequency) of nodes with low degree is decreased (as compared to lower x_0 values) and the peak of the distribution gradually shifts towards the right bottom. This behavior is consistent across the different values we tested for x_0 within the legal range.

In-degree distribution exhibits a similar behavior to the overall degree distribution as shown in Figure 4.4a: maximum degree increases with increasing the value of x_0, and the minimum degree is still unaffected, which squishes the peak density of the lower degrees into a flatter curve with lower peaks as compared to smaller x_0 values.

As for out-degree, increasing x_0 still increases the maximum out-degree. However, x_0 contribution to increase of out-degree is limited compared to in-degree as illustrated in Figure 4.5a.

Varying x_1

For the degree distribution, Figure 4.3b shows the effect of varying x_1. A similar trend to x_0's is observed. However, increasing x_1 rapidly increases the maximum degree to a higher value.

On the other hand, changing x_1 shows a different trend of increasing both maximum and minimum ***In-degree*** as shown in Figure 4.4b, even though the increase in the minimum degree is slower. Also, the maximum in-degree for the maximum x_1 value is significantly lower than that of the overall degree. So, most of the degree increase accounts to the out-degree increase and not the in-degree. An extended range of maximum degree increase proves

this behavior in Figure 4.5b.

Varying x_2

The effect of varying x_2 on the resulting graph degree is similar to that of x_1 as shown in Figure 4.3c, and with similar values for maximum degree.

However, the opposite is true for ***In-Degree and Out-Degree***. Figure 4.4c illustrates that in-degree increase dominates the majority of the degree increase. The peak density still decreases with increasing x_2, but the range of in-degrees that fall within the peak density is narrower than what is observed for x_1.

x_2 changes the out-degree distribution in smaller steps than it does for the in-degree distribution as illustrated by 4.5c.

Varying x_3

Figure 4.3d demonstrates the effect of varying x_3 on the ***overall degree*** distribution, where increasing x_3 value affects both the minimum and maximum degree across nodes. Smaller x_3 values have both smaller minimum and maximum degree values as compared to higher x_3 values. Changing x_3 values shifts the degree distribution to the right or the left (change min and max degree), but has a smaller effect on the degree density across nodes (peak density/frequency is slightly affected), which is different from the effect of changing x_0, x_1 , and x_2 on the degree density as described before.

Similarly, for ***In-Degree*** both tails of the density distribution are extended without a significant change in the peak density as exhibited in Figure 4.4d. x_3 variation has an almost identical effect on both in-degree and ***out-degree***.

General Observations

From the above analysis, it is obvious that certain features are more sensitive to certain parameters than others. For example: Figure 4.3 shows that degree is more sensitive to varying

x_3 than x_0. In-degree (Figure 4.4) is highly sensitive to changes in x_1, while out-degree is more reactive to changes of x_2. Also, as x_3 values increases (gets closer to x_0), degree distributions start to diverge quickly (the mean degree shifts significantly). Additionally, Figure 4.4b shows that x_1 change has the highest effect of in-degree, and Figure 4.5c demonstrates that x_2 has a significant impact on out-degree distribution.

4.4.2 Varying noise

To evaluate the robustness of the correlation between the graph descriptor (Kronecker initiator matrix values) and the graph structure, we inject random noise to observe how the structure reacts to noise. The noise injected is a random sparse matrix, that is added to the adjacency matrix of the generated Kronecker graph. Then, we vary the density (sparsity) of the random matrix and evaluate the effect of noise with different values for density (alpha).

Figures 4.6a, 4.6b, and 4.6c show that for lower values of alpha (up to around 2%), the resulting graph maintains a very similar degree distribution, compared to the original graph with no noise. Beyond that point, the random noise starts to take over until the Kronecker characteristics of the original graph are no longer recognizable.

Figure 4.6d shows a similar behavior for the betweenness centrality of the graphs. Random noise with alpha above 4% distorts the original graph betweenness centrality distribution.

Closeness centrality distribution is more sensitive to random noise as shown in Figure 4.6e, where the entire distribution shifts to the right, with the minimum closeness tail almost fixed. However, the pattern still persists where for injected noise with alpha beyond 4%, the distribution completely changes.

Laplacian centrality in Figure 4.6f is less sensitive to the noise as compared to closeness centrality, for lower values of alpha (up to 4%). With higher alpha, the distribution skews and is mainly random noise.

The Clustering coefficient by nature is highly sensitive to graph structure changes as

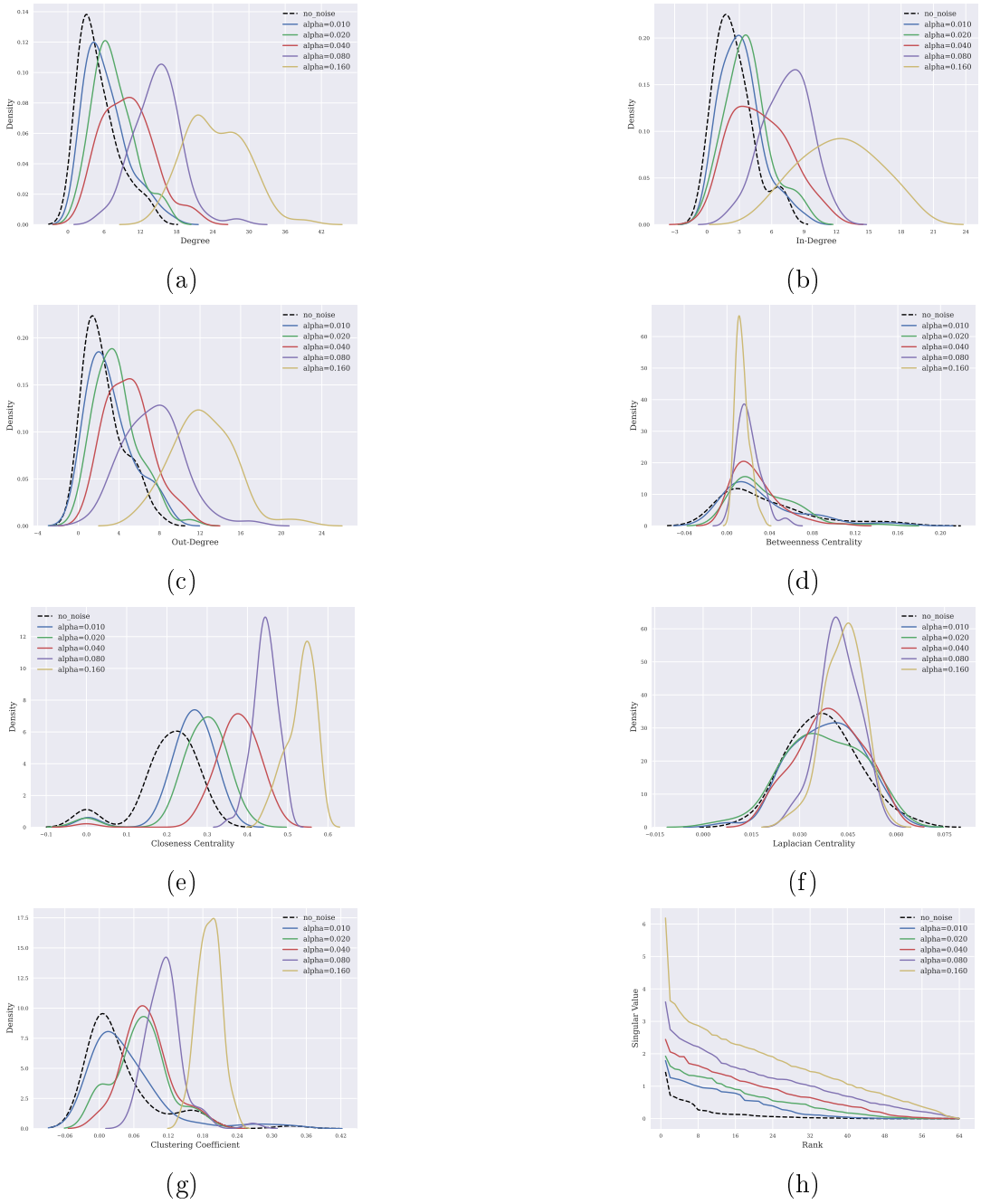


Figure 4.6: The effect of changing the injected sparse random noise matrix density on (a) degree, (b) in-degree, (c) out-degree, (d) betweenness centrality, (e) closeness centrality, (f) Laplacian centrality, and (d) scree plot of the resulting Graph. Each of the sub-figure is a KDE plot where the degree distribution are on the horizontal axis, and the density is on the vertical axis.

shown in Figure 4.6g. Any noise with alpha greater than 1% immediately disrupts the distribution. Such behavior is expected since the clustering coefficient captures the local connections in a graph, and how it propagates globally. When adding a few new edges (by injecting random noise with low alpha value), the local connectivity patterns are immediately affected: local clusters may be disrupted or new clusters may form, eventually leading to changes in the clustering coefficient values for individual nodes, and then the overall distribution. However, one can still notice the significant difference between inducing noise with lower alpha values (up to 4%) and inducing noise with higher alpha values.

The Scree plot of a graph captures the relative importance of components (nodes) in the graph. The importance of a component in a graph is mainly represented by the number of connections (degree) of the component. Inducing the random noise in our experiments simply adds new edges to the graph, so its relative effect on scree plot is not expected to be significant as shown in Figure 4.6h. However, adding noise with higher alpha values increases the (absolute) spectral gap between adjacent singular values.

4.5 Summary

In this chapter, we proposed a novel framework to evaluate graph descriptors. The framework takes as input the graph descriptor (generation parameters) and generation function, and evaluates the co-relation between the descriptor values and the underlying graph structure. It captures the sensitivity of the graph structure to the descriptor values, as well as to injected random noise. Changes in graph structure are detected by observing the change in different graph structural properties such as degree, in-degree, out-degree, betweenness centrality, closeness centrality, Laplacian centrality, clustering co-efficient, and singular values. We provided a case study of using the framework by analyzing the sensitivity of stochastic Kronecker graph structure to the initiator matrix values used to generate them, and to induced random noise (sparse random matrix) with varying density. The design of the

framework is modular so that it can evaluate different existing and future graph models, and additional graph properties can be easily calculated for evaluated graphs.

Chapter 5

A Framework for Analyzing the Performance of Graph Models

5.1 Introduction

Large, sparse, and irregular data is central in the domains such as graph analytics, graph neural networks, fluid mechanics, and finite element analysis. Specifically, if the dynamic relationship between elements in a dataset can be captured as an edge-pair relationship between vertices, then graphs provide a natural representation of that data. Furthermore, if the analysis of complex relationships in the data can be performed through sequential linear algebra-like operations over the adjacency matrices of these datasets, then operations such as Sparse Matrix times Vector Multiplication (SpMV) are critical to the performance of computations in these domains. However, optimizing operations like SpMV is challenging because the structure of the sparse data, the implementation of the operation, and the architecture of the target all have a tremendous bearing on the execution time of these operations. If a sparse operation is tuned for one class of data, that performance may not generalize to another class. The core of this work is to provide a benchmarking framework for correlating performance with the structural features of sparse data.

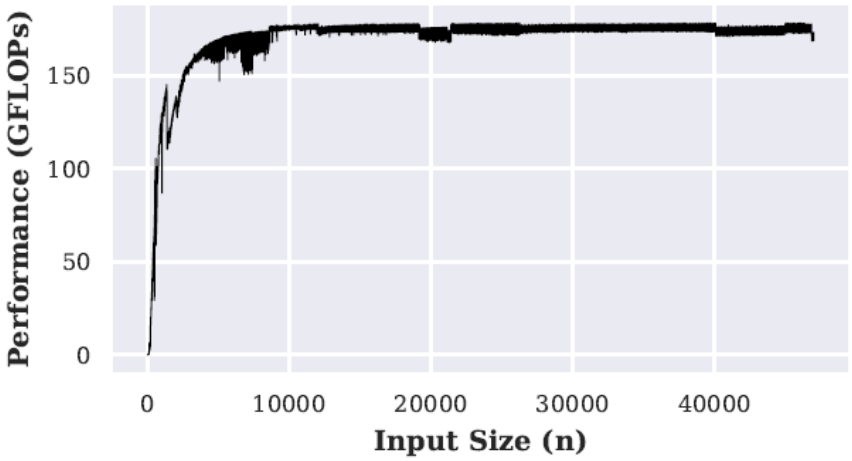


Figure 5.1: Double Precision Dense General Matrix-Vector Multiplication Performance using cuBLAS on NVIDIA RTX A6000 GPU as a function of input size

For dense matrices, performance evaluation and visualization is straightforward. Figure 5.1 shows an example of performance evaluation of the general matrix-vector multiplication (GEMV) on a RTX A6000 GPU using cuBLAS. For simplicity, evaluated matrices are assumed to be of square dimensions: $n \times n$. The horizontal axis represents the different values of n evaluated, and the vertical axis shows the performance in GFLOPs. Moving along the horizontal axis (from left to right and from right to left) shows a clear correlation between the matrix dimensions ($n \times n$) and GEMV performance. Performance interpolation from existing data points is possible, based on the dense matrix dimensions (n)

On the other hand, Figure 5.2 shows a corresponding performance evaluation for SpMV using cuSparse. Sample matrices from the SuiteSparse collection [21] are evaluated. Table 5.1 lists the properties of these matrices. The horizontal axis in Figure 5.2 represents different matrices, and the vertical axis shows the SpMV performance in GFLOPs. In this case, moving along the horizontal access does not provide any meaningful insights regarding how SpMV performance changes across different matrices. This is because different matrices have totally different characteristics and there is no correlation between them. Using dimension size on the x-axis (similar to the dense case) also does not help draw conclusions since: a)two matrices can be of the same dimensions, but have different sparsity ratios, and b)sparse ma-

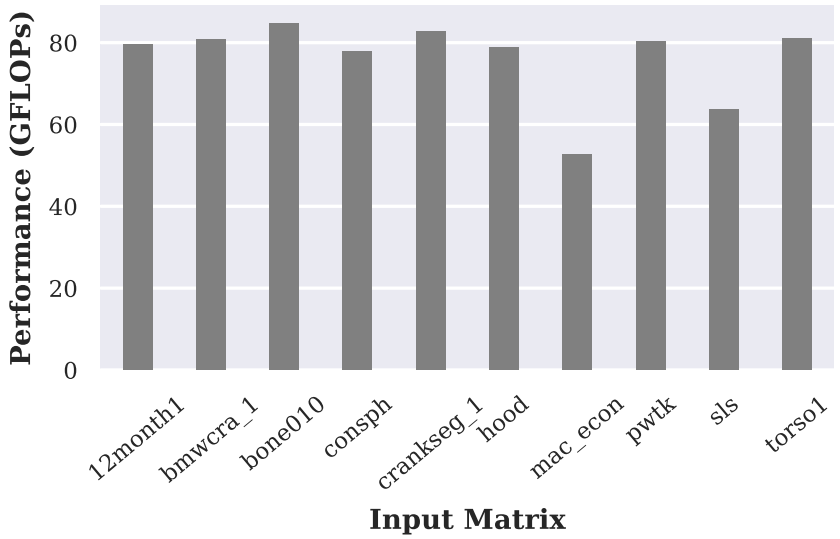


Figure 5.2: Double Precision Sparse Matrix-Vector Multiplication Performance using cuSparse on NVIDIA RTX A6000 GPU for a selected set of matrices from SuiteSparse, using the COOrdinate data representation.

trices from different applications can have extremely different dimensions. Another variable to use on the horizontal axis can be the number of non-zeros (NNZ). However, also NNZ does not always provide enough insight on the sparsity ratio or the dimensions of the sparse matrix. Two matrices with similar NNZ can have completely different dimensions and sparsity ratios. Moreover, while existing performance evaluation models work on providing multiple different performance metrics [9], face a set of limitations adapting to sparse data workloads with regard to using a representative feature/parameter being related to performance [93]. Also, the dependence on discrete sets of graphs/sparse matrices for benchmarking such operations [60, 21, 10] limits the ability to make performance interpolation and generalization across a more diverse set of input data. Additionally, many existing optimization techniques focus solely on the sparse operation to be optimized, and do not have a global breakdown of the system-level execution time, prohibiting further optimization opportunities on other potential performance bottlenecks, such as I/O.

To address the above limitations, we propose a novel end-to-end framework for performance analysis and evaluation of sparse matrices and graph operation. The framework em-

Table 5.1: Properties of the evaluated sparse matrices

Matrix	rows	columns	nnz
12month1	12471	872622	22624727
bmwcra	148770	148770	10641602
bone010	986703	986703	47851783
consph	83334	83334	6010480
crankseg	52804	52804	10614210
pwtk	217918	217918	11524432
hood	220542	220542	9895422
sls	1748122	62729	6804304
torso1	116158	116158	8516500
mac_econ_fwd500	206500	206500	1273389

ploys parameterized graph models to generate synthetic graphs, account for different sources of noise in model parameters and choice of the correct model and evaluates the sensitivity of performance to these sources. The framework systematically analyzes the relationships between the input parameters of the sparse matrix/graph generators, and the performance of the sparse operations (e.g., SpMV) over the sparse matrix outputs of those generators. The idea being that if we can build this predictive understanding between the generator and the performance, then we can make guided decisions when operating over data that is approximated by those generators. Our framework focuses on choosing a representative set of features/parameters that relate to the performance of the operations on the input data, enabling a new horizon of performance optimizations. It provides multiple efficient ways of visualizing and relating performance to different parameters. Our framework also analyzes the overall system-level execution time to capture additional performance bottlenecks and drive optimization decisions.

The main contributions of this work are as follows:

1. Propose an extensible framework for performance analysis and evaluation for sparse data operations and driving design choice for performance optimizations.

2. Evaluate the usage of different graph model parameters and how it relates to performance interpolation and extrapolation.

3. Provide an alternative to using discrete graph sets for benchmarking sparse data workloads.

4. Estimate the effect of different noise sources in performance and integrating it into potential performance interpolations.

5. Present a system-level breakdown of execution time, rather than only focusing on the kernel to be optimized, unleashing new potentials for additional system-wide performance optimizations.

The rest of this chapter is organized as follows: Section 5.2 introduced the necessary background and discusses related work, its limitations, and how they motivate the proposal of our framework to address them. Section 5.3 details the description of our proposed framework. Section 5.4 shows a set of experiments conducted through our framework and discusses the results and observations. Finally, Section 5.5 summarizes the findings of the chapter.

5.2 Background and Related Work

5.2.1 Sparse Matrix and Graph Operations

Many traditional and modern applications require the operation on sparse data in the form of sparse matrices. Examples of these operations are Sparse Matrix-Dense Vector Multiplication (SpMV), Sparse Matrix-Matrix Multiplication (SpMM), and Sampled Dense-Dense Matrix Multiplication (SDDMM). SpMV, for instance, is used in many applications such as Natural Language Processing (NLP), scientific simulations, finite element analysis, image processing, solvers for partial differential equations (PDEs), and recommender systems. Also, traditional graph operations can be cast as linear algebra operations [47]. Due to the unique nature of sparse matrices, different sparse data formats were implemented to efficiently store them in memory. Examples of popular formats are COOrdinate (COO), Compressed Sparse Row

(CSR), and Compressed Sparse Column (CSC) [78]. The implementation of an algorithm for a sparse operation (e.g. SpMV) is traditionally tightly coupled to the sparse data format used for storing the sparse data. Multiple highly tuned linear algebra libraries are available to perform different sparse operations using different sparse data formats. Examples of vendor-specific libraries are Intel Math Kernel Library (MKL) [91] for CPU, NVIDIA cuSparse [74], and AMD rocSPARSE [5].

5.2.2 Graph Models for Sparse Data

Many performance evaluation techniques for sparse data have emerged in response to the generation of such data from engineering and physics problems. However, many real data is more accurately represented by large scale-free synthetic data that follow a power-law distribution such as Kronecker graphs [61, 58] or a combination of Kronecker + Random [82]. Kronecker graphs are a class of synthetic graphs that have been widely used to model real-world networks, and are generated by recursively applying the Kronecker product of a small base graph with itself. Let A and B be two matrices. Then, their Kronecker product $A \otimes B$ is given by

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix} \tag{5.1}$$

where a_{ij} are the entries of A. The resulting graph has a power-law degree distribution and exhibits a hierarchical structure that captures both the local and global connectivity patterns of the underlying real-world network.

To generate Kronecker graphs, an initiator matrix (typically of size 2×2 or 3×3) is chosen, and the Kronecker product is applied to this matrix by itself K times, where K is the Kronecker power. Then, a randomly generated probability matrix is used to mask out random values in the Kronecker matrix (remove edges from the Kronecker graph). Other

compute-efficient methods can be used to generate Kronecker graphs such as ball dropping and grass hopping [77].

Additionally, many graph models have been proposed to generate synthetic data that have similar properties to real graphs [26, 4, 35, 37, 87, 68, 101]. Frameworks have been proposed to classify input sparse data/graph into one of these models [3], and to evaluate the robustness of such graph models in terms of sensitivity of the graph structure to model parameters and noise [2].

While generative graph models (e.g. Kronecker graphs) provide a parameterized way of generating synthetic graphs similar to real graphs, tools that try to fit real data to such model (e.g. Kronfit [58]) are limited in estimation accuracy of the model parameters. Hence, our framework uses different generation models, but accounts for different sources of noise, including noise in graph generation parameters.

5.2.3 Performance Evaluation for Sparse Data Operations

In order to correctly understand how modern algorithms contribute to improving performance, several frameworks have been proposed. The Graph Algorithm Iron Law (GAIL) [9] targets graph processing algorithms, and proposes the usage of more adequate metrics, other than just execution time, to quantify performance contributions in regard to graphs. The proposed metrics include algorithmic work, communication volume, and bandwidth utilization. While these metrics can provide a better understanding of the performance improvements of different algorithms, the main focus of this work is performance metrics (vertical axis of performance plots), and not the graph model features/parameters which can affect these performance metrics (horizontal axis of performance plots).

The roofline model [93] has been the standard model used in performance evaluation, where theoretical machine peak performance and bandwidth bounds are calculated, and application performance is recorded as a point under the theoretical bounds curve. The x value for each point is the operational intensity (FLOPs/byte), and the y value is performance

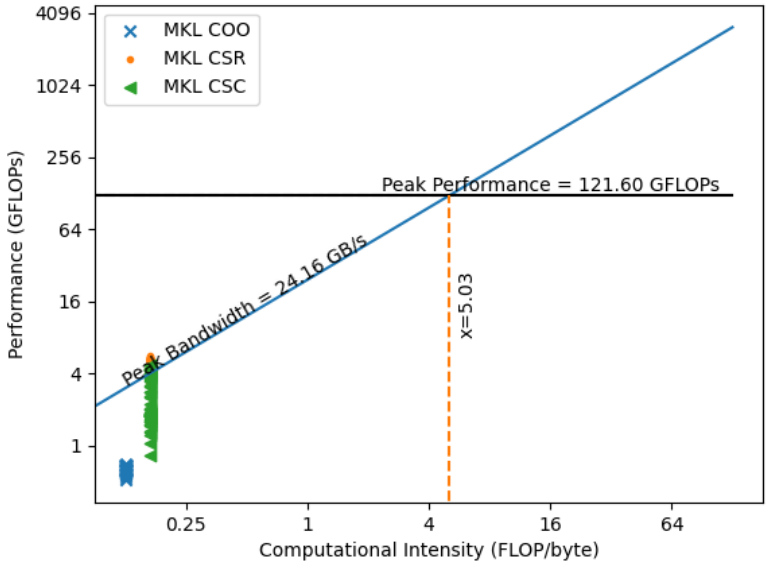


Figure 5.3: Performance Evaluation of SpMV in MKL for different sparse formats (COO, CSR, and CSC) on a set of sparse matrices using the Roofline model. Since the arithmetic intensity is imposed by the sparse data format, little insights are provided on how to optimize performance. Arithmetic intensity was estimated based on memory footprint for different storage formats and SpMV FLOPs.

(FLOPs). Additionally, many derived variants of the roofline model have been developed to accommodate for different memory hierarchy assumptions [71], capture the hardware changes in modern architectures such as GPUs [51, 98, 42], and work on finer granularity than a FLOP/byte such as instruction/transaction [23]. However, the application of the roofline model for applications on sparse data is less than adequate. Assuming we try to optimize SpMV operation, and we evaluate the performance of each algorithm using operational (arithmetic) intensity, and FLOPs. Since SpMV operations are directly coupled to a specific sparse data format (COO, CSR, CSC), the operational intensity for any implementation is fixed for a specific sparse data format, since the format imposes the size (bytes) of the data pieces involved in the SpMV operation. Figure 5.3 illustrates this issue, where the SpMV performance was evaluated using COO, CSR, and CSC for a set of input graphs. Most of the data points lie on the same vertical line in the figure, since they have similar arithmetic intensity imposed by the sparse data format. This kind of plots provides little insights on

how to optimize such operations on sparse data.

In response to the above limitations, we developed our framework to evaluate using different features on the horizontal axis of performance plots, which have the potential of being exploited for performance optimization. These features can be parameters used to generate graphs/sparse matrices using a specific graph model. In addition, our framework is flexible to incorporate any performance metric (vertical axis) similar to the ones proposed in existing work (e.g. GRAIL), while providing a better representation of datasets to enable performance interpolation.

5.2.4 Benchmarking Sparse and Graph data Workloads

In order to benchmark sparse and graph data workloads, appropriate input data needs to be fed to developing algorithms. Most of existing work in literature on different performance optimization for sparse matrices and graphs uses SNAP dataset [60], SuiteSparse Matrix Collection [21], and GAP [10]. These benchmarks provide a set of synthetic and real graphs/matrices from different applications and structures. However, they are limited in the sense that they are a discrete collection of graphs. Interpolating the performance of unseen graphs from a set of discrete graphs with no common continuity feature is challenging. For example, the GAP benchmark graph dataset consists of only five graphs. Tuning new algorithms on a discrete set of graphs, it is difficult to expect the performance interpolations to generalize across other sparse matrices and graphs. A recent work [32] proposed the use of artificial sparse matrix generators to tackle the issue of potential biased performance decisions based on discrete sets of matrices. However, the proposed generator parameters are limited to the observed SpMV bottlenecks features (average nnz per row, standard deviation of nonzeros per row, bandwidth of matrix, etc.). Hence, additional work is needed to provide a more comprehensive performance analysis framework that takes into consideration any input graph/sparse matrix generation models.

5.3 Methods

Our framework aims at providing a modern infrastructure for describing the performance of applications where sparse data and graphs are involved. As demonstrated in Section 5.2, existing techniques fall short in this category of irregular memory access applications. Our framework provides a means of interpolating and extrapolating the performance of different sparse matrices/graphs, based on models they closely fit.

The main goal of our framework is to find a relationship between the graph generation mechanism and the resulting performance of operations in which the graph is involved as an operand. Such analysis allows for the identification of promising graph generation parameters/features that show direct influence on performance. Algorithms can be developed to exploit these parameters to optimize the performance of operations where graphs of this model are used as operands.

The advantage of using our framework over profiling a single application is that it allows for a deeper understanding of the performance of applications that would operate over any matrix/graph that fits a specific graph model. It uses more representative features beyond dimensions or arithmetic intensity that might not be suitable in many cases.

5.3.1 High-Level Overview

A general overview of the framework is shown in Algorithm 1. All Graphs discussed are directed weighted graphs, where the vertices are row/column indices, and the weights on edges are non-zero values. Initial Graphs are generated using a parameterized graph model (generator). Each model takes as input a set of parameters. In addition to the initial set of graphs, the framework generates additional sets G_s by varying the input parameters within the legal range of values for each, while fixing the rest of the values.

Then, the framework induces two forms of noise indicated in Algorithm 1 as `noiseA` and `noiseB`. `noiseA` tries to capture noisy prediction of real data to the model parameter.

Algorithm 1 General Framework Description

for *param* in *model_gen_params* **do**
  for *val* in *param_legal_values* **do**
    $new\_params = val \cup (params - param\_old\_value)$
    $G = gen(new\_params)$
    append G to Gs
    for (n0=0;n0<nA_thresh;n0+=nA_step) **do**
      $G_{NA} = gen\_noiseA(G, n0)$
      append G_{NA} to Gs_{NA}
    end for
    for (n1=0;n1<nB_thresh;n1+=nB_step) **do**
      $G_{NB} = gen\_noiseB(G, n1)$
      append G_{NB} to Gs_{NB}
    end for
    for *op* in *operations* **do**
      for *impl* in *implementations*[*op*] **do**
        for fmt in $sparse\_formats$ **do**
          for *graph* in $G_s \cup Gs_{NA} \cup Gs_{NB}$ **do**
            $r = record(perf\_eval(impl(fmt(graph))))$
            append r to *results*
          end for
        end for
      end for
    end for
  end for
end for
for $feature$ in $features$ **do**
  $visualize(results, feature)$
end for

Algorithm 2 Performance Evaluation Routine

time $graph = load\_file(graph\_file)$
time $graph\_data = conv\_to\_fmt(graph)$
time $dev\_data = alloc\_mem\_dev(sizeof(graph\_data))$
time $cp\_mem\_to\_dev(dev\_data, graph\_data)$
time $warm\_up(operation, dev\_data, times)$
time $result\_dev = execute(operation, dev\_data, times)$
time $cp\_mem\_from\_dev(result\_host, result\_dev)$
time $verify(result\_host, golden\_result)$
time $deallocate\_mem()$

Graph models are expected to produce synthetic graphs with similar features to real-world graphs, but `noiseA` tests the effect of errors in these graph model parameters. `noiseB` on the other hand assesses the cases in which the model alone does not entirely describe the real data. Real graphs do not appear as pure representation of a model, noisy data might be added in the process of reading, transmitting, or pre-processing such graphs. Also, those graphs do not hold any node ordering guarantees.

The framework injects `noiseA` into G_s through the arithmetic addition of G_s with a set of random sparse matrices generated using a parameterized that varies from (`n0`) to `nA_thresh` from a uniform distribution, and a user-defined step of `nA_step` producing a new set of Graphs: Gs_{NA}. Injecting this noise is an arithmetic matrix addition between the adjacency matrices of the random sparse graph and the original graph. In this process, new edges may be added, and/or existing edges weight may change. Additionally, `noiseB` is added to G_s as a random sparse matrix with density `n1` in steps of `nB_step` up to a maximum of `nB_thresh`, generating the Gs_{NB} graph set.

After the completion of the graph sets generation phase, performance of such graphs involved as operands in operations is to be evaluated. Multiple operations can be executed where these graphs are operands, for example Sparse Matrix-Vector Multiplication (SpMV), in which the graph represents the sparse matrix. The graph or the sparse matrix can be represented using different sparse formats (COO, CSR, CSC, etc.), and each of these have their own implementation. The framework evaluates the performance of SpMV using the different formats and implementations for all generated graph sets.

The final step is to relate performance to different features and parameters of the graph model. These features and parameters are then used to represent the horizontal axis of the performance plots. The goal of such representation is to find a relationship between a feature or a set of features, and the performance of the operation on the graph. Using this information, more efficient algorithms can be tuned to optimize performance by exploiting features that exhibit strong correlation with performance.

5.3.2 Performance Evaluation

Algorithm 2 shows a more detailed description of the performance evaluation model. Generated graphs are stored as files. In order to evaluate their performance in an operation (SpMV), first step is to load each file into main memory. Then, depending on the evaluated sparse data format, a format conversion might be needed. If a device (GPU) is employed, necessary memory needs to be allocated on that device, and graph data structures in the target format need to be copied from the host to the device. Memories are warmed up a number of times to reduce performance numbers reporting errors. The main operation is then executed for a number of times, to record a distribution of execution times and check for any variance or outliers in the reported numbers. Operation results are copied back to the host and are verified for correctness using test harness. Finally, unused memory is freed. The application is instrumented and each of the described steps is separately timed to report the break-down of execution time and identify potential performance bottlenecks as well.

5.4 Evaluation and Results

The general framework described in Algorithm 1 generates a high-dimensional set of experiments involving different combinations of parameters and noise values. We conducted a sub-set of experiments to showcase the capabilities of our framework. In this section, we report planar slices of some of the experiments. Our framework was evaluated for both CPU and GPU. Table 5.2 shows the configuration for the system used in our experiments. For MKL, the number of threads used is the number of CPU cores on the system. For each of our experiments, ten iterations of SpMV computation are measured after ten warmup iterations. There is little variance of the execution time across the ten runs, so the average execution time is reported and converted into GFLOPs/s throughout the rest of this section. Table 5.3 shows the properties of the different synthetic sparse matrices (graphs) that were used in the evaluation.

Table 5.2: System Configuration

Component	Specification
GPU	NVIDIA RTX H100
GPU Memory	80 GB
CUDA Version	12.0
CPU	Intel Xeon Gold 6338 @ 2.00GHz
CPU Sockets	2
CPU Cores per Socket	32
CPU Threads per Core	2
MKL Version	2022.1.0
Main Memory	256 GB DDR4

Table 5.3: Used Synthetic Matrices Properties

Model	Dimensions	Min NNZ	Max NNZ
K21	2097152×2097152	26556004	365044062
K15	32768×32768	200896	1306053
Random	8192×8192	1024	67108864

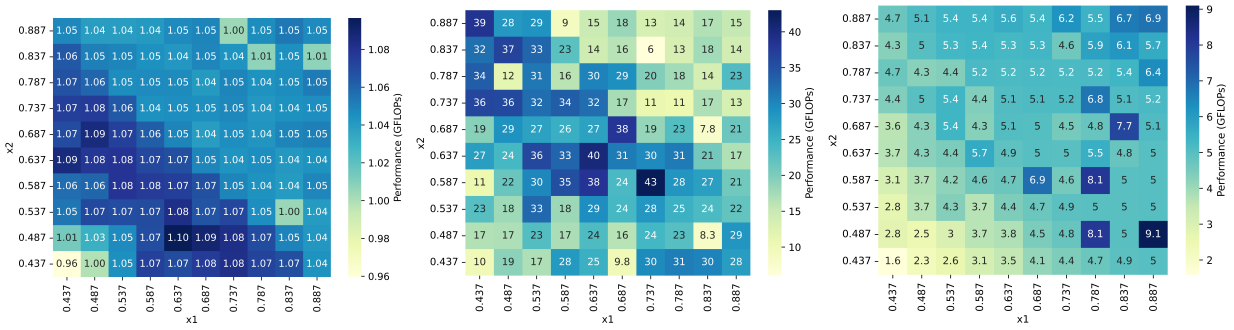
5.4.1 Graphs Generated by Varying Model Parameters

In the first phase of our framework, a set of graphs is generated by varying different graph model parameters. For this experiment, we used the Kronecker Graph model. The choice of the Kronecker graph model as one model throughout the rest of this chapter is because (1) the Kronecker model has a small and clear set of input parameters that can be dialed, and (2) tools exist that can map real-world graphs to their best fit Kronecker graph approximation (e.g, Kronfit [58]). If a real graph is well approximated by this model, then we would expect the performance of operations over that graph fit the profile produced by our framework. However, these components are replaceable. For example, different parameterizable generative models that capture dense block structures could be used, and the performance could be measured on a more SIMD amenable format such as Blocked Compressed Sparse Row (BCSR). This would provide an ideal range of inputs for this format, so our framework provides the entry points for adding noise to help find where BCSR would break down. This is in contrast to the standard approach of benchmarking sparse matrix and graph operations

of using standard collections of datasets, where such an evaluation may not be possible.

K15 Graphs with Varying Initiator Matrix – Heatmaps

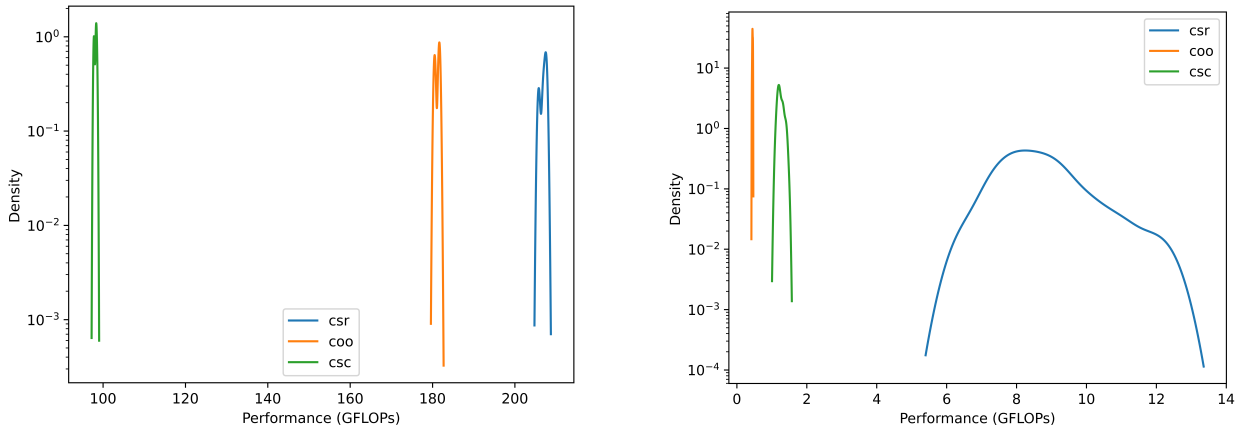
A Kronecker power of 15 was used, and the initiator matrix values were varied as follows: we start with a sample 2×2 initiator matrix of the values $[0.999, 0.437; 0.4370.484]$, matching the estimated initiator values by Kronfit [58] for the High Energy Physics - Phenomenology Collaboration (CA-HEP-PH) Graph [59] from the SNAP dataset. Then, we fix the first and last initiator matrix values, while varying the other two, producing different combinations of them. For each of the new generated initiator matrices, a new Kronecker graph is generated. Finally, we evaluate the performance of each as a sparse matrix in a SpMV operation.



(a) COO (b) CSR (c) CSC

Figure 5.4: MKL SpMV performance of K15 Kronecker Graphs with varying initiator matrix values x_1 (x-axis), and x_2 (y-axis). The graph is represented in (a) COO, (b) CSR, and (c) CSC formats.

Figure 5.4 shows heatmaps generated by our framework, representing the performance of SpMV for the generated K15 graphs, using Intel MKL for different sparse data formats: COO, CSR, and CSC. The choice of heatmap for the visualization of the Kronecker graph performance enables observing relationship between two different features (parameters) of the model (two initiator matrix values), and how the performance changes with varying both of the parameters. Also, the comparison between different sparse data formats (COO, CSR, and CSC) drives the decision of choosing the ideal data format, for the given input graph model (K15), tool (MKL), and architecture (CPU). The figure clearly shows that CSR is



(a) cuSparse on H100

(b) MKL on Intel Xeon Gold

Figure 5.5: SpMV performance KDE for 100 Kronecker graphs generated from the same initiator matrix using a Kronecker power of 21. Tools evaluated are (a)cuSparse on H100 GPU, and (b)MKL on Intel Xeon Gold CPU. COO, CSR, and CSC sparse formats are evaluated. Performance in GFLOPs is shown on the horizontal axis, and density is on the vertical axis.

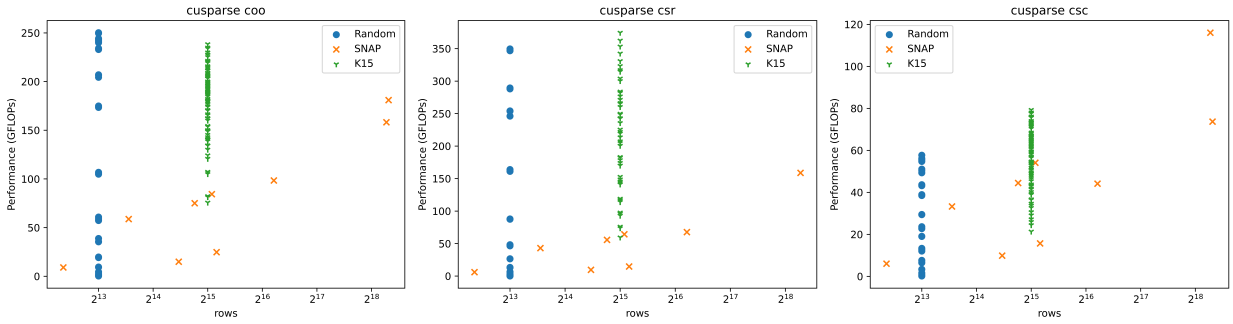
a winner among the three evaluated formats in this specific situation. It also shows that the performance of COO is stable across different x_1 and x_2 values, so no potential benefit appears from optimizing using these two parameters for this specific format.

K21 Graphs with the Same Initiator Matrix – KDE

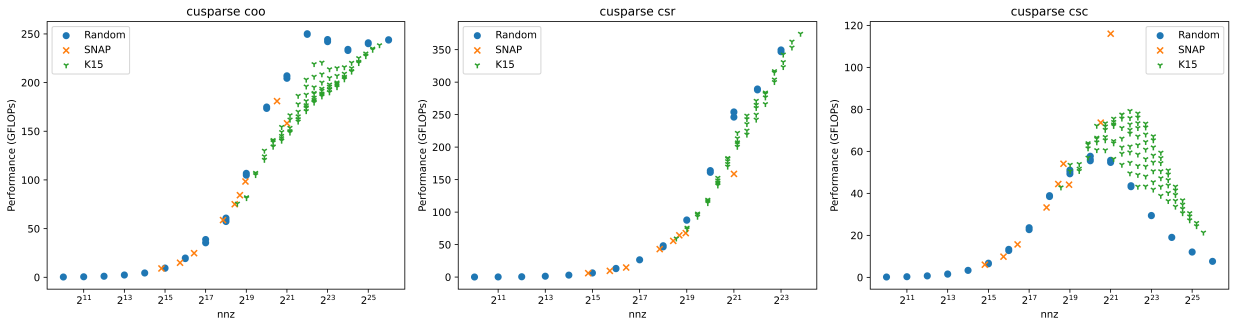
In contrary to the previous experiment, we fix all initiator matrix values and generate 100 different Kronecker graphs using the same initiator matrix. However, we vary the Kronecker power from 15 to 21. Then, the framework evaluates the performance of the generated graphs using two tools: NVIDIA cuSparse on GPU, and Intel MKL on CPU. For each of the tools, we evaluate three different sparse representations: COO, CSR, and CSC.

Figure 5.5 shows the performance results for this experiment. Instead of using heatmaps to observe the performance variation across a grid of different initiator matrix values, we use Kernel Density Estimation (KDE) plots to visualize the frequency of different performance ranges (in GFLOPs). The horizontal axis represents the SpMV performance, while the vertical access represents the number of graphs achieving that performance. This type of

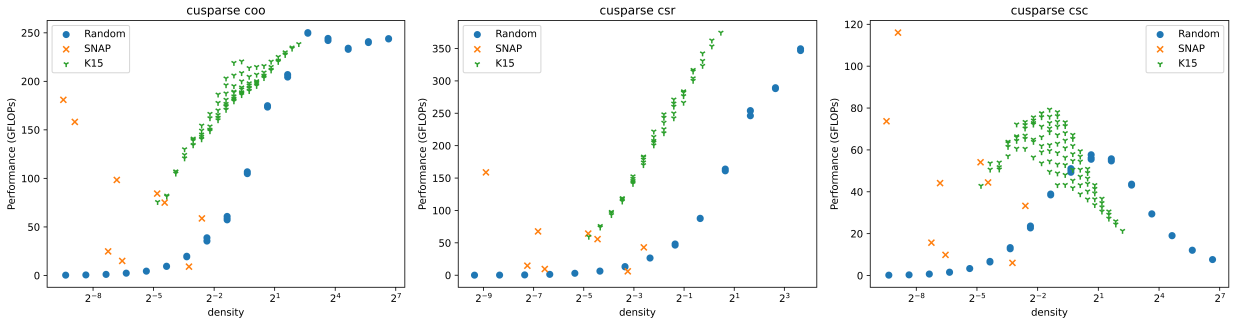
plots is informative when a decision about the optimal sparse format is to be made. Figures 5.5a and 5.5b show that CSR is also the best performing format for the generated K21 graphs of the same initiator matrix on both CPU and GPU.



(a) COO Performance vs nrows (b) CSR Performance vs nrows (c) CSC Performance vs nrows



(d) COO Performance vs nnz (e) CSR Performance vs nnz (f) CSC Performance vs nnz



(g) COO Performance vs density (h) CSR Performance vs density (i) CSC Performance vs density

Figure 5.6: cuSparse SpMV performance for: random, SNAP, and K15 graphs plotted against number of rows, number of non-zeros, and density on the horizontal axis. COO, CSR, and CSC formats are evaluated.

5.4.2 Multiple Different Models with Different Features

In this experiment, we evaluate the performance of different graph models (vertical axis) and relate that performance to different features of the models (horizontal axis). The purpose of this experiment is to show if we can directly compare the performance of different sparse matrix/graph models using common features. This shows if we can interpolate or extrapolate the performance of different model from existing performance results, by dialing different parameters/feature.

To conduct this experiment, we used three graph models: random graphs generated using the density parameter, Kronecker graphs generated using K power 15 and different initiator matrix values and select graphs from SNAP dataset collection. The selected SNAP graph are shown in Table 5.4.

Figure 5.6 shows the performance results of this experiment using cuSparse on H100 GPU, plotted against number of rows, number of non-zeros, and density used as features on the horizontal axis. Each subplot illustrates the performance of a specific sparse data representation out of the three we evaluated: COO, CSR, and CSC.

Looking at the relationship between Performance and number of rows (Figure 5.6a, 5.6b, 5.6c), one can observe that it is not a suitable feature to tune for performance, as compared to the case in dense matrices. For example, for random sparse matrices of the same number of rows, performance varies significantly across the entire range of observed performance.

Regarding the choice of ideal format, Figure 5.6 shows the need of using our framework to sweep across a wide range of graph parameters and noise to generate graphs and make optimization decisions. For the SNAP subset of graphs we evaluated, the maximum attained performance was for the `web-NotreDam` in COO format, at 181 GFLOPs. If one was to tune for only this subset of SNAP graphs, a conclusion to use the COO format would have been made. However, throughout our experiment, we can see that CSR shows the global highest performance across the three graph models, using cuSparse on the H100 GPU.

Random graph generators use a main parameter: density. All of the generated random

graphs were of the same dimensions (square). We can see that density (Figure 5.6g, 5.6h, 5.6i) as a feature on the x-axis, capture performance well for the random graph, since it is directly a graph model parameter. However, it does not work as well for Kronecker graphs; multiple graphs with the same density exhibit different performance characteristics. Also, for SNAP, looking at the scattered performance points, one cannot interpolate or extrapolate the performance (using existing performance data) at different density values that have not been evaluated.

For number of non-zeros (Figure 5.6d, 5.6e, 5.6f), we can see it can be better utilized to an extent to interpolate SNAP graphs performance. However, Kronecker graphs still illustrate performance variations for similar `nnz` values, where some formats (CSC) are more unstable than others (CSR).

Table 5.4: Properties of the evaluated SNAP graphs

Graph	**Nodes**	**Edges**
soc-Epinions1	75,879	508,837
cit-HepPh	34,546	421,578
cit-HepTh	27,770	352,807
ca-HepPh	12,008	118,521
web-NotreDame	325,729	1,497,134
ca-GrQc	5,242	14,496
p2p-Gnutella25	22,687	54,705
p2p-Gnutella30	36,682	88,328
com-DBLP	317,080	1,049,866

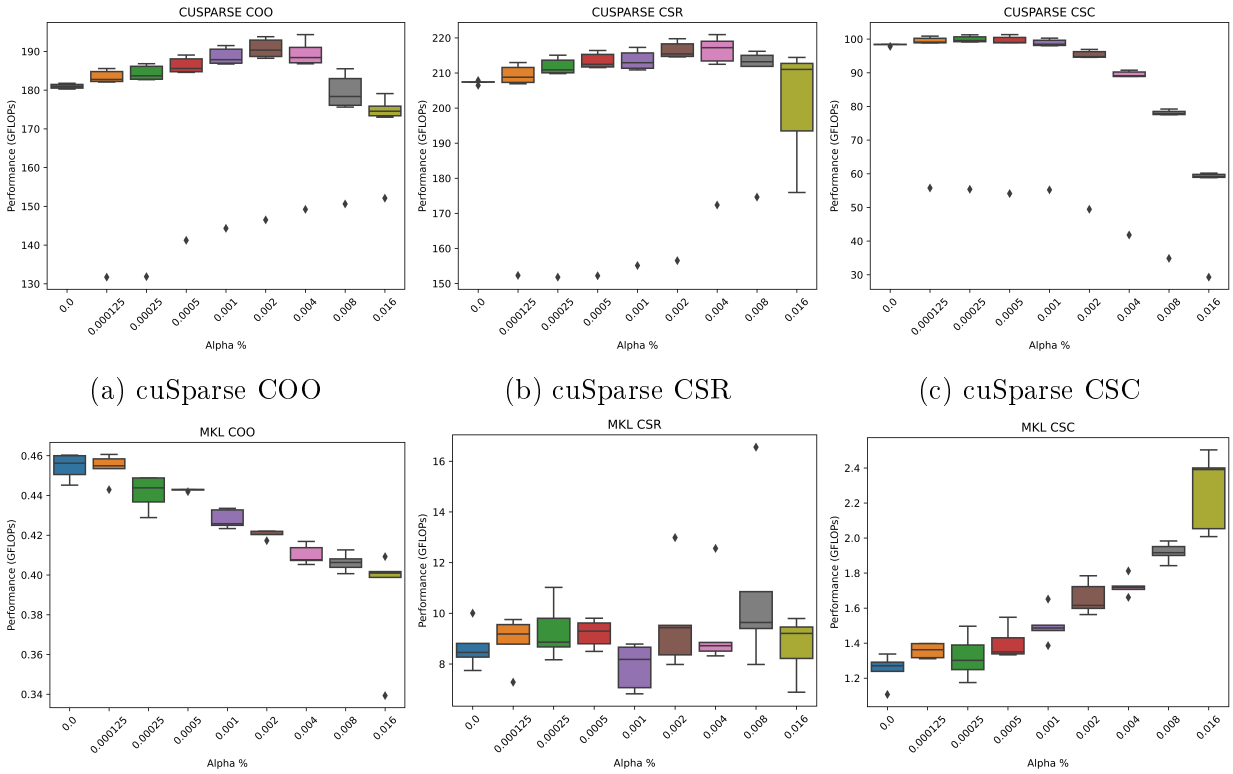
5.4.3 Inducing Noise

Following the general description provided in algorithm 1, our framework evaluates the sensitivity of performance to different types of noise: `noiseA`, which represents the error from fitting a dataset to the model being generated, and `noiseB`, which represents using the wrong model for a dataset. The heatmaps shown in Figure 5.4 can also serve as a means of evaluating `noiseA`: noise added to input graph parameters, since each cell in the heatmap represent the performance of a graph generated by varying two initiator matrix values.

For `noiseB`, we present two slices of the high-dimensional search space of noise: the first being injected noise in the form of adding a random sparse graph of varying edge densities to the original graph, and the second being swapping (re-labelling) nodes in the original graph a number of times.

Adding Random Sparse Graphs with Varying Density

For this experiment, we use a smaller subset of the K21 graphs generated before. For each of them, we generate a number of sparse random matrices, with varying density. In our experiment the lowest noise density was 0.000125% ($0.00000125 \times 2^{21} \times 2^{21} = 5,497,559$ additional random edges).



(a) cuSparse COO (b) cuSparse CSR (c) cuSparse CSC

(d) MKL COO (e) MKL CSR (f) MKL CSC

Figure 5.7: SpMV Performance Boxplots for adding noise in the form of random sparse matrix with varying density (Alpha %) to a subset of the generated K21 graphs. Performance Evaluation is performed using both cuSparse and MKL for COO, CSR, and CSC sparse data formats.

For cuSparse, we can see that adding noise to Kronecker graphs in the form of random sparse matrices, changes the median performance within ±10 GFLOPs for COO and CSR (Figure 5.7a, 5.7b). However, for MKL, the range of change for median performance is limited: less than 0.06 GFLOPs for COO, around a single GFLOP for CSR and CSC. This kind of performance sensitivity to noise analysis enables the estimation of performance of different graph models, given an expected amount of noise, architecture, tool, and operation.

Relabelling Graph Nodes

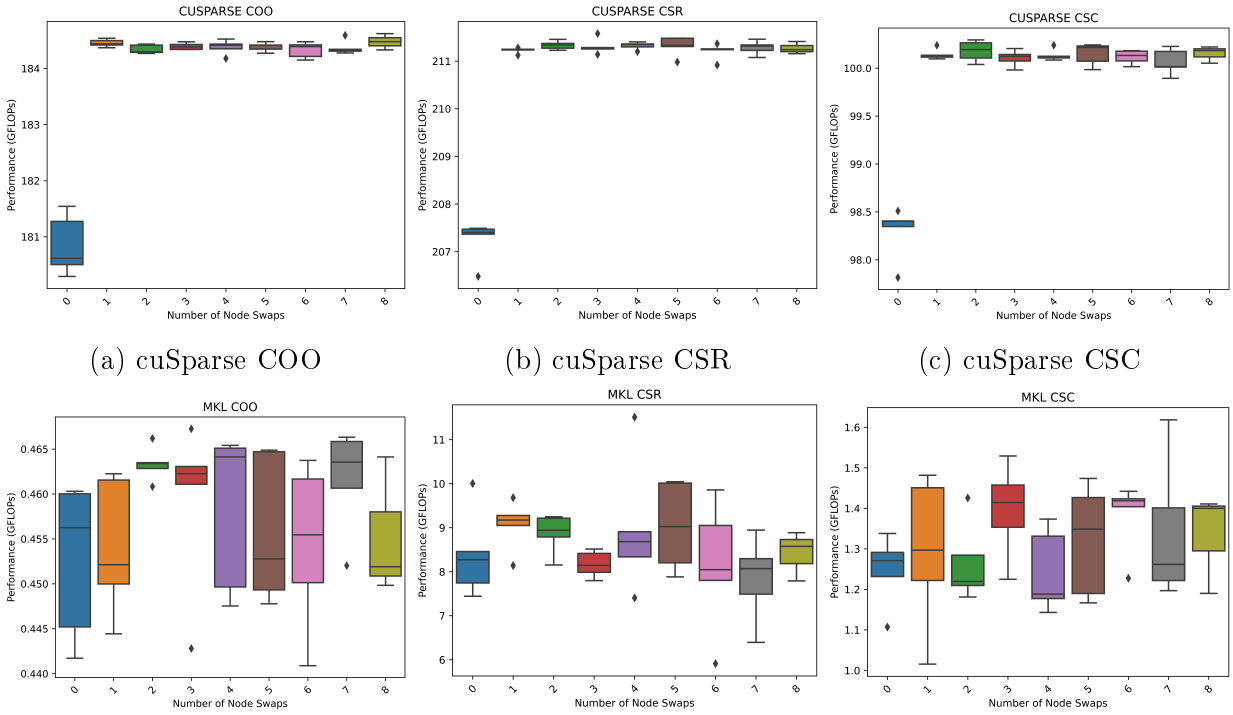
Another kind of `noiseB` that we evaluated using our framework is re-labelling nodes. The re-labelling mechanism is implemented as follows: two nodes are chosen using a weighted probability (heavier nodes have a higher chance of being picked). Then, we swap the two nodes labels in all edges they are a source or a destination in. This counts as a single swap, and we evaluate a different number of swaps. We evaluated up to eight swaps only because the swapping operation is computationally expensive.

Figure 5.8 shows the performance (vertical axis) sensitivity to swapping node labels in the original graphs for a number of times (horizontal axis). A subset of the previously generated K21 Kronecker graphs was also used for this experiment. For cuSparse, the range of change for the median performance (GFLOPs) is around 3 GFLOPs. Most of the change happens as soon as the first swap happens, and then performance almost stabilizes for up to 8 swaps. The same effect is observed across all the three evaluated sparse formats (COO, CSR, and CSC) on the GPU.

For MKL, the effect of swapping nodes up to 8 swaps is limited on performance, since the original performance range of SpMV for these graphs is narrower than that of cuSparse.

5.4.4 System-Level Runtime – Practical Considerations

All performance results reported so far are only for the actual operation (e.g. SpMV) invoked on the sparse data. However, in a practical setting, this is not the only overhead that needs to



(a) cuSparse COO (b) cuSparse CSR (c) cuSparse CSC

(d) MKL COO (e) MKL CSR (f) MKL CSC

Figure 5.8: SpMV Performance Boxplots for adding noise in the form of swapping the labels of node pairs, selected based on a weighted probability according to their degree. The swap is performed a number of times (horizontal axis). Performance Evaluation is performed using both cuSparse and MKL for COO, CSR, and CSC sparse data formats.

be evaluated, as the overall system runtime involves additional steps as shown in Algorithm 2, where recording the performance of the actual computation represents only `line 6` of the algorithm.

To this end, we isolated a single run and instrumented the different phases of execution. For this experiment, we used cuSparse with one input graph from the generated K21 Kronecker Graphs in COO format. Table 5.5 shows the percentage of execution time each of the activities is taking. The "other" set of activities include loading dynamic libraries, creating cuSparse different structures, allocating cuSparse SpMV specific buffers, etc. From this table, we can see that loading the graph file (from disk) and parsing it to store the sparse data according to the COO structure takes up the majority of execution time. In our experiments, we used the Matrix Market (`.mtx`) format to store our graphs.

This observation suggests that it is crucial to develop more efficient techniques to load, parse, and store sparse data in different sparse formats. Another direction is devising more efficient file formats for sparse data. Also, using the same graph a large number of times in multiple computations can amortize for the high cost of loading the graph. Our framework provides detailed analysis and insights that can greatly drive the optimization process for different practical settings.

Table 5.5: Execution Time Breakdown for 1 instance of SpMV in COO using cuSparse on H100 GPU as percentages of the total binary execution time.

Activity	**Percentage**
Loading (from disk) and Parsing Graph File in COO	89.17%
Memory allocation on GPU	0.001%
Copy from host to GPU	0.12%
SpMV Warmup (10 times)	0.008%
Actual SpMV (1 time)	0.0008%
Copy result from GPU to host	0.0024%
Result Verification on host	0.967%
Free memory (host and GPU)	0.061%
Other	9.66%

5.5 Summary

In this chapter, we propose a highly modular framework for evaluating and analyzing the performance of sparse matrix and graph operations. Our proposed framework makes use of parameterized graph models to generate graphs by varying these parameters and observing performance. It also evaluates the effect of inducing different types of noise to the performance of sparse data operations: noise due to error in model fitting tools, and noise rising from using the wrong model for the data. Our framework focuses on evaluating performance (using different metrics) against representative parameters/features (horizontal axis of performance plots), from which performance interpolations and extrapolation can be performed. It also aims at overcoming the existing limitation of using discrete graph sets to tune the performance of sparse matrix and graph kernel. We show results from sets of experiments, conducted through our framework to show the potential it provides to draw insightful performance optimization decisions.

Chapter 6

High-Level Optimizations for Sparse Computations

6.1 Introduction

Optimizing computational operations to improve performance is a cornerstone of computational linear algebra, especially for operations like matrix-vector multiplication (MatVec). A well-known optimization technique that exploits data reuse (locality) is *tiling*, which organizes computations to maximize the efficiency of data cached during execution. Such optimizations are well-established in the context of dense linear algebra operations, notably in General Matrix Multiply (GEMM) and dense MatVec operations, where they can be applied relatively straightforwardly due to the regularity and predictability of data access patterns. The benefits of these optimizations, including reduced memory bandwidth usage and enhanced cache utilization, are significant, leading to substantial performance improvements on modern computing architectures.

However, when it comes to sparse operations, such as Sparse Matrix-Vector multiplication (SpMV), the scenario drastically changes. The irregularity of non-zero elements' distribution in sparse matrices complicates the direct application of high-level optimizations

like tiling. The unpredictability of data access patterns in sparse operations challenges traditional optimization strategies, as the benefits of data locality and reuse become much harder to exploit. Consequently, while techniques such as loop unrolling and vectorization remain applicable, their effectiveness is often limited by the sparsity pattern. This inherent difficulty in optimizing sparse operations stems from the need to navigate the trade-offs between the computational overhead introduced by managing sparse data structures and the potential performance gains from optimization techniques. Thus, optimizing sparse operations demands more sophisticated, context-aware strategies that can adapt to the unique characteristics of each sparse matrix.

As established in the previous chapters and in literature, the performance of sparse operations mainly depends on the structure of the sparse matrix, and the sparse storage format used. The structure of the sparse matrix can be represented using the parameters used to generate the sparse matrix using graph models as discussed in Chapters 4 and 5. Ultimately, our goal is providing a model that takes as input the graph model and its parameters, and outputs the optimizations that yield the best performance for the input. In this work, we explore different high-level optimization strategies and its impact on sparse performance. We use Sparse-Matrix Vector Multiplication (SpMV) as an example for sparse operations.

6.2 Background and Related Work

Tensor algebra is an essential computational paradigm across various domains, including data analytics, machine learning, engineering, and physical sciences. Addressing the unbounded complexity of tensor expressions and the challenges posed by sparse tensor representations, Kjolstad et al. introduced *TACO* [50], a pioneering code generation tool. TACO is designed to automate the generation of efficient, parallelized kernels for both dense and sparse tensor algebra expressions. This tool not only aids in the practical execution of tensor algebra oper-

ations but also significantly reduces the development effort by obviating the need for manual kernel optimization. Taco's versatility extends to handling mixed kernels, thereby offering a comprehensive solution for the increasing demand for sophisticated tensor computations.

For example, in order to generate the code for SpMV using TACO, one can use the provided command line tool and provide a simple expression as follows:

```
taco "y(i) = A(i,j) * x(j)" -f=x:d -f=A:ds -f=y:d
```

Here `y` is the output dense vector, `A` is the sparse matrix, and `x` is the input dense vector. Notice the usage of index variables `i,j` in tensor notation to provide the semantics of the multiplication operation. Notice also that the user can provide the formats for each dimension of the input vectors/matrices/tensors through the command line option `-f` where the value `d` means a dense dimension, while `s` means a sparse dimension. In TACO, the format `ds` used here corresponds to CSR format. In addition to the standalone command line tool, TACO also provides APIs for both C++ and Python. The TACO code generated for this kernel is shown in Figure 6.1.

TACO stands out by its ability to navigate the intricacies of sparse tensor storage and computation. Traditionally, the sparsity in tensors, indicative of a predominant presence of zeros, necessitates bespoke compressed formats and tailored computational kernels for efficient storage and processing. TACO automates this process, generating kernels that adeptly manage sparse formats, thereby optimizing memory usage and computational performance. The tool supports a broad spectrum of tensor algebra expressions using tensor index notation. With TACO, developers can quickly prototype and optimize tensor operations, exploring various storage formats and computational strategies without delving into the complexities of hand-written kernel development.

Tiling or blocking is a crucial loop optimization used to improve temporal locality in many applications [44, 94, 40, 1]. The basic idea of tiling is to divide input data into smaller tiles (blocks) that fit into cache, and keep them in the cache for as long as possible

```
int compute(taco_tensor_t *y, taco_tensor_t *A, taco_tensor_t *x
    ) {
    int y1_dimension = (int)(y->dimensions[0]);
    double* restrict y_vals = (double*)(y->vals);
    int A1_dimension = (int)(A->dimensions[0]);
    int* restrict A2_pos = (int*)(A->indices[1][0]);
    int* restrict A2_crd = (int*)(A->indices[1][1]);
    double* restrict A_vals = (double*)(A->vals);
    int x1_dimension = (int)(x->dimensions[0]);
    double* restrict x_vals = (double*)(x->vals);

    #pragma omp parallel for schedule(runtime)
    for (int32_t i = 0; i < A1_dimension; i++) {
       double tjy_val = 0.0;
       for (int32_t jA = A2_pos[i]; jA < A2_pos[(i + 1)]; jA++) {
       int32_t j = A2_crd[jA];
       tjy_val += A_vals[jA] * x_vals[j];
       }
       y_vals[i] = tjy_val;
    }
    return 0;
    }
```

Figure 6.1: TACO generated code for SpMV (CSR format)

```
for (int i = 0; i < n; i++){
   for (int j = 0; j < n; j++){
      for (int k = 0; k < n; k++){
         C[i,j] += A[i,k] * B[k,j];
      }
   }
}
```

(a) Naive Implementation

```
for (int i = 0; i < n; i += BLOCK_DIM){
   for (int j = 0; j < n; j += BLOCK_DIM){
      for (int k = 0; k < n; k+= BLOCK_DIM){
         for (int ii = i ; ii < i+BLOCK_DIM; ii++){
            for (int jj = j; jj < j+BLOCK_DIM; jj++){
               for (int kk = k; kk < k+BLOCK_DIM; kk++){
                  C[ii,jj] += A[ii,kk] * B[kk,jj];
               }
            }
         }
      }
   }
}
```

(b) Tiling with block size= BLOCK_DIM × BLOCK_DIM

Figure 6.2: General Matrix-Matrix Multiplication Code for the Expression $C_{ij} = A_{ik} \cdot B_{kj}$ using (a) naive three-loop implementation, (b) tiling.

while computing on them. For example, in most general dense matrix-matrix multiplication (GEMM) implementations, tiling is employed to split input matrices into tiles (blocks) of dimensions that fit into cache and perform partial matrix multiplication on these tiles while in cache. Tiling is usually implemented using two other loop optimizations: stripmining, where an original loop is split into two loops, and permutation where the order of two loops is permuted. Figure 6.2 shows an example code for a simple GEMM for the expression $C_{ij} = A_{ik} \cdot B_{kj}$ where A, B, and C are square $n \times n$ matrices. The naive implementation (Figure 6.2a) uses three loops. In order to perform tiling, the three loops are stripmined (split), each into two loops, then the order of the loops is permuted to reflected tile traversal correctly as shown in Figure 6.2b.

6.3 Methods

6.3.1 Sparse Tiling using TACO

The main idea of our optimization is to simulate a tiling effect of sparse matrix-vector multiplication through a data layout transformation. The tensor index expression for SpMV is: given a matrix A and a vector x, the result of the multiplication $y = Ax$ can be expressed in tensor index notation as:

$$y_i = A_{ij} \cdot x_j$$

Here, y_i represents the i-th component of the result vector y, A_{ij} represents the elements of the matrix A, and x_j represents the components of the vector x. The indices i and j range over the dimensions of the matrix A and the vector x, respectively. The operation implies a summation over the index j, which aligns with the Einstein summation convention where repeating indices in a term imply summation over those indices.

This notation succinctly captures the essence of the matrix-vector multiplication, where

each element of the resulting vector y is calculated as the dot product of a row of matrix A with the vector x.

While there is no re-use in the sparse matrix involved in this operation, there is a potential of re-use in the dense vectors by tiling (blocking) the sparse matrix. However, tiling in sparse matrices is expensive, as it requires scanning the sparse matrix to find dense regions.

One way to circumvent this limitation is to bake the tiling in the sparse matrix data layout itself. Given an input two-dimensional matrix, we raise it to a higher-dimensional tensor (4D). The conversion from 2D to 4D in the data is simply by converting the indices of non-zeros from 2D (i, j) to 4D (i_o, j_o, i_i, j_i) as shown in 3.

Algorithm 3 Transform 2D indices to 4D. Here, nb is the number of rows per block, and mb is the number of columns per block.

function TO_4D(i, j, nb, mb)
  $io \leftarrow i \div nb$ ▷ Compute the outer row index
  $ii \leftarrow i \mod nb$ ▷ Compute the inner row index
  $jo \leftarrow j \div mb$ ▷ Compute the outer column index
  $ji \leftarrow j \mod mb$ ▷ Compute the inner column index
  return (io, ii, jo, ji)
end function

Now, after raising the sparse matrix into a 4D tensor, the SpMV operation can be converted to a tensor-times-matrix operation as follows:

$$Y_{i_o j_o} = A_{i_o j_o i_i j_i} \cdot X_{i_i j_i}$$

The code generated by TACO for this computation by invoking

```
taco "Y(io,jo) = A(io,jo,ii,ji) * X(ii,ji)" -f=A:ssss -f=X:dd
    -f=Y:dd
```

is shown in Figure 6.3

```
int compute(taco_tensor_t *Y, taco_tensor_t *A, taco_tensor_t *X) {
  int Y1_dimension = (int)(Y->dimensions[0]);
  int Y2_dimension = (int)(Y->dimensions[1]);
  double* restrict Y_vals = (double*)(Y->vals);
  int* restrict A1_pos = (int*)(A->indices[0][0]);
  int* restrict A1_crd = (int*)(A->indices[0][1]);
  int* restrict A2_pos = (int*)(A->indices[1][0]);
  int* restrict A2_crd = (int*)(A->indices[1][1]);
  int* restrict A3_pos = (int*)(A->indices[2][0]);
  int* restrict A3_crd = (int*)(A->indices[2][1]);
  int* restrict A4_pos = (int*)(A->indices[3][0]);
  int* restrict A4_crd = (int*)(A->indices[3][1]);
  double* restrict A_vals = (double*)(A->vals);
  int X1_dimension = (int)(X->dimensions[0]);
  int X2_dimension = (int)(X->dimensions[1]);
  double* restrict X_vals = (double*)(X->vals);

  #pragma omp parallel for schedule(static)
  for (int32_t pY = 0; pY < (Y1_dimension * Y2_dimension); pY++) {
    Y_vals[pY] = 0.0;
  }

  #pragma omp parallel for schedule(runtime)
  for (int32_t ioA = A1_pos[0]; ioA < A1_pos[1]; ioA++) {
    int32_t io = A1_crd[ioA];
    for (int32_t joA = A2_pos[ioA]; joA < A2_pos[(ioA + 1)]; joA++) {
      int32_t jo = A2_crd[joA];
      int32_t joY = io * Y2_dimension + jo;
      double tiiY_val = 0.0;
      for (int32_t iiA = A3_pos[joA]; iiA < A3_pos[(joA + 1)]; iiA++) {
        int32_t ii = A3_crd[iiA];
        for (int32_t jiA = A4_pos[iiA]; jiA < A4_pos[(iiA + 1)]; jiA++) {
          int32_t ji = A4_crd[jiA];
          int32_t jiX = ii * X2_dimension + ji;
          tiiY_val += A_vals[jiA] * X_vals[jiX];
        }
      }
      Y_vals[joY] = tiiY_val;
    }
  }
  return 0;
}
```

Figure 6.3: TACO generated code for 4D Sparse Tensor Times Dense Matrix, where all tensor dimensions are sparse

6.3.2 A Header-only Library for Sparse Tiling

In addition to evaluating TACO, we introduce a novel C++ header-only library for sparse tiling. The main idea is to split the input sparse matrix into multiple tiles (blocks) regardless of the sparse operation. This means that tiling is tied to the data layout. Figure 6.4 shows a pseudo-code example of using the library. The user first would read an input sparse matrix from a file (Matrix Market File) by creating an instance of the `SpMat` class, providing the precision as a template parameter for the class. Then, the user can invoke the function `tile` on the `SpMat` instance, passing the desired number of rows per tile, and number of columns per tile. Afterwards, the user reads or generate the dense vector operator, and allocate memory for the result dense vector. Finally, an invocation of `multiply` on the `SpMat` instance is needed, passing both dense vectors as arguments.

tile call divides the matrix into a number of tiles or submatrices, each with number of rows = `rowsPerTile` and number of columns = `colsPerTile`. These sub-matrices or tiles are stored in CSR format (`CSRMatrix<T>`) within a `std::map` that keeps track of the tile indices (tile row and tile column) as a key.

multiply iterates over the map containing CSR blocks, and performs partial multiplication for each with the corresponding slice of the dense vector. This potentially improves data re-use for the dense vector. The backend used for performing individual tile sub-vector multiplication can be plugged in or out to use any SpMV implementation (MKL, cuSparse, etc.).

```
1    SpMat<float> inputMatrix = SpMat<float>("matrix_file.mtx");
2    inputMatrix.tile(rowsPerTile, colsPerTile);
3    std::vector<float> dVec;
4    std::vector<float> result(inputMatrix.nrows());
5    inputMatrix.multiply(dVec, result);
```

Figure 6.4: An example for using our library for a simple SpMV operation

6.4 Evaluation and Results

6.4.1 TACO experiments

Table 6.1: System Configuration

Component	Specification
CPU	Intel Xeon Gold 6338 @ 2.00GHz
CPU Sockets	2
CPU Cores per Socket	32
CPU Threads per Core	2
TACO Version	0.1+git 823e8dac
Main Memory	256 GB DDR4

Table 6.1 shows the system configuration used to conduct TACO experiments. To evaluate the efficiency of raising the 2D sparse matrices into 4D sparse tensors before the sparse multiplications, we generate a dataset of 1716 matrices using the Kronecker graph model [56]. The Kronecker power for the graphs generated vary from 16 to 22. For the initiator matrix values, we start from an initial set of values based on Kronfit's [58] estimated Kronecker initiator values [56] of the High Energy Physics - Phenomenology Collaboration (CA-HEP-PH) Graph [59]. Then, we vary each of these values while fixing the others with a step = 0.001 in both directions (up and down) within the domain of legal values (between 0 and 1).

For each of input matrices, to generate 4D tensors, we use Algorithm 3 to convert the 2D indices into 4D indices. For each of the input matrix files, we generate a set of tensor files for different block sizes (powers of two, up to 4096 for both original dimensions). We try multiple different ordering of indices: `(io, ii, jo, ji)`, `(io, jo, ii, ji)` to reflect different traversal orders.

The following variants of sparse multiplications are then evaluated:

1. Baseline SpMV: The Normal SpMV with sparse matrix. For the input sparse matrix, we evaluate different formats for the dimensions `ds,ss`

2. Blocked SpMV: On top of the normal SpMV, we feed TACO a schedule to stripmine then re-order the multiplication loop. Such schedule can be provided to TACO by splitting the first index variable i into two index variable io, ii using the number of rows per block nb, and repeating similarly for the j index variable as follows:

```
-s='split(i,io,ii,\$nb),split(j,jo,ji,\$mb),reorder(io
   ,jo,ii,ji)
```

   Here also `nb` and `mb` are varied through powers of two up to 4096.

3. Tensorized Multiplication: We use the generated 4D tensors to do sparse tensor-dense matrix multiplication.

4. Tensorized Multiplication with Re-ordering: Same as tensorized multiplication, but we change the order of the two innermost index variables in the tensor notation. For re-ordering we used two different mechanisms: (a) Using TACO schedules, and (b) Manually flipping the order in the tensor index expression.

Each of the experiments is run ten times, and then the median execution time is used as representative of performance. It serves as a fair metric since the standard deviation of the execution time distribution of the ten runs is much lower than the average execution time. Median execution time is then used to estimate performance in GFLOPs per second based on the number of FLOPs for SpMV.

6.4.2 Baseline SpMV

Figure 6.5 shows the relationship between the Kronecker graph generation parameters (K power and initiator matrix values) and baseline SpMV performance. Each line represents a set of initiator matrix values. Due to the large number of permutations of these values, they were omitted from the figure legend. Some observations from this figure are:

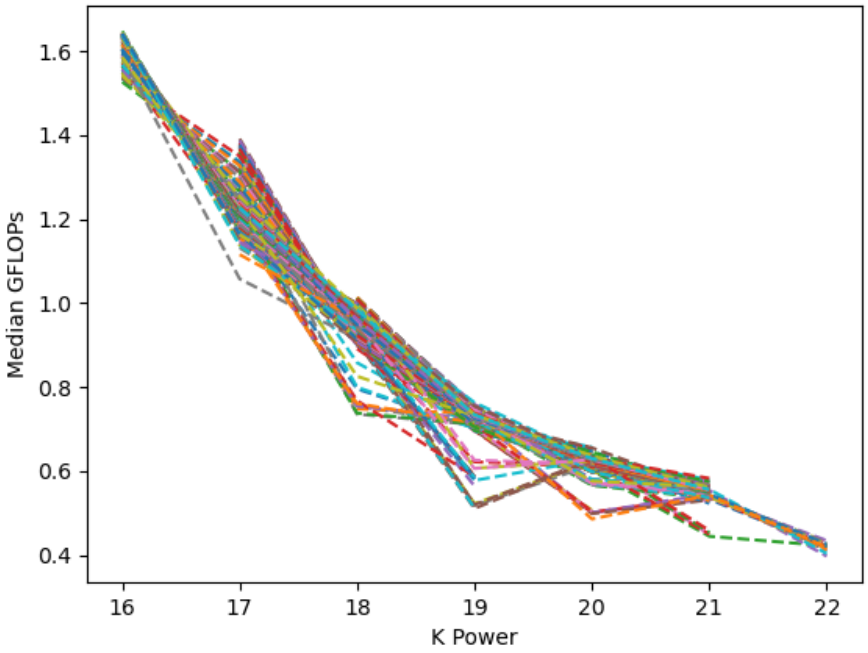


Figure 6.5: Kronecker Power (horizontal axis) versus Performance in GFLOPs (vertical axis) for baseline TACO SpMV with no blocking using CSR format

- Performance decreases as the Kronecker power increases. This is because as the Kronecker power increases, the size of the graph (matrix) gets bigger, and it does not fit in any level of cache. Hence, most values need to be fetched from main memory. This applies to the dense vectors as well, as they have to match the dimensions of the sparse matrix so that the multiplication is valid.

- Varying the initiator matrix values for the same Kronecker power has varying effect. This is because the summation of the initiator matrix values directly affects the total number of non-zeros in the sparse matrix, and hence the size of the matrix. The gap between the best and worst performing initiator matrix values however varies based on the K power, since K power determines the total number of nodes in the graph. The Kronecker power and the summation of the initiator matrix values together determine the size, sparsity, and the sparsity pattern of the matrix. Table 6.2 shows the gap between the best and the worst initiator matrix values for each of the evaluated

Kronecker power values.

Table 6.2: Performance Gap between Worst and Best Performing Initiator Matrix Values for different Kronecker Power values for Baseline SpMV in TACO using CSR. Time in milliseconds.

K Power	Worst Values	Worst Time	Best Values	Best Time
16	0.899 0.537 0.537 0.584	4.06704	0.899 0.537 0.526 0.484	2.13947
17	0.899 0.537 0.537 0.583	14.9136	0.964 0.437 0.437 0.484	2.85515
18	0.899 0.537 0.537 0.565	50.1947	0.899 0.437 0.437 0.484	5.09403
19	0.899 0.537 0.537 0.583	189.332	0.899 0.437 0.437 0.484	14.3231
20	0.899 0.537 0.52 0.484	221.799	0.899 0.437 0.437 0.484	39.3673
21	0.899 0.537 0.437 0.484	252.357	0.899 0.438 0.437 0.484	99.1911
22	0.908 0.437 0.437 0.484	324.235	0.9 0.437 0.437 0.484	281.922

6.4.3 Blocked SpMV

To generate the blocked (tiled) variant of SpMV, we provide a schedule to TACO. The schedule starts by splitting (stripmining) each loop into two loops, with a step of number of rows per block and number of columns per block respectively. This step introduces a new point loop under each original loop. Then, the two innermost loop nests (after splitting) are permuted. The two variables in this schedule are: the number of rows per block nb and the number of columns per block mb. We evaluate all power of two permutations for both nb and mb up to 4096. The performance distribution of Blocked SpMV versus baseline SpMV is shown in Figure 6.6. Results show that blocking for CSR using this schedule in TACO severely worsens SpMV performance as compared to baseline. This is due to the overhead of adding additional loops while searching for non-zeros within block boundaries (nb and mb) which is not straightforward in sparse storage formats such as CSR. This additional overhead is not amortized by the re-use in the output dense vector.

6.4.4 Tensorized SpMV

We evaluated the performance of raising SpMV tensor-times-matrix in different ways:

- Different storage formats for each of the four dimensions.

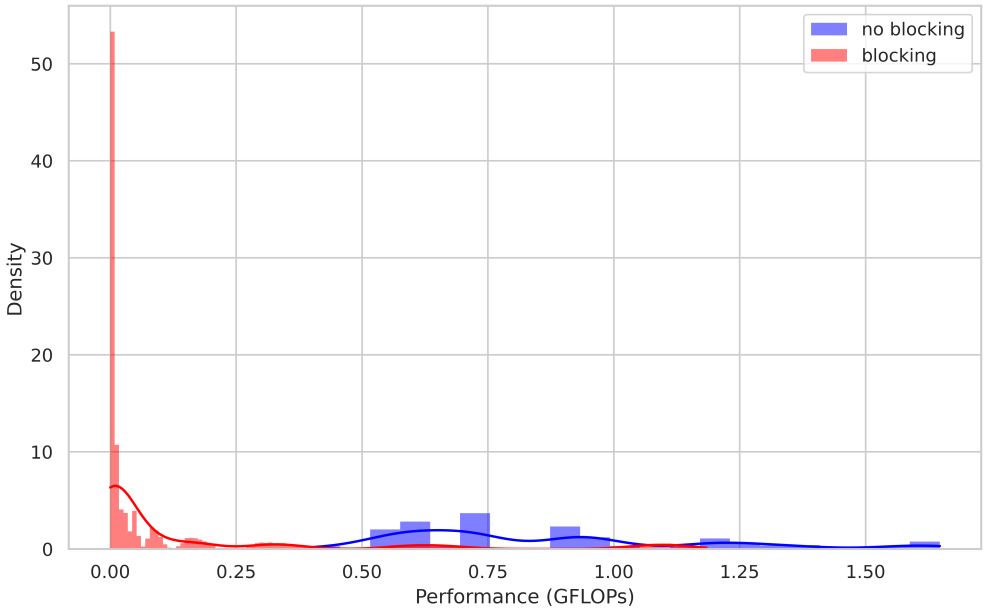


Figure 6.6: Performance Distributions for both baseline SpMV (no blocking) and Blocked SpMV. Blocked SpMV is generated by providing a split then re-order schedule to TACO.

- Different indices order in the data layout transformation pre-processing step.
- Different schedules to reorder the loops in TACO.
- Different index variable orders in the tensor expression (no schedule provided).

Table 6.3 shows the relative performance of different evaluated configurations as compared to baseline SpMV. For the tensor storage format, each dimension can be dense (d) or sparse (s). DLT refers to Data Layout Transformation, where 2D matrix files are converted to 4D tensor files. Schedule in the table refers to a re-ordering schedule that swaps the order of the second and third loop nests. Manual reorder refers two simply changing the order of the index variables in the tensor expression without providing a schedule.

We compare tensor performance to the performance of its corresponding matrix it was generated from. For example, matrix `mtx` is used to generate many tensors `tns1, tns2, tns3, ...` each reflecting a different block size. When evaluating the performance of the tensor-matrix multiplication operation on tensor `tns1`, we compare it to SpMV performance on matrix `mtx`. We notice that providing a re-ordering schedule for the 4D tensor results

in TACO generating incorrect code, which fails to compile. For the rest of configurations, TACO produces code that compiles and runs (in many cases, but for some sizes still fails to run). However, all variants exhibit performance which is worse compared to the 2D matrix SpMV performance. The magnitude at which the performance is worse varies significantly based on the used block size.

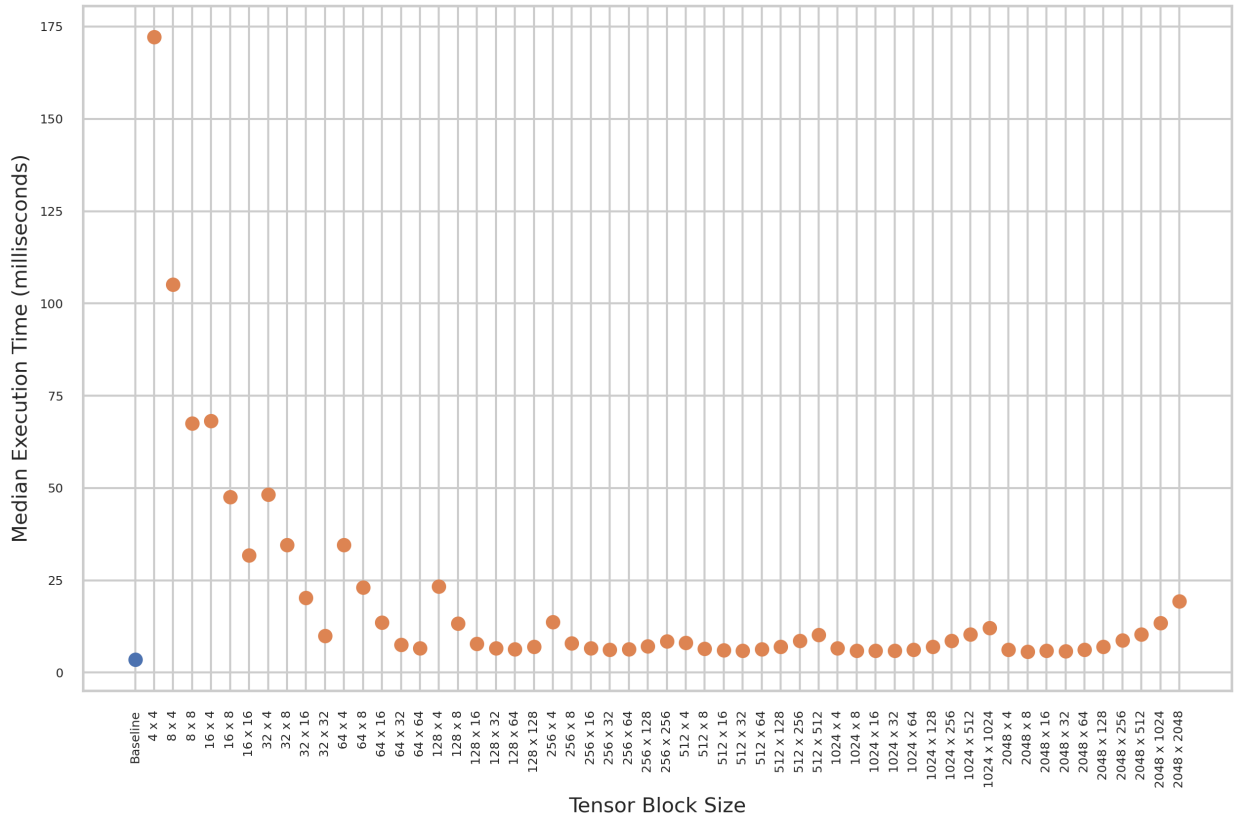


Figure 6.7: Performance comparison between Tensorized multiplication with manual indices order (io, ii, jo, ji) for different block sizes generated from a K16 matrix with the initiator values `[0.899 ; 0.537 ; 0.537 ; 0.561]`, and baseline SpMV using TACO. Storage Format for 4D Tensors is *ssss*

Taking one random K16 matrix as an example with the following initiator matrix values : `[0.899 ; 0.537 ; 0.537 ; 0.561]`, the total number of DLT2 + manual reorder attempted to evaluate was 100. 45 of these failed to run by TACO. Figure 6.7 shows median execution time (milliseconds) comparison between tensorized and baseline. We can observe that baseline SpMV is consistently better than tensorized, but the gap increases when block

sizes are small (4×4 is the worst among those evaluated).

Table 6.3: Relative Performance Results for Different Tensorized Configuration in TACO as compared to a baseline SpMV.

Storage Format	DLT1 (io, jo, ii, ji)	DLT2 (io, ii, jo, ji)	DLT1 + schedule	DLT2 + schedule	DLT1 + manual reorder	DLT2 + manual reorder
ssss	Worse	Worse	All Failed	All Failed	Worse	Worse
dsss	Worse	Worse	All Failed	All Failed	Worse	Worse
sdss	Worse	Worse	All Failed	All Failed	Worse	Worse
ssds	Worse	Worse	All Failed	All Failed	Worse	Worse

6.4.5 Proposed Library Experiments

For the experimental setup, we use the same setup for TACO shown in Table 6.1, but for the SpMV backend we use MKL version 2023.1.0. In order to find a range of high-performing tile sizes for the system, we evaluate the performance of SpMV using MKL (no tiling) for a sample initiator matrix `[0.899 0.537; 0.526 0.484]` for all Kronecker powers varying from 1 to 20 as shown in Figure 6.8. We can observe that for the system configuration, peak performance is achieved between K power = 13 and 17. Hence, we choose this range of size to evaluate as tile sizes for our library implementation experiments.

For different Kronecker powers from 17 to 20, we evaluate the range of peak tile sizes obtained from the aforementioned experiment, using the same initiator matrix values. Figrue 6.9 shows performance results for using our library to implement data-side tiling using different tile sizes. The main observations from this set of experiments is that increasing the number of columns per block up to the maximum evaluated sizes, improves performance, with tiles having the maximum number of columns per tile showing similar or better performance to no tiling. This is mainly due to the high sparsity of the Kronecker graphs; fewer columns per block mean fewer non-zeros to operate on, not justifying the additional loop overhead and metadata overhead due to maintaining a large number of tiles. We can also observe that with increasing the Kronecker power, the performance gap between the best tiling configuration and the no-tiling configuration increases in favor of the tiling. For K=17, the best tiling configuration (131072×131072) achieves similar performance to no tiling. For K=18, the best tiling configuration (131072×262144) achieves 1.05% better performance

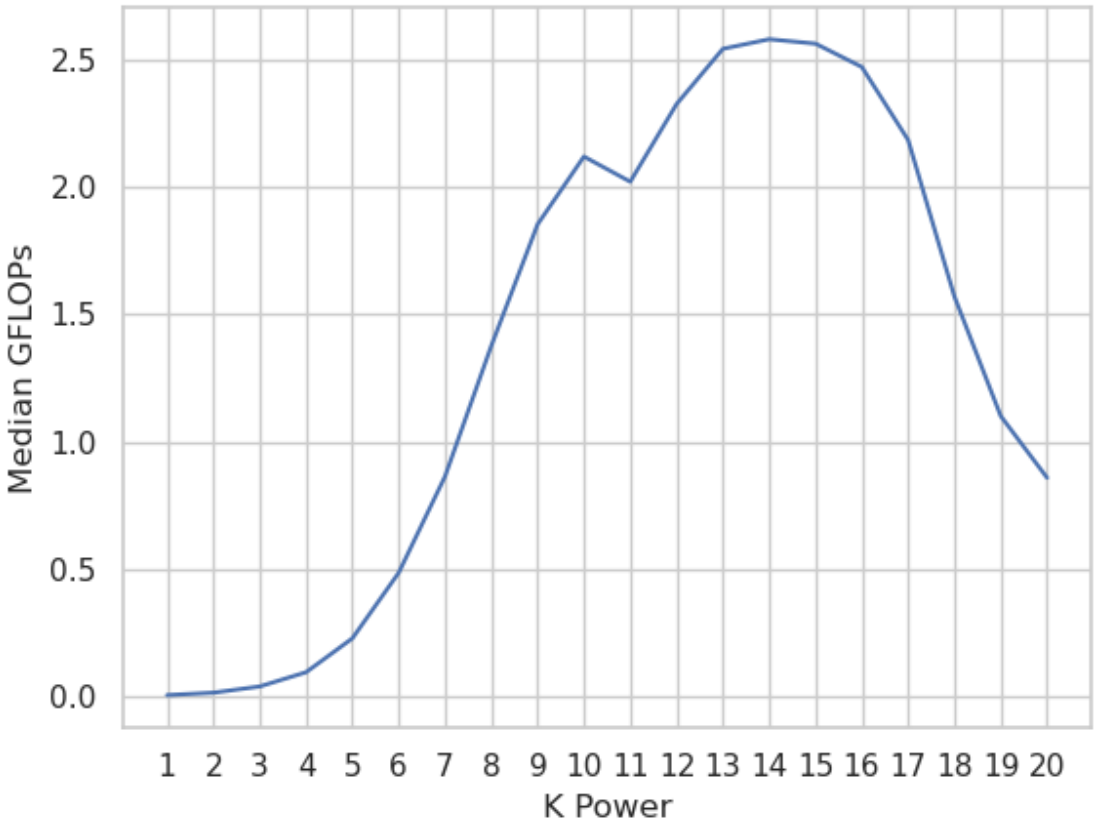
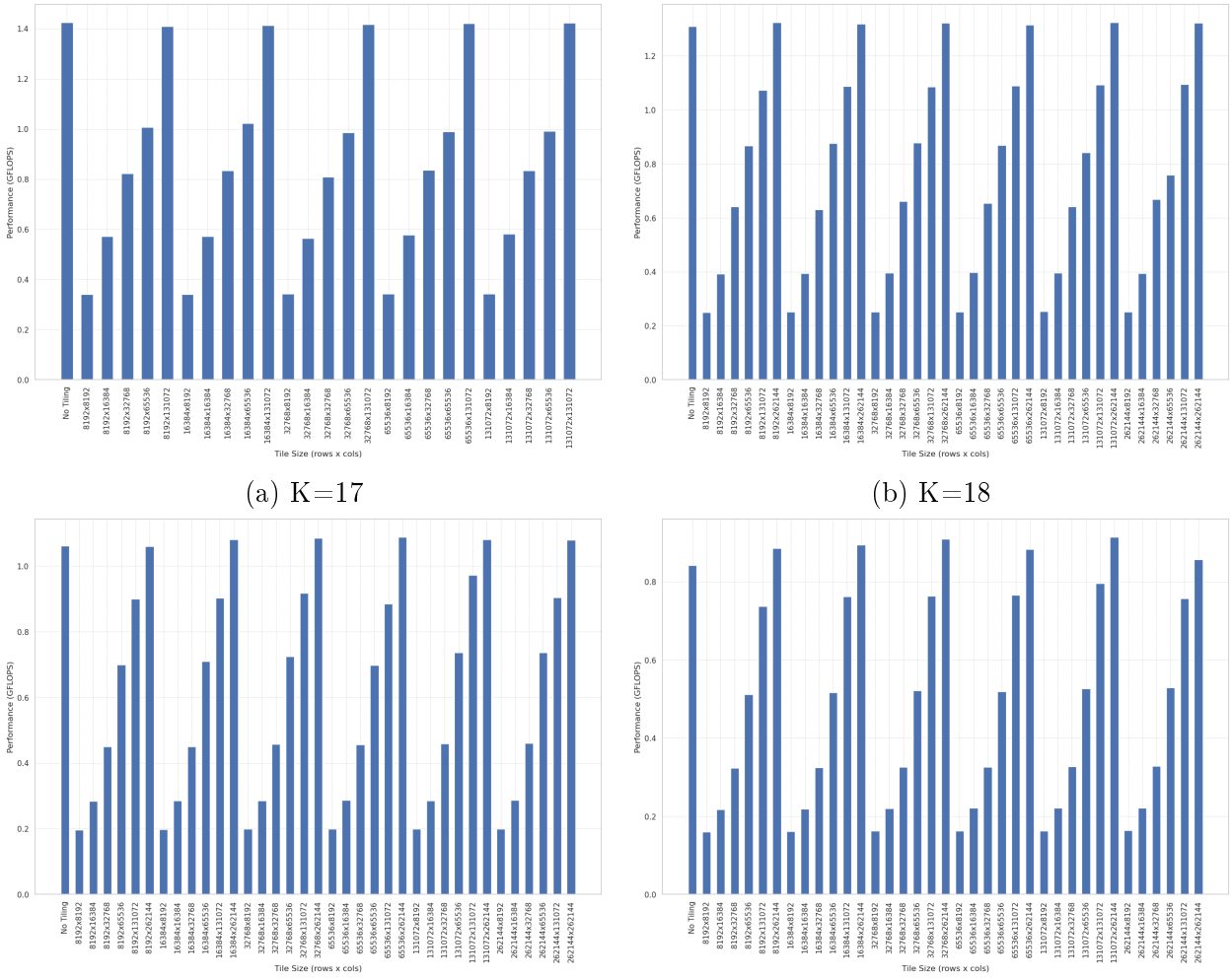


Figure 6.8: MKL SpMV performance in GFLOPs (vertical axis) versus Kronecker Power (x-axis) for the Kronecker initiator matrix `[0.899 0.537; 0.526 0.484]`



(a) K=17

(b) K=18

(c) K=19

(d) K=20

Figure 6.9: Performance (GFLOPS) of our library's tiled SpMV with different tile configurations using MKL for (a)K=17, (b)K=18, (c)K=19, and (d)K=20 for the initiator matrix [0.899 0.537; 0.526 0.484].

than no tiling. For K=19, the gap is 2.56% better in favor of 65536×262144 over no tiling. For K=20, the tiling configuration 131072×262144 is 8.6% better than no tiling. This is mainly due to the fact that increasing Kronecker graph power (sparse matrix dimensions) increases the dense vector dimension as well. When the dense vector increases significantly in space, the effect of re-use via tiling becomes more pronounced.

6.5 Summary

In this chapter we explored a set of high-level optimizations for SpMV using TACO as a code generator. Example optimizations include tiling (blocking) 2D sparse matrices, raising matrices to higher dimensional 4D tensors through data layout transformation and adjusting loop iteration ordering. For each of the optimizations, we evaluated plenty of sparse storage formats for the sparse matrix/tensor. As expected, tiling over a 2D matrix incurred an additional overhead and resulted into worse performance as compared to a baseline CSR SpMV with no blocking. The reason being tiling includes scanning the sparse storage format for dense blocks within the row/column block size boundaries. Raising the matrix to higher dimensional tensors did not show any performance improvement to baseline SpMV. It is possible that TACO lacks the mechanisms to generate efficient code for this specific situation, which was clear when it generated incorrect code for an instance where a re-ordering schedule was provided for the 4D tensor multiplication case. Changing the sparse format resulted in performance changes, however, no performance benefit over the baseline implementation was observed. Our results show the need for more specialized tools to perform such optimizations in the sparse world, as existing tools falls short once the number of dimensions increase and specialized optimizations need to be tailored. We introduced a novel header-only library to perform tiling on the data itself using CSR format, achieving up to 1.09× better performance than no tiling. This shows the potential of data layout transformation and opens the door to further improve performance gains by carefully reducing metadata overhead, exploring other sparse storage formats (in addition to CSR), evaluating the effects of the approach on a wide range of Kronecker graph parameters (initiator matrix values and Kronecker power), and exploring additional optimizations.

Chapter 7

Conclusion

This dissertation has embarked on a journey to bridge the gap between sparse matrix computation and graph models, elucidating the intricate interplay between these two pivotal areas in computational science. Through a comprehensive exploration encompassing the optimization of sparse operations, leveraging graph neural networks (GNNs) for identifying sparse matrix structures, and the development of frameworks for analyzing the robustness and performance of graph models, this work has contributed significantly to advancing our understanding and capabilities in handling sparse matrices and graphs.

7.1 Main Contributions

The dissertation's contributions are multifaceted, addressing both theoretical and practical aspects of sparse matrix computations and graph models. Firstly, we introduced an innovative approach using GNNs to accurately identify the structure of sparse matrices. This methodology not only enhances our ability to predict matrix properties but also opens new avenues for optimizing sparse matrix storage and operations. Furthermore, we developed frameworks that allow for the analysis of graph models' robustness and performance, providing valuable tools for researchers and practitioners to evaluate and improve their graph-based algorithms and applications. Additionally, this dissertation explored a set of high-level

optimizations for sparse computations to evaluate its efficiency and to assess the quality of current code generation tools for sparse kernels.

7.2 Future Directions

While this dissertation has made strides in bridging sparse matrix computation and graph models, several exciting avenues for future research have emerged. One potential direction involves the further refinement and expansion of the GNN-based framework for identifying sparse matrix structures. Exploring additional features and network architectures could yield even more accurate and efficient predictions. Working on a parallel framework for regression to predict the numeric parameters used to generate the input sparse matrix (after classifying its generator) is another important future work to move a step closer to predicting the optimal sparse format. Additionally, the developed frameworks for analyzing graph models' robustness and performance present opportunities for integration with machine learning techniques to automate and enhance the decision-making process regarding the choice of sparse matrix formats and optimization strategies. As a result of adopting graph models, working on the efficient generation of sparse data is a crucial direction to investigate. Optimized generators that make use of system heterogeneity can assist in generating large-scale sparse data more efficiently. In addition, as observed in Chapter 5, in a typical SpMV workflow, sparse matrix file loading and parsing takes up the majority of the end-to-end system wall time. Hence, introducing efficient file formats for sparse data helps minimizing this significant portion of the workflow execution. Additionally, given the shortage of specialized efficient sparse tensor algebra tools in capturing and improving sparse performance, new tools are needed to fill in this gap. The ultimate goal is to have a model that takes as input graph model and generation parameters, and comes up (as output) with the optimal optimization strategy for a given sparse operation (e.g., SpMV).

7.3 Final Thoughts

In conclusion, this dissertation stands as a testament to the profound interconnectedness of sparse matrix computations and graph models, highlighting the importance of continued research and development in these areas. The contributions made herein not only advance our theoretical understanding but also provide practical tools and frameworks that can be leveraged across various domains of computational science. As we look to the future, it is clear that the exploration of sparse matrices and graph models will continue to be a rich and fruitful field of inquiry, promising to unlock new discoveries and innovations at the nexus of computation, mathematics, and data science.

Bibliography

[1] Khaled Abdelaal and Martin Kong. Tile size selection of affine programs for gpgpus using polyhedral cross-compilation. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 13–26, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Khaled Abdelaal and Richard Veras. A framework for analyzing the robustness of graph models. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023.

[3] Khaled Abdelaal and Richard Veras. Observe locally, classify globally: Using gnns to identify sparse matrix structure. In Ignacio Rojas, Gonzalo Joya, and Andreu Catala, editors, *Advances in Computational Intelligence*, pages 149–161, Cham, 2023. Springer Nature Switzerland.

[4] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.

[5] AMD. rocsparse documentation https://rocsparse.readthedocs.io/en/latest/, 2024. [Online; accessed 11-March-2024].

[6] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 781–792, 2014.

[7] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P. Sadayappan. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 273–282, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[9] Scott Beamer, Krste Asanović, and David Patterson. Gail: The graph algorithm iron law. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures*

and Algorithms, IA³ '15, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.

[11] Mehmet Belgin, Godmar Back, and Calvin J. Ribbens. Pattern-based sparse matrix representation for memory-efficient smvm kernels. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 100–109, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[13] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. Sparse matrix format selection with multiclass svm for spmv on gpu. In *2016 45th International Conference on Parallel Processing (ICPP)*, pages 496–505, 2016.

[14] Aydin Buluc and John R Gilbert. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11. IEEE, 2008.

[15] Chen Cai and Yusu Wang. A simple yet effective baseline for non-attribute graph classification. *arXiv preprint arXiv:1811.03508*, 2018.

[16] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM Comput. Surv.*, 38(1):2–es, jun 2006.

[17] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 115–126, New York, NY, USA, 2010. Association for Computing Machinery.

[18] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *SIGPLAN Not.*, 45(5):115–126, jan 2010.

[19] Hejie Cui, Zijie Lu, Pan Li, and Carl Yang. On positional and structural node features for graph neural networks on non-attributed graphs. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 3898–3902, New York, NY, USA, 2022. Association for Computing Machinery.

[20] Hoang-Vu Dang and Bertil Schmidt. The sliced coo format for sparse matrix-vector multiplication on cuda-enabled gpus. *Procedia Computer Science*, 9:57–66, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012.

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

[22] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

[23] Nan Ding and Samuel Williams. An instruction roofline model for gpus. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, 2019.

[24] Mikhail Drobyshevskiy and Denis Turdakov. Random graph modeling: A survey of the concepts. *ACM Comput. Surv.*, 52(6), dec 2019.

[25] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. Alphasparse: Generating high performance spmv codes directly from sparse matrices. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[26] Paul Erdős, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. math. inst. hung. acad. sci*, 5(1):17–60, 1960.

[27] Xiaowen Feng, Hai Jin, Ran Zheng, Kan Hu, Jingxiang Zeng, and Zhiyuan Shao. Optimization of sparse matrix-vector multiplication with variant csr on gpus. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 165–172, 2011.

[28] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[29] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on gpgpus. *ACM Trans. Math. Softw.*, 43(4), jan 2017.

[30] Basilio B. Fraguela, Jia Guo, Ganesh Bikshandi, María J. Garzarán, Gheorghe Almási, José Moreira, and David Padua. The hierarchically tiled arrays programming approach. In *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems*, LCR '04, page 1–12, New York, NY, USA, 2004. Association for Computing Machinery.

[31] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training compression for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[32] Dimitrios Galanopoulos, Panagiotis Mpakos, Petros Anastasiadis, Nectarios Koziris, and Georgios Goumas. Invited paper: An artificial matrix generator for multi-platform spmv performance analysis. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 574–577, 2023.

[33] Alan George. Sparse matrix aspects of the finite element method. In R. Glowinski and J. L. Lions, editors, *Computing Methods in Applied Sciences and Engineering*, pages 3–22, Berlin, Heidelberg, 1976. Springer Berlin Heidelberg.

[34] Saeid Ghafouri and Seyed Hossein Khasteh. A survey on exponential random graph models: an application perspective. *PeerJ Computer Science*, 6, 2020.

[35] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[36] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3, 2010.

[37] Xiaojie Guo and Liang Zhao. A systematic survey on deep generative models for graph generation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5):5370–5390, 2022.

[38] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. Alto: adaptive linearized storage of sparse tensors. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 404–416, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks, 2021.

[40] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 300–314, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics (TOG)*, 38(6):1–16, 2019.

[42] Khaled Z Ibrahim, Samuel Williams, and Leonid Oliker. Performance analysis of gpu programming models using the roofline scaling trajectories. In *International Symposium on Benchmarking, Measuring and Optimization*, pages 3–19. Springer, 2019.

[43] Ivan imecek, D. Langr, and P. Tvrdík. Space-efficient sparse matrix storage formats for massively parallel systems. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 54–60, 2012.

[44] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 319–329, New York, NY, USA, 1988. Association for Computing Machinery.

[45] Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. An extended compression format for the optimization of sparse

matrix-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 24(10):1930–1940, 2013.

[46] Jeremy Kepner. Analytic theory of power law graphs. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2008.

[47] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy Mattson, and Jose Moreira. Mathematical foundations of the graphblas. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9, 2016.

[48] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

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

[50] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[51] Elias Konstantinidis and Yiannis Cotronis. A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37–56, 2017.

[52] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Csx: An extended compression format for spmv on shared memory systems. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 247–256, New York, NY, USA, 2011. Association for Computing Machinery.

[53] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R. Bishop. Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1696–1702, 2012.

[54] Daniel Langr and Pavel Tvrdík. Evaluation criteria for sparse matrix storage formats. *IEEE Transactions on Parallel and Distributed Systems*, 27(2):428–440, 2016.

[55] Christoph Lehnert, Rudolf Berrendorf, Jan P. Ecker, and Florian Mannuss. Performance prediction and ranking of spmv kernels on gpu architectures. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing*, pages 90–102, Cham, 2016. Springer International Publishing.

[56] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, mar 2010.

[57] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 631–636, New York, NY, USA, 2006. Association for Computing Machinery.

[58] Jure Leskovec and Christos Faloutsos. Scalable modeling of real graphs using kronecker multiplication. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, page 497–504, New York, NY, USA, 2007. Association for Computing Machinery.

[59] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2–es, mar 2007.

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

[61] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In Alípio Mário Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, and João Gama, editors, *Knowledge Discovery in Databases: PKDD 2005*, pages 133–145, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[62] Carson K. Leung, Ryan Middleton, Adam G. M. Pazdor, and Yeyoung Won. Mining 'following' patterns from big but sparsely distributed social network data. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 916–919, 2018.

[63] J. Li, J. Sun, and R. Vuduc. Hicoo: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 238–252, 2018.

[64] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui 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*, PLDI '13, page 117–126, New York, NY, USA, 2013. Association for Computing Machinery.

[65] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication. *SIGPLAN Not.*, 48(6):117–126, jun 2013.

[66] Kenli Li, Wangdong Yang, and Keqin Li. Performance analysis and optimization for spmv on gpu using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):196–205, 2015.

[67] Yishui Li, Peizhen Xie, Xinhai Chen, Jie Liu, Bo Yang, Shengguo Li, Chunye Gong, Xinbiao Gan, and Han Xu. Vbsf: a new storage format for simd sparse matrix–vector multiplication on modern processors. *The Journal of Supercomputing*, 76:2063–2081, 2020.

[68] Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Will Hamilton, David K Duvenaud, Raquel Urtasun, and Richard Zemel. Efficient graph generation with graph recurrent attention networks. *Advances in neural information processing systems*, 32, 2019.

[69] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381, 2014.

[70] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 339–350, New York, NY, USA, 2015. Association for Computing Machinery.

[71] André Lopes, Frederico Pratas, Leonel Sousa, and Aleksandar Ilic. Exploring gpu performance, power and energy-efficiency bounds with cache-aware roofline modeling. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 259–268, 2017.

[72] Marco Maggioni and Tanya Berger-Wolf. Adell: An adaptive warp-balancing ell format for efficient sparse matrix-vector multiplication on gpus. In *2013 42nd International Conference on Parallel Processing*, pages 11–20, 2013.

[73] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 678–689, 2016.

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

[75] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. Fastspmm: An efficient library for sparse matrix matrix product on gpus. *The Computer Journal*, 57(7):968–979, 2014.

[76] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[77] Arjun S Ramani, Nicole Eikmeier, and David F Gleich. Coin-flipping, ball-dropping, and grass-hopping for generating random graphs from matrices of edge probabilities. *SIAM Review*, 61(3):549–595, 2019.

[78] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

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

[80] Jonathan Schwarz, Siddhant Jayakumar, Razvan Pascanu, Peter E Latham, and Yee Teh. Powerpropagation: A sparsity inducing weight reparameterisation. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 28889–28903. Curran Associates, Inc., 2021.

[81] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P. Sadayappan. Automatic selection of sparse matrix representation on gpus. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 99–108, New York, NY, USA, 2015. Association for Computing Machinery.

[82] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. An in-depth study of stochastic kronecker graphs. In *2011 IEEE 11th International Conference on Data Mining*, pages 587–596, 2011.

[83] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. An in-depth analysis of stochastic kronecker graphs. *J. ACM*, 60(2), may 2013.

[84] P. Stathis, S. Vassiliadis, and S. Cotofana. A hierarchical sparse matrix storage format for vector processors. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 8 pp.–, 2003.

[85] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010.

[86] Bor-Yiing Su and Kurt Keutzer. Clspmv: A cross-platform opencl spmv framework on gpus. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 353–364, New York, NY, USA, 2012. Association for Computing Machinery.

[87] Mohammed Suhail, Abhay Mittal, Behjat Siddiquie, Chris Broaddus, Jayan Eledath, Gerard Medioni, and Leonid Sigal. Energy-based learning for scene graph generation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 13936–13945, 2021.

[88] Guangming Tan, Junhong Liu, and Jiajia Li. Design and implementation of adaptive spmv library for multicore and many-core architecture. *ACM Trans. Math. Softw.*, 44(4), aug 2018.

[89] Charles Van Loan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[90] R. Veras, T. M. Low, and F. Franchetti. A scale-free structure for power-law graphs. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2016.

[91] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. *Intel Math Kernel Library*, pages 167–188. Springer International Publishing, Cham, 2014.

[92] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):1–40, 2016.

[93] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[94] M. Wolfe. More iteration space tiling. In *Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, Supercomputing '89, page 655–664, New York, NY, USA, 1989. Association for Computing Machinery.

[95] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[96] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[97] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. Yaspmv: Yet another spmv framework on gpus. *SIGPLAN Not.*, 49(8):107–118, February 2014.

[98] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system. *Concurrency and Computation: Practice and Experience*, 32(20):e5547, 2020.

[99] Deqing Yang, Ziyi Wang, Junyang Jiang, and Yanghua Xiao. Knowledge embedding towards the recommendation with sparse user-item interactions. In *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '19, page 325–332, New York, NY, USA, 2020. Association for Computing Machinery.

[100] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27168–27183. Curran Associates, Inc., 2022.

[101] Minji Yoon, Yue Wu, John Palowitch, Bryan Perozzi, and Russ Salakhutdinov. Graph generative model for benchmarking graph neural networks. 2023.

[102] Aya Zaki, Mahmoud Abdallah Attia, Doaa Hegazy, and Safaa El-Sayed Amin. Comprehensive survey on dynamic graph models. *International Journal of Advanced Computer Science and Applications*, 7, 2016.

[103] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng 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*, PPoPP '18, page 94–108, New York, NY, USA, 2018. Association for Computing Machinery.

[104] Cong Zheng, Shuo Gu, Tong-Xiang Gu, Bing Yang, and Xing-Ping Liu. Biell: A bisection ellpack-based storage format for optimizing spmv on gpus. *Journal of Parallel and Distributed Computing*, 74(7):2639–2647, 2014. Special Issue on Perspectives on Parallel and Distributed Processing.