A Framework for Analyzing the Performance of Sparse Matrix and Graph Operations

Khaled Abdelaal
School of Computer Science
The University of Oklahoma
Norman, OK, USA
khaled.abdelaal@ou.edu

Richard Veras
School of Computer Science
University of Oklahoma
Norman, OK, USA
richard.m.veras@ou.edu

***Abstract*—Thorough performance analysis is critical when developing new algorithms, implementation, optimizations, and transformations for compute intensive operations. This is especially true for operations that serve as the basic building blocks for scientific libraries, such as the Basic Linear Algebra Subprograms (BLAS). In the case of dense computations like the BLAS it is sufficient to know the dimensions and strides of one's dataset to predict the performance from prior runs on similar hardware. Thus, it is customary in the evaluation of these types of dense routines to provide a performance profile that sweeps through problem sizes to characterize the behavior of an implementation. However, for operations over sparse matrices, dimensions do not alone relay sufficient information to predict the performance. The standard approach for evaluating the performance of operations over sparse data is through the performance evaluation of the operation over a canonical set of sparse matrices. However, the ability to generalize these results beyond those matrices to new datasets is limited.**

In this paper, we present a framework to evaluate the performance of operations using sparse matrix and graph models. It visualizes the performance using parameterized graph models and evaluates the efficacy of using different sets of parameters to describe the input matrix and observe performance on different implementations. The framework also assesses the tolerance of the performance of different implementations to multiple sources of noise induced to the input data on the performance, and to the model parameters themselves. Our framework is fully modular and extensible in the sense that users can seamlessly plug in their different graph model generators, with different parameter sets, operation implementations, and additional types of noise to inject and evaluate. Our framework also takes into account system-wide performance rather than kernel-only performance.

***Index Terms*—Sparse Matrix, Graph, Performance Evaluation and Visualization, Kronecker Graphs**

I. 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 a 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. Further, if the analysis of the complex relationship in the data can be analyzed as sequence 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 are challenging because the structure of the sparse data, the implementation of the operation, and the architecture of the target have tremendous bearing over 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 relating performance against structural features of sparse data.

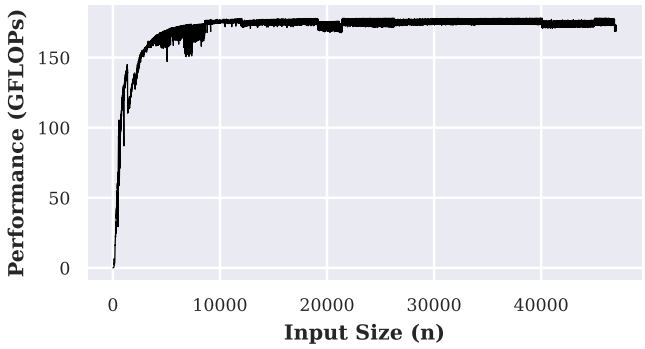


Fig. 1: Double Precision Dense General Matrix-Vector Multiplication Performance using cuBLAS on NVIDIA RTX A6000 GPU as a function of input size

For dense matrices, performance evaluation and visualization is straightforward. Figure 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 2 shows a corresponding performance evaluation for SpMV using cuSparse. Sample matrices from the SuiteSparse collection [1] are evaluated. Table I lists the properties of these matrices. The horizontal axis in Figure

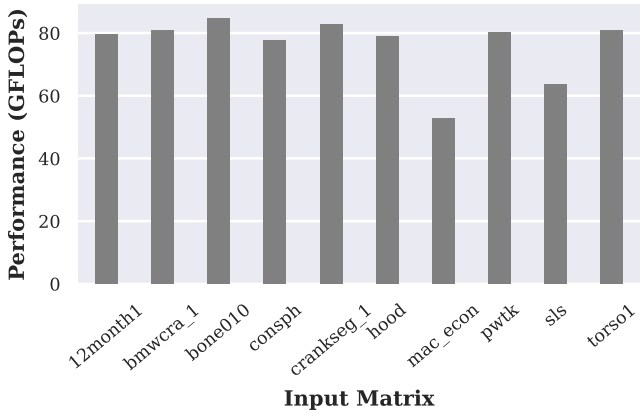


Fig. 2: Double Precision Sparse Matrix-Vector Multiplication Performance using cuSparse on NVIDIA RTX A6000 GPU for a selected set of matrices from SuiteSparse, using the COOrdinate data representation.

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 not 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 matrices 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 [2], face a set of limitations adapting to sparse data workloads with regard to using a representative feature/parameter being related to performance [3]. Also, the dependence on discrete sets of graphs/sparse matrices for benchmarking such operations [1], [4], [5] 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 employs 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. 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 derive optimization decisions.

TABLE I: 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

The main contributions of this work is 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 paper is organized as follows: Section II introduced the necessary background and discusses related work, its limitations, and how they motivate the proposal of our framework to address them. Section III details the description of our proposed framework. Section IV shows a set of experiments conducted through our framework and discusses the results and observations. Finally, Section V summarizes the findings of our paper.

II. Background and Related Work

A. 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 into linear algebra operations [6]. 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) [7]. The implementation of an alogrithm 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) [8] for CPU, NVIDIA cuSparse [9], and AMD rocSPARSE [10].

B. Graph Models for Sparse Data

Many performance evaluation techniques for sparse data emerged as a response to the generation of such data from engineering and physics problem types. However, many real data are more accurately represented by large scale-free synthetic data that follow a power-law distribution such as Kronecker graphs [11], [12] or a combination of Kronecker + Random [13]. 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

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

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 [14].

Additionally, many graph models have been proposed to generate synthetic data that have similar properties to real graphs [15]–[21]. Frameworks have been proposed to classify input sparse data/graph into one of these models [22], and to evaluate the robustness of such graph models in terms of sensitivity of the graph structure to model parameters and noise [23].

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 [12]) 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.

C. 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) [2] 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).

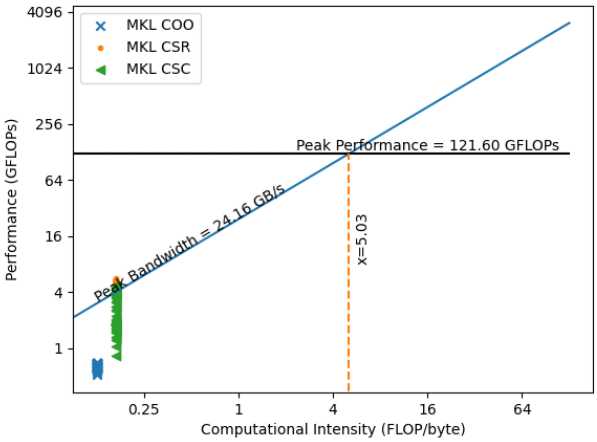


Fig. 3: Performance Evaluation of SpMV in MKL for different sparse formats (COO, CSR, and CSC) on a set of sparse matrices using the Roofline model. Since the arithmetic intensity is imposed by the sparse data format, little insights are povided on how to optimize performance.

The roofline model [3] 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 (FLOPs). Additionally, many derived variants of the roofline model have been developed to accommodate for different memory hierarchy assumptions [24], capture the hardware changes in modern architectures such as GPUs [25]–[27], and work on finer granularity than a FLOP/byte such as instruction/transaction [28]. However, the application of roofline model for applications on sparse data is less than adequate. Assuming we are trying to optimize SpMV operation, and we evaluate the performance of each algorithm using operational

(arithmetic) intensity, and FLOPs. Since SpMV 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 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 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.

D. 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 [4], SuiteSparse Matrix Collection [1], and GAP [5]. 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.

III. 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 II, 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 finding a relationship between the graph generation mechanism and the resulting performance for 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.

Algorithm 1 General Framework Description

```
for param in model_gen_params do
  for val in param_legal_values do
    new_params = val∪(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 ∪ Gs_NA ∪ 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()
```

A. High-Level Overview

A general overview of the framework is shown in Algorithm 1. 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. 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 by adding small random values (`n0`) from a uniform distribution with a maximum threshold of `nA_thresh` and a user-defined step of `nA_step` to the model generation parameters, producing a new set of Graphs: Gs_{NA}. 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 use 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.

B. 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 harness tests. 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.

IV. 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 II shows the configuration for the system used in our experiments.

TABLE II: System Configuration

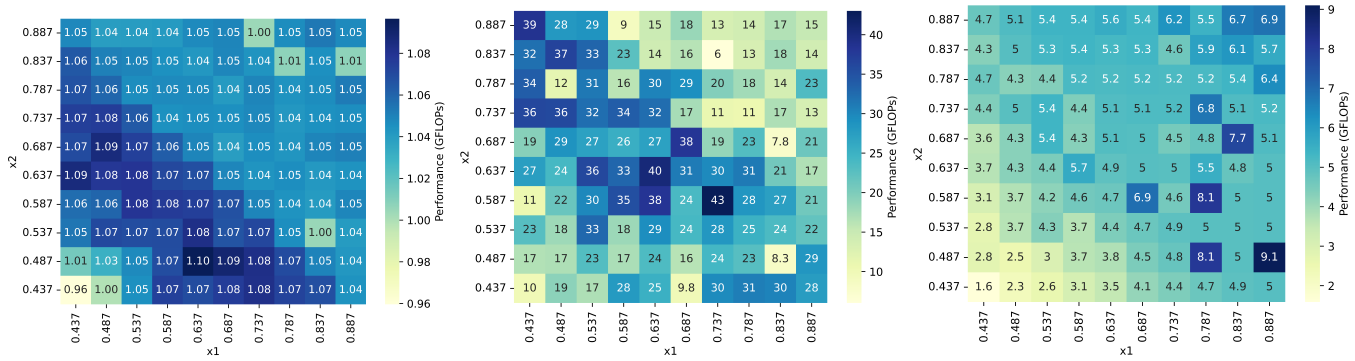
Component	Specification
GPU	NVIDIA RTX H100
GPU Memory	80 GB
CUDA Version	12.0
CPU	Intel Xeon Gold 6338
CPU Sockets	2
CPU Cores	128
MKL Version	2022.1.0
Main Memory	256 GB DDR4

A. 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.

1) 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 [12] for the High Energy Physics - Phenomenology Collaboration (CA-HEP-PH) Graph [29] 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. Figure 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 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.

2) 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



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

Fig. 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.

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.

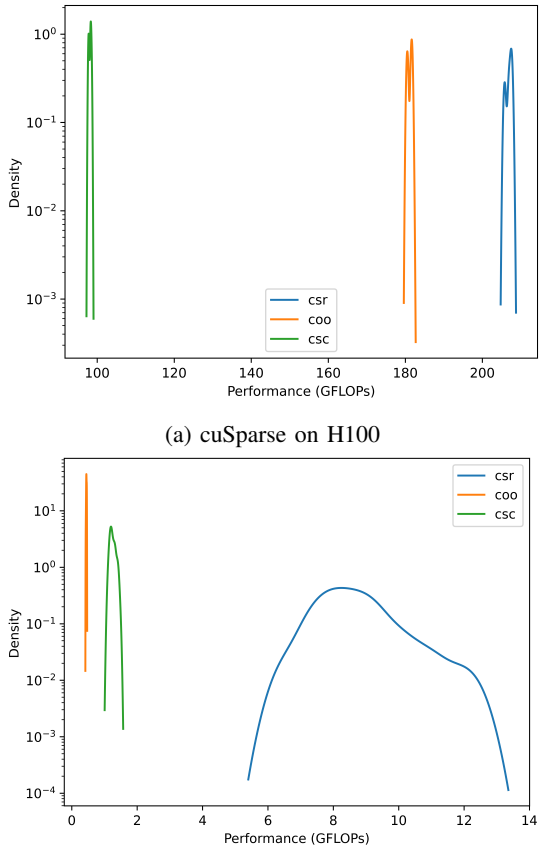
Figure 5 shows the performance results for this experiment. Instead of using heatmaps to observe the performance variation across a grid of different initiator matrix values, we use Kernel Density Estimation (KDE) plots to visualize the frequency of different performance ranges (in GFLOPs). The horizontal 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 5a and 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.

B. 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 3 types of graphs: 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 III.

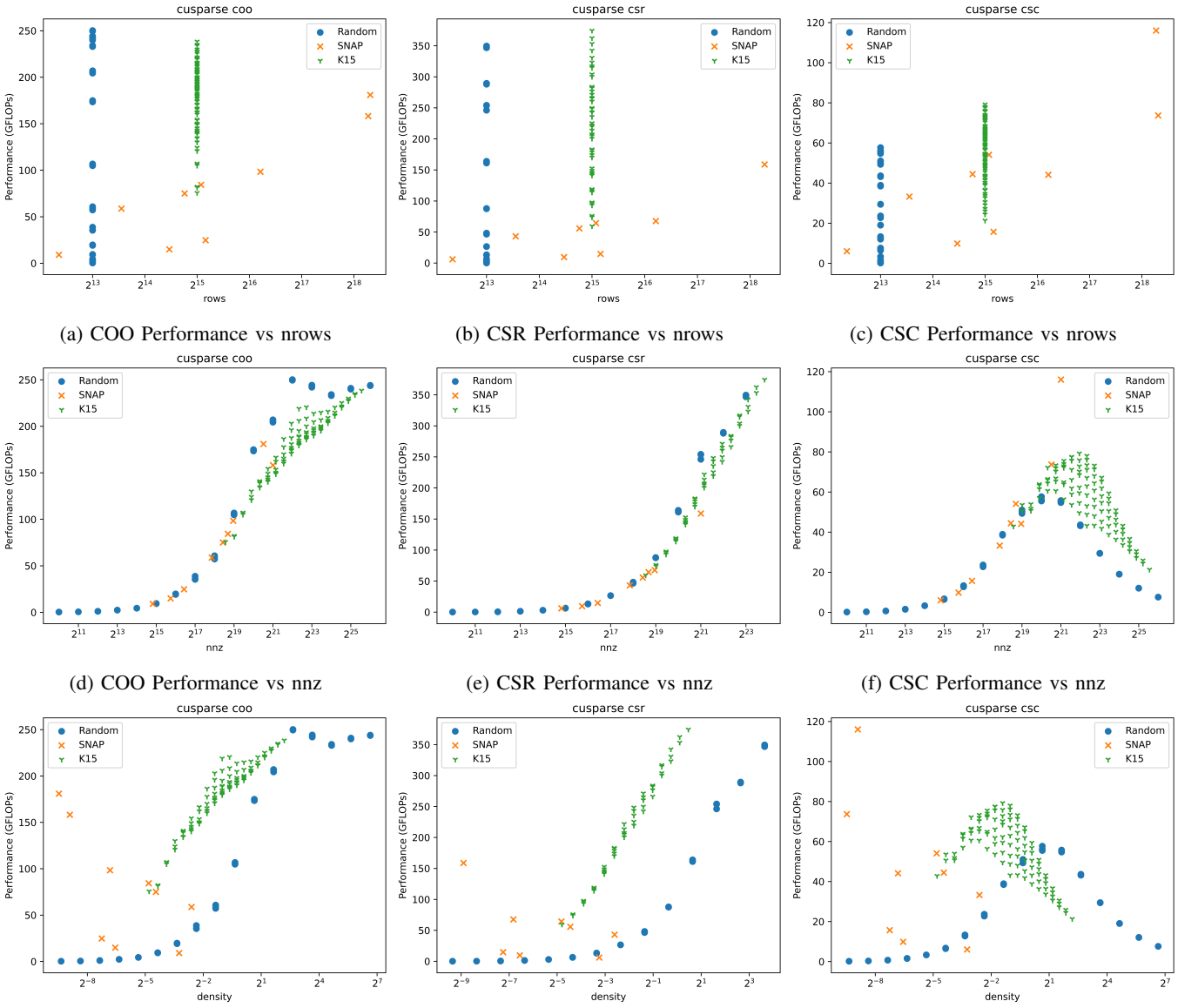
Figure 6 shows the performance results of this experiment using cuSparse on H100 GPU, plotted against number of rows, number of non-zeros, and density used as features on the horizontal axis. Each subplot illustrates the performance of a specific sparse data representation out of the three we evaluated: COO, CSR, and CSC.



(a) cuSparse on H100

(b) MKL on Intel Xeon Gold

Fig. 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) 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

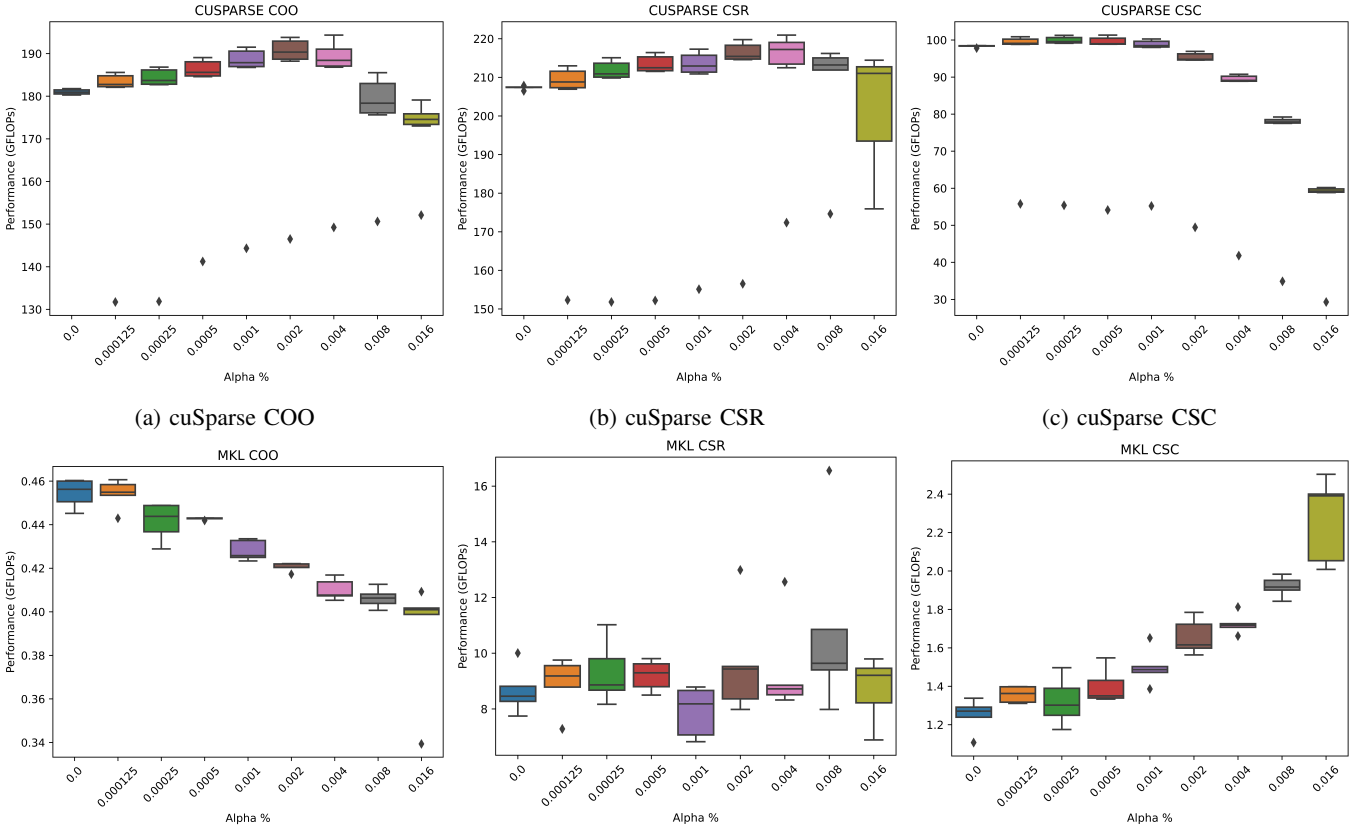
(g) COO Performance vs density (h) CSR Performance vs density (i) CSC Performance vs density

Fig. 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.

Looking at the relationship between Performance and number of rows (Figure 6a,6b,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 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 6g,6h,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



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

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

Fig. 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.

existing performance data) at different density values that have not been evaluated.

For number of non-zeros (Figure 6d,6e,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 III: 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

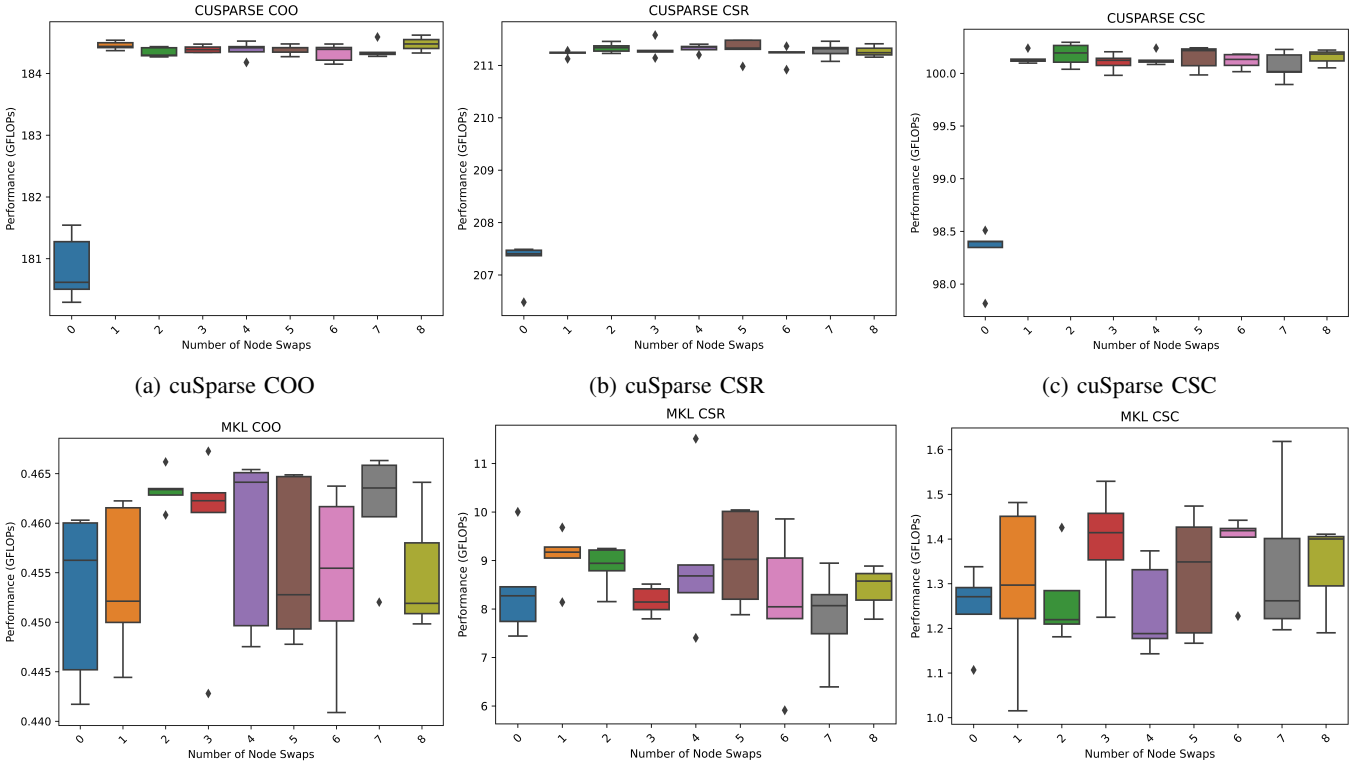
C. Inducing Noise

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 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.

1) Adding Random Sparse Graphs with Varying Density: For this experiment, we use the 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).

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 7a,7b). However, for MKL, the range of change for



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

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

Fig. 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.

median performance is limited: less than 0.06 GFLOPs for COO, around 1 GFLOPs 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.

2) 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 8 swaps only because the swapping operation is computationally expensive.

Figure 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.

D. 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 settings, this is not the only overhead that needs to 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 IV 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 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 IV: 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%

V. Conclusion

In this paper, 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.

References

[1] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[2] S. Beamer, K. Asanović, and D. Patterson, "Gail: The graph algorithm iron law," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA¡sup¿3¡/sup¿ '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833179.2833187

[3] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

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

[5] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017.

[6] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira, "Mathematical foundations of the graphblas," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–9.

[7] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[8] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *Intel Math Kernel Library*. Cham: Springer International Publishing, 2014, pp. 167–188.

[9] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.

[10] AMD, "rocsparse documentation https://rocsparse.readthedocs.io/en/latest/," 2023, [Online; accessed 5-October-2023]. [Online]. Available: https://rocsparse.readthedocs.io/en/latest/

[11] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication," in *Knowledge Discovery in Databases: PKDD 2005*, A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 133–145.

[12] J. Leskovec and C. Faloutsos, "Scalable modeling of real graphs using kronecker multiplication," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 497–504. [Online]. Available: https://doi.org/10.1145/1273496.1273559

[13] C. Seshadhri, A. Pinar, and T. G. Kolda, "An in-depth study of stochastic kronecker graphs," in *2011 IEEE 11th International Conference on Data Mining*, 2011, pp. 587–596.

[14] A. S. Ramani, N. Eikmeier, and D. F. Gleich, "Coin-flipping, ball-dropping, and grass-hopping for generating random graphs from matrices of edge probabilities," *SIAM Review*, vol. 61, no. 3, pp. 549–595, 2019.

[15] P. Erdős, A. Rényi *et al.*, "On the evolution of random graphs," *Publ. math. inst. hung. acad. sci*, vol. 5, no. 1, pp. 17–60, 1960.

[16] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, pp. 47–97, Jan 2002. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.74.47

[17] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[18] X. Guo and L. Zhao, "A systematic survey on deep generative models for graph generation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 5, pp. 5370–5390, 2022.

[19] M. Suhail, A. Mittal, B. Siddiquie, C. Broaddus, J. Eledath, G. Medioni, and L. Sigal, "Energy-based learning for scene graph generation," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2021, pp. 13 936–13 945.

[20] R. Liao, Y. Li, Y. Song, S. Wang, W. Hamilton, D. K. Duvenaud, R. Urtasun, and R. Zemel, "Efficient graph generation with graph recurrent attention networks," *Advances in neural information processing systems*, vol. 32, 2019.

[21] M. Yoon, Y. Wu, J. Palowitch, B. Perozzi, and R. Salakhutdinov, "Graph generative model for benchmarking graph neural networks," 2023.

[22] K. Abdelaal and R. Veras, "Observe locally, classify globally: Using gnns to identify sparse matrix structure," in *Advances in Computational Intelligence*, I. Rojas, G. Joya, and A. Catala, Eds. Cham: Springer Nature Switzerland, 2023, pp. 149–161.

[23] ——, "A framework for analyzing the robustness of graph models," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*, 2023.

[24] A. Lopes, F. Pratas, L. Sousa, and A. 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)*, 2017, pp. 259–268.

[25] E. Konstantinidis and Y. Cotronis, "A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling," *Journal of Parallel and Distributed Computing*, vol. 107, pp. 37–56, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731517301247

[26] C. Yang, T. Kurth, and S. Williams, "Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 20, p. e5547, 2020.

[27] K. Z. Ibrahim, S. Williams, and L. Oliker, "Performance analysis of gpu programming models using the roofline scaling trajectories," in *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 2019, pp. 3–19.

[28] N. Ding and S. Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18.

[29] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graph evolution: Densification and shrinking diameters," *ACM Trans. Knowl. Discov. Data*, vol. 1, no. 1, p. 2–es, mar 2007. [Online]. Available: https://doi.org/10.1145/1217299.1217301