UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

GPU-ACCELERATED KRONECKER GRAPH FITTING AND A GENERAL FRAMEWORK FOR GRAPH STRUCTURE DESCRIPTOR LANGUAGE MODELS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

by MOEZ ULLAH KHAN

Norman, Oklahoma

2026

GPU-ACCELERATED KRONECKER GRAPH FITTING AND A GENERAL FRAMEWORK FOR GRAPH STRUCTURE DESCRIPTOR LANGUAGE MODELS

A THESIS APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Richard Veras (Chair)

Dr. Chao Lan

Dr. Anindya Maiti

Acknowledgments

The computing for this project was performed at the OU Supercomputing Center for Education & Research (OSCER) at the University of Oklahoma (OU).

This material is based upon work supported by the National Science Foundation under Grant No. 2440646.

Table of Contents

List of Tables

List of Figures

Abstract

Understanding the complex behaviors of large-scale graphs is an important task. It helps us comprehend phenomena like the spread of viruses across online networks, the spread of diseases, and the spread of tumors in the brain, etc. Using graph models as surrogates for real data allows domain experts to analyze the behavior of these networks from compact representations. This requires fitting model parameters for the given graph data. For dense graphs, these can be processed with relative ease due to the performance improvements made over time; however, problems arise when the graphs are sparse, as most existing methods for dense graphs are not designed for graphs where the computation performed is affected by their structure. For hyper-sparse graphs, KronFit addresses this challenge by fitting the parameters to a Kronecker graph model. These parameters can be used to recreate a synthetic graph with similar properties to the original graph to understand their underlying workings. However, these tools are limited to singular models, like KronFit for Kronecker graphs and MAGfit for multiplicative Attribute graphs. They are sequential, meaning that for large enough graphs, the computation time to run these tools is too high to be usable.

We present MFit, a heterogeneous CPU–GPU pipeline for Kronecker graph fitting that runs Metropolis–Hastings MCMC permutation sampling on the CPU and offloads likelihood and gradient computations to thousands of GPU cores in parallel. MFit frames graph model fitting as a modern machine learning optimization task, leveraging PyTorch automatic differentiation and the Adam optimizer in place of hand-derived gradient calculations. Evaluated across 20 benchmark graphs, MFit achieves speedups exceeding 10× over KronFit while producing equal or better parameter recovery quality, making it a practical choice at scales where single-core fitting becomes intractable. We also extend MFit into GSDLFit, a more

general fitting framework that can fit any graph model expressible in the Graph Structure Descriptor Language, not just Kronecker graphs.

Chapter 1

Introduction

Large-scale graphs are a major part of human society. Complex interactions in real life are often represented or stored in terms of graphs. They can also be used to describe relations in complex networks or even analyze and predict their behaviors. For example, graph-based methods are used to detect patterns of fraud in banking credit card transactions, online banking transactions, and other types of banking operations Patil et al. (2024). They have also been used in understanding Reddit discussion dynamics, YouTube video sharing and viewer clusters understanding, Twitter information diffusion and influence, etc. N et al. (2024). They even have medical uses like analyzing the behavior of a drug through cellular pathways in the body Veras (2025). So large graphs are deeply tied to our world and have very important use cases. They are the medium through which influence, disease, and information travel.

Modeling these graphs is therefore a problem of practical importance. A powerful approach to understanding real-world graphs is to fit a model to the observed data. A model is just a compact parameterization that captures the structural properties of a real network in a small set of stochastic values. Once these values are tuned for a particular graph, it has several practical uses. First, it compresses the graph into a small set of parameters, enabling efficient storage and sharing. These parameters can also be used to generate synthetic graphs that have similar properties to the original graph, allowing data to be shared without revealing sensitive information. The model can also be used to extrapolate the graph to higher

scales to predict future growth, or to produce even smaller samples for experimentation purposes Leskovec and Faloutsos (2007).

Fitting such a model to a real graph is, however, a hard computational problem. The likelihood of a graph under a generative model requires summing over all possible permutations of the graph, which has a cost of $N!$ terms for a graph of N nodes which is practically infeasible. Even with well-designed approximations, the computation is still irregular and depends heavily on the data. Because of this, it does not benefit much from the performance improvements that modern hardware typically provides for dense and well-structured tasks Veras (2025). There are existing tools, like KronFit, that address this issue with clever algorithmic reductions, but they remain bound to sequential, single-core execution. For large graphs, this means the algorithm can take several hours to run.

This thesis presents MFit, a GPU-accelerated Kronecker graph fitting framework developed as part of the Graph Structure Descriptor Language (GSDL) project Veras (2025). MFit transforms the fitting process into a differentiable optimization problem. Instead of using manually derived gradients like earlier methods, it relies on PyTorch's automatic differentiation and uses the GPU to compute gradients. The result is a heterogeneous CPU–GPU pipeline where MCMC permutation sampling runs in parallel on the CPU, while likelihood and gradient computation run on the GPU. MFit is also designed to generalize beyond the Kronecker model, laying the groundwork for fitting arbitrary graph structure models within the GSDL framework.

This thesis makes the following contributions:

- **MFit: A GPU-accelerated Kronecker graph fitting pipeline.** A heterogeneous CPU–GPU implementation of Kronecker graph fitting that replaces KronFit's hand-derived, sequential gradient engine with PyTorch automatic differentiation on GPU, while retaining Metropolis–Hastings MCMC permutation sampling on the CPU via Numba-compiled kernels.

- **Gradient variance reduction via chunked ensembling.** Rather than computing

a single gradient per iteration, MFit partitions the MCMC steps into C chunks and averages gradients across them, reducing the variance of noisy MCMC-derived gradient estimates and improving convergence stability.

- **Parallel synthetic graph generator.** A parallel R-MAT graph generator that uses the recursive structure of Kronecker graphs to distribute work across CPU cores, making it possible to generate 12,000 large graph instances efficiently without external libraries.

- **GSDLFit: A general model-agnostic fitting framework.** An extension of MFit that replaces hardcoded Kronecker assumptions with a wrapper interface, allowing the same CPU–GPU pipeline to fit any graph model expressible in the Graph Structure Descriptor Language, including Kron+Kron, Kron+Random, and MAG.

- **Comprehensive empirical evaluation.** A systematic evaluation of MFit across four experimental dimensions: comparison against KronFit on 20 benchmark datasets, optimizer selection, CPU thread scaling, and numerical precision studies, characterizing the performance, accuracy, and resource trade-offs of the system.

- **Identification of the Dead Zone.** A discovery that Kronecker graph generation via R-MAT produces systematically degenerate graphs at $k = 6$ across all tested initiator parameters, traced to the interaction between exponential node space growth and probability concentration in the initiator matrix.

Experiments on multiple real-world graphs show that MFit is much faster than KronFit while producing results of similar or better quality. The accuracy is measured using the L2 difference between the learned parameters and the ground-truth parameters. For example, at $k = 20$, for graphs from SNAP N et al. (2024) and from the paper Kronecker graphs: An Approach to Modeling Networks Leskovec et al. (2009) MFit has achieved over 10× speedup, and this increases as the graph size increases.

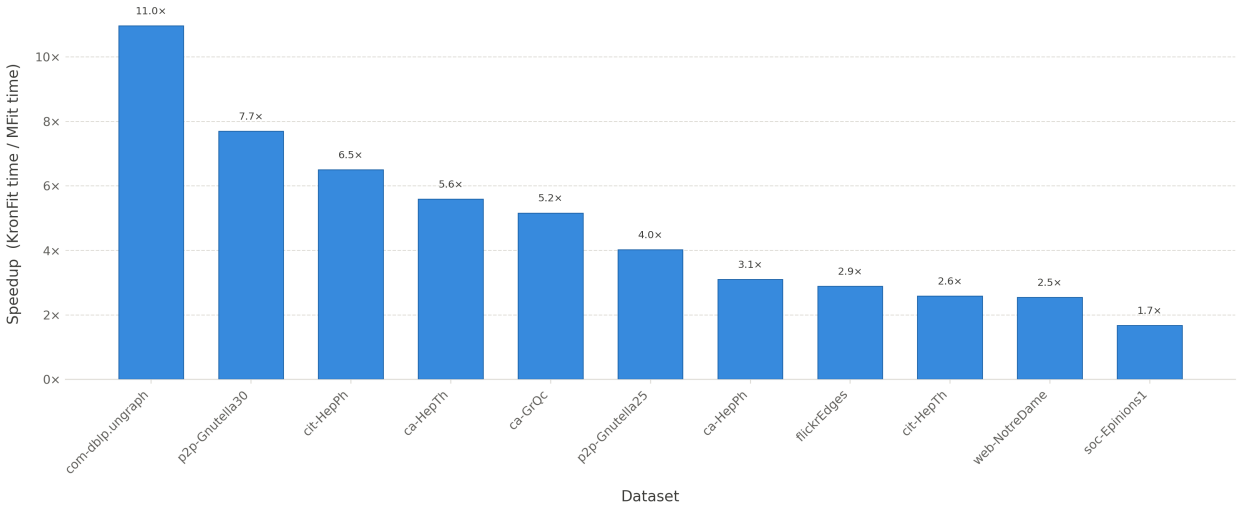
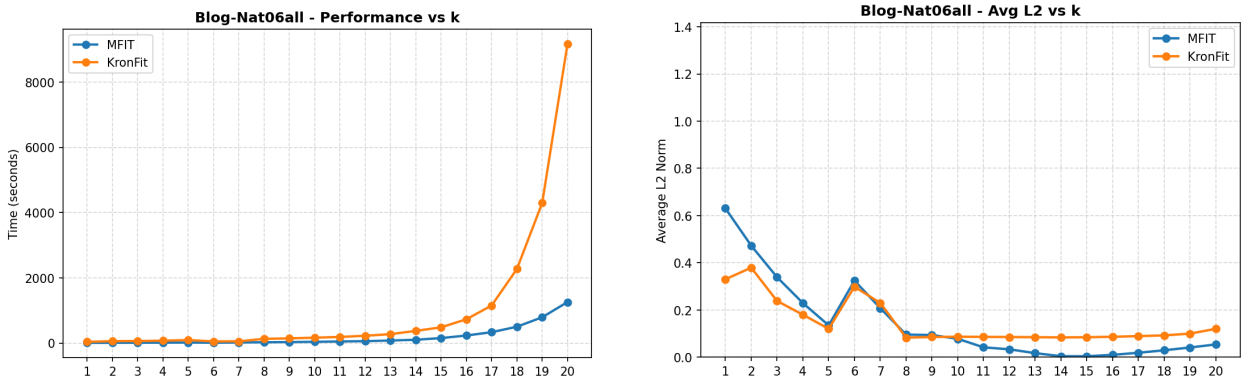


Figure 1.1: Wall-clock speedup of MFit over KronFit across eleven representative real-world graph datasets

Table 1.1 shows the runtime and log-likelihood values behind these speedups. The datasets come from the SNAP repository N et al. (2024) and match the benchmark graphs used in Leskovec et al. Leskovec et al. (2009). Across all eleven datasets, MFit is both faster than KronFit and achieves higher log-likelihood values, showing that GPU-based gradient optimization produces a better fit than the sequential approach.

Table 1.1: Comparison of MFit and KronFit across datasets

	Best LL		Time (s)	
Network	**MFit**	**KronFit**	**MFit**	**KronFit**
com-dblp.ungraph	-12183146.3	-12915650.2	54.53	597.77
cit-HepPh	-3434725.8	-3584058.2	74.43	483.60
soc-Epinions1	-3721640.9	-3911779.3	228.05	380.29
web-NotreDame	-15184333.3	-15304772.1	296.79	754.11
p2p-Gnutella25	-516968.2	-545045.5	31.47	126.57
p2p-Gnutella30	-881141.2	-934799.8	20.50	157.80
ca-HepTh	-204065.3	-416232.5	26.19	146.39
ca-HepPh	-724023.1	-1326058.1	161.18	499.87
flickrEdges	-16599520.0	-16738438.3	502.56	1448.50
ca-GrQc	-98066.6	-169098.1051	26.9	138.64
cit-HepTh	-2538001.9	-2660710.0372	180.4	466.61



(a) Performance Comparison

(b) L_2 Convergence Comparison

Figure 1.2: Comparative analysis for the Blog-Nat06all dataset. The left plot (a) compares total runtime (in seconds) as a function of Kronecker power k for both MFit and KronFit. The right plot (b) tracks the average L_2 error versus k, illustrating the consistency of parameter recovery accuracy as the graph scale increases.

Chapter 2

Background

As discussed in the introduction, Leskovec proposed the Kronecker model that satisfies the properties of real networks. The core idea of the Kronecker model is the recursive expansion of a smaller initiator matrix. We will call this small adjacency matrix the initiator because it is used as a base to build larger matrices. Let's start with a smaller adjacency matrix of size 3×3 and call it Θ_1 with N_1 nodes, K_1 edges, and size $n \times n'$. The Kronecker product of the initiator matrix with itself looks something like the following:

1	1	0
1	1	1
0	1	1

(a) Θ_1

Θ_1	Θ_1	0
Θ_1	Θ_1	Θ_1
0	Θ_1	Θ_1

(b) $\Theta_2 = \Theta_1 \otimes \Theta_1$

Figure 2.1: Kronecker product applied to initiator matrix Θ_1 Leskovec and Faloutsos (2007)

The Kronecker Model

The Kronecker product of two matrices is defined as the product of the elements of some matrix with all the elements of the other matrix. Let's say we have another matrix B of size $m \times m'$, its Kronecker product with the matrix Θ_1 will be of the following form:

$$C = \Theta_1 \otimes B = \begin{pmatrix} a_{1,1}B & a_{1,2}B & \cdots & a_{1,m}B \\ a_{2,1}B & a_{2,2}B & \cdots & a_{2,m}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1}B & a_{n,2}B & \cdots & a_{n,m}B \end{pmatrix} \tag{2.1}$$

If this process is repeated with the initiator matrix Θ_1 of size $N_1 \times N_1$ K iterations, the resulting adjacency matrix will be:

$$\Theta_k = \underbrace{\Theta \otimes \Theta \otimes \cdots \otimes \Theta}_{k \text{ times}} = \Theta^k \tag{2.2}$$

Θ^k represents the k^{th} Kronecker product of the initiator matrix Θ_1. This graph Θ_k will have N_1^K nodes and E_1^K edges Leskovec and Faloutsos (2007). A visual of how the adjacency structure is preserved by using Kronecker product is shown in Figure 2.2

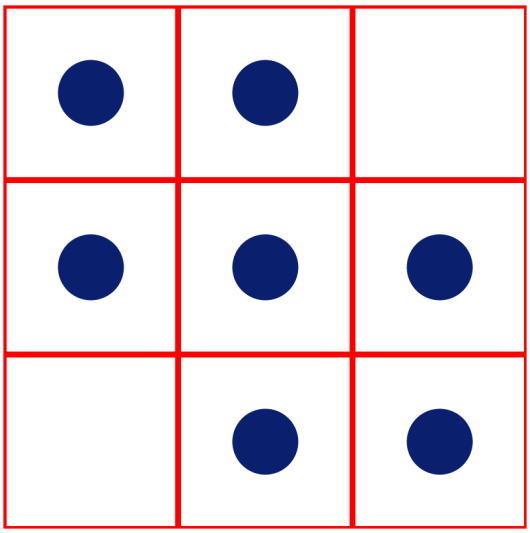
It can be seen in Figure 2.2 how the Kronecker product preserves the structure of the initial matrix Leskovec et al. (2009). Kronecker graphs have also been proven to preserve properties exhibited by real-life networks. These include multinomial degree distributions, multinomial eigenvalue and eigenvector distributions, densification power laws, etc. Leskovec et al. (2009).

In our case, instead of 1's and 0's, we will be working with Stochastic Kronecker Graphs and thus the entries for our initiator will form a probability matrix Θ_1 of size $N_1 \times N_1$ where

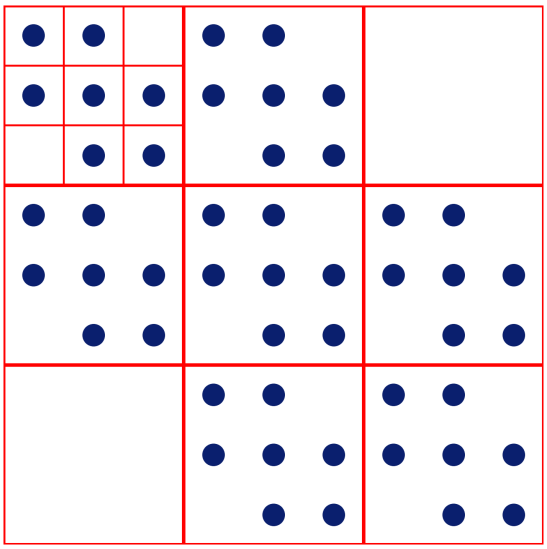
$$\Theta_1 = [\theta_{ij}] \tag{2.3}$$

where θ_{ij} is the probability of an edge existing, and its value range is $[0, 1]$. Once the k^{th} Kronecker power of Θ_1 is calculated as $\mathcal{P} = \Theta^k$, then the probability p_{uv} of an edge (u, v) can be calculated very easily by leveraging the hierarchical structure of Kronecker graphs.

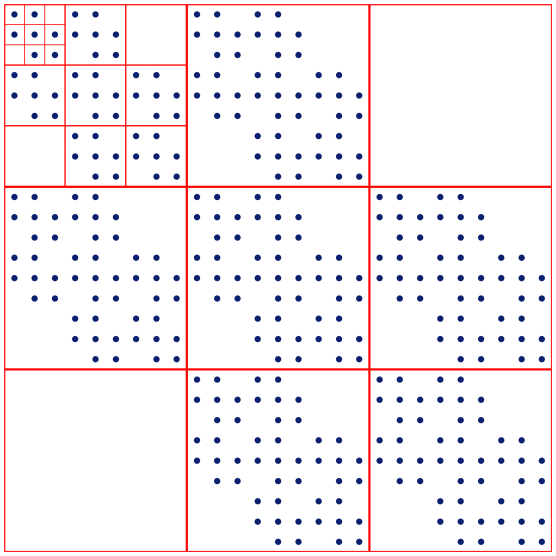
Now that we can calculate the probability of an edge, we can use Equation 2.6 to calculate $P(G)$ which is the likelihood of graph G being generated by some model, or Θ_1 in our case. To



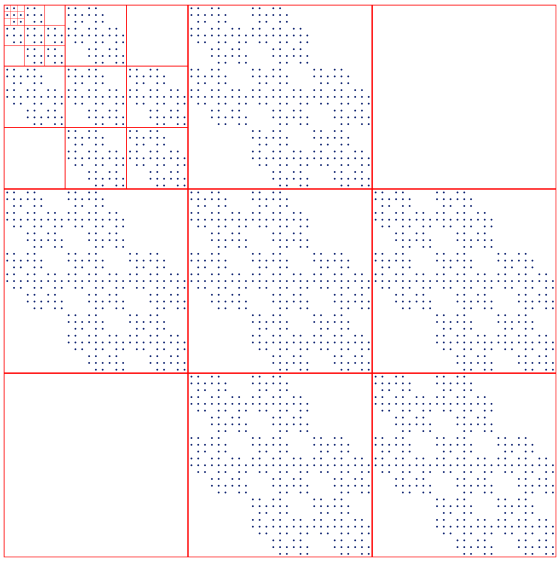
(a) Θ_1 adjacency matrix (3×3)



(b) Θ_2 adjacency matrix (9×9)



(c) Θ_3 adjacency matrix (27×27)



(d) Θ_4 adjacency matrix (81×81)

Figure 2.2: Image a shows a visual of the adjacency matrix Θ_1 where the images b, c, and d show the 2nd, 3rd and 4th Kronecker products of the adjacency matrix Θ_1 with itself, respectively. Blue dots represent 1's inside the adjacency matrix, while the empty boxes represent 0's

tackle this problem, Leskovec and Faloutsos (2007) created a tool **KRONFIT** that evaluates $P(G|\sigma)$ in linear time $O(E)$.

KronFit: Fitting Kronecker Graphs to Real Networks

The goal of KronFit is that given a graph G, how to find the initiator matrix Θ that best represents it. In other words, we need to find the initiator matrix Θ that has the highest probability of generating the input graph or in simple terms yields the biggest $P(G)$.

$$\arg\max_{\Theta} P(G \mid \Theta) \tag{2.4}$$

To evaluate Equation 2.4, two things are needed.

First is **Likelihood Estimation**. As shown in Equation 2.3, the entries of $\mathcal{P}$ are the probabilities of edges existing. An adjacency matrix $\mathcal{P}$ of size $N \times N$ will take $O(N^2)$ time and space even if we only want to get the probability of a single edge, which is highly inefficient time and memory-wise. The second issue is **Node Labeling**. A graph G has a set of unique labels for each of its N nodes. Since these labels do not carry any meaning, the same graph could be assigned different labels, and this would change the likelihood calculation. This results in the same graph having $N!$ permutations, or isomorphic representations Leskovec and Faloutsos (2007). This brings our probability calculation to a total of $O(N! \cdot N^2)$ which is computationally impossible for large graphs Leskovec et al. (2009).

KronFit addresses both problems with a set of clever algorithmic reductions, which we describe in detail in the next chapter. In short, the hierarchical structure of Kronecker matrices enables the probability of a single edge to be computed in $O(k)$ without materializing full Θ^k. Utilizing a combination of Taylor approximation and Kronecker structure, the non-edge probability computation is reduced from $O(N)$ to $O(1)$. To avoid enumerating all $N!$ permutations, KronFit uses Metropolis–Hastings MCMC sampling to focus on the high-likelihood regions of the permutation space. Overall, KronFit reduces fitting complexity for a graph to $O(E)$ where E is the number of edges in the graph. Together, these reduce the overall fitting complexity to $O(E)$ per gradient step. Despite this, KronFit remains bound to sequential, single-core execution, which becomes a practical bottleneck for large graphs. MFit directly

addresses this limitation by offloading gradient computation to the GPU while retaining the same mathematical foundations established here.

Edge Probabilities

Earlier, we found that after some k iterations of Kronecker product on some initiator Θ, we get $\mathcal{P} = \Theta^k$, where the entries of this matrix, p_{uv}, are the probabilities of an edge existing between some node u and v. To get the probability $P(G)$ of this entire graph, we can just multiply all of its entries. But for high enough k, this adjacency matrix will be enormous, not only to be materialized in memory but also computing the product of its entries would take forever.

Luckily, the hierarchical structure of Kronecker graphs lets us calculate the probability of an edge in $O(k)$ time without generating the massive Θ^k. Θ^k is built by doing the Kronecker product of a small 2×2 matrix Θ by itself k times, which creates a massive grid. At every i^{th} Kronecker product up to k, we choose one of the 4 values inside Θ. We can think of it as a binary tree where at each node we can choose one of the 4 branches. That is why, if we convert a node's index in binary, we get a list of 1's and 0's, where the node u's binary values tell us about whether that edge falls in the top or bottom half, and node v's binary values tell us about the edge falling into either the left or the right half of the quadrant. Then, the edge probability becomes simply the product of the corresponding entries Leskovec et al. (2009).

$$\begin{bmatrix} \text{Top-Left } (u=0,v=0): \ p_{00} & \text{Top-Right } (u=0,v=1): \ p_{01} \\ \text{Bottom-Left } (u=1,v=0): \ p_{10} & \text{Bottom-Right } (u=1,v=1): \ p_{11} \end{bmatrix} \tag{2.5}$$

We are recursively traversing the block arrangement and tracking the edges in the Θ_1 (as shown in Figure 2.2) whose product leads to the probability of the current edge. This can be represented in the form of the following equation, as shown in Leskovec et al. (2009).

$$p_{uv} = \prod_{i=0}^{k-1} \Theta \left[\left\lfloor \frac{u-1}{N_1^i} \right\rfloor \pmod{N_1} + 1, \left\lfloor \frac{v-1}{N_1^i} \right\rfloor \pmod{N_1} + 1 \right] \tag{2.6}$$

This is why it takes $O(k \cdot logN_1)$ Leskovec and Faloutsos (2007) time to compute the probability of an edge. This also makes it possible to calculate the probability on the go instead of generating the entire adjacency matrix. But one could argue that if you need to find $P(G)$, you have to perform this calculation for every edge present in the graph. If the graph has N nodes, where $N = N_1^k$, then the total number of possible edges in this graph will be $O(N^2)$ which brings our likelihood calculation for the entire graph to $O(k \cdot logN_1 \cdot N^2)$ which is again massive time-complexity-wise, even if we don't have to materialize Θ^k. But an observation from Leskovec and Faloutsos (2007) shows that real graphs, including Kronecker graphs are sparse, which means that the number of edges is linear compared to the number of nodes. This reduces the N^2 to N. Thus, our complexity reduces to $O(k \cdot logN_1 \cdot N)$ or simply $O(N)$, since k and $log(N_1)$ are very small, and can be considered constants.

Log-Likelihood

Our aim is to calculate the likelihood of seeing a graph given all its permutations. This can be represented as

$$\ell(\Theta) = \log P(G \mid \Theta) = \log \sum_{\sigma} P(G \mid \Theta, \sigma)\, P(\sigma) \tag{2.7}$$

where σ is some permutation of the graph G Leskovec et al. (2009). We use log-likelihood for convenience instead of raw probabilities. For a given permutation, this equation can be rewritten as

$$\ell(\Theta) = \sum_{(u,v)\in E} \log p_{uv} \; + \sum_{(u,v)\notin E} \log(1 - p_{uv}) \tag{2.8}$$

where u and v are arbitrary nodes of an edge, and E is the set of edges in the graph. The first sum accumulates the contribution of all observed edges, while the second sum runs

over all pairs of nodes that do not form an edge. We will call these non-edges. Using the log of the initiator entries, the term for edges can be computed very fast, but for sparse graphs where most pairs are non-edges, naively computing the non-edge term can be very expensive, as we would have to check all the cells of the adjacency matrix for the graph. For sparse graphs, this term dominates because the number of non-edges grows at a higher order than the number of edges.

To address this issue, Leskovec et al. (2009) proposed using Taylor approximation. Since the p_{uv} values are probabilities, when generating a Kronecker graph, these probabilities have to be multiplied with each other and they get even smaller. We can safely assume that these values are close to 0. Using this information, we can use a Taylor Expansion around $x = 0$ to approximate the function.

$$\log(1 - x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \cdots \tag{2.9}$$

We mentioned earlier that the values of x are close to 0, which makes the higher-order terms like x^3 and x^4 negligible. Thus, we can safely truncate the series after the second term, which gives

$$\log(1 - x) \approx -x - \frac{x^2}{2} \tag{2.10}$$

Even with this truncated form, we still have to sum over possibly N^2 terms. This is where we can leverage the structure of Kronecker matrices. Instead of calculating $log(1 - P_{uv})$ for all the missing edges, we first calculate the sum for all possible edges as if the graph was empty, and then correct it for the edges that do exist, as proposed by Leskovec and Faloutsos (2007).

For a Kronecker graph, the sum of all the entries in its adjacency matrix is simply the sum of the terms in the initiator matrix raised to the power of k. Lets say we have some initiator matrix $\mathcal{P}$ of the form

$$\mathcal{P} = \begin{bmatrix} p_{00} & p_{01} \\ p_{10} & p_{11} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

then

$$\sum_{u,v} P_{uv} = \left(\sum \mathcal{P}\right)^k = (p_{00} + p_{01} + p_{10} + p_{11})^k \tag{2.11}$$

Similarly, for the squared term:

$$\sum_{u,v} P_{uv}^2 = \left(\sum \mathcal{P}^2\right)^k = \left(p_{00}^2 + p_{01}^2 + p_{10}^2 + p_{11}^2\right)^k \tag{2.12}$$

We can prove this using a simple example. Lets take the above initiator matrix and do a Kronecker product on it with itself.

$$\mathcal{P} \otimes \mathcal{P} = \begin{bmatrix} a\mathcal{P} & b\mathcal{P} \\ c\mathcal{P} & d\mathcal{P} \end{bmatrix} = \begin{bmatrix} aa & ab & ba & bb \\ ac & ad & bc & bd \\ ca & cb & da & db \\ cc & cd & dc & dd \end{bmatrix}$$

If we want the Taylor expansion, we need to add up all 16 of these elements.

$$\text{Sum} = (aa + ab + ac + ad) + (ba + bb + bc + bd) + (ca + cb + cc + cd) + (da + db + dc + dd)$$

Factoring out the first term from each group:

$$\text{Sum} = a(a + b + c + d) + b(a + b + c + d) + c(a + b + c + d) + d(a + b + c + d)$$

Now factoring out the common term $(a + b + c + d)$:

$$\text{Sum} = (a+b+c+d)(a+b+c+d)$$

$$\text{Sum} = (a+b+c+d)^2 \tag{2.13}$$

As shown above, when we performed one Kronecker product and created two copies of the initiator, meaning that $k = 2$, we ended up with Equation 2.13. If we repeat this process for some arbitrary k value, then we end up with the equations Equation 2.11 and Equation 2.12. Using these two equations and Equation 2.10, we get a new equation for the likelihood of the entire graph being empty.

$$\ell_{empty} \approx -\left(\sum \mathcal{P}\right)^k - \frac{1}{2}\left(\sum \mathcal{P}^2\right)^k \tag{2.14}$$

Now we can subtract likelihood of the real edges being non edges that we assumed were non edges in Table 5 and add the likelihood of these edges

$$\ell(\Theta) = \sum_{(u,v)\in E} log(P_{uv}) + \ell_{empty} - \sum_{(u,v)\in E}\left[-P_{uv} - \frac{1}{2}P_{uv}^2\right] \tag{2.15}$$

The complexity of the Edge log term is $O(E \cdot k \cdot log(N_1))$ using Equation 2.6. $log(N_1)$ is usually very small so we can safely consider it $O(1)$. Thus the complexity of edge log term is really $O(E \cdot k)$. The complexity is the same for the edge correction term since we are just iterating through the edges and calculating the contribution using Taylor approximation. The complexity for the empty graph term is also very small and can be considered $O(1)$. This reduces the overall complexity of calculating the likelihood of some graph G with some permutation σ to $O(E \cdot k)$ or simply $O(E)$.

Node Permutation

Now that we have discussed how to calculate the log-likelihood for a given permutation, it's time to return to the second major problem, the node labeling. If we have some graph G with N nodes, those nodes can have any labels. If we randomly relabel the nodes of this graph, then the adjacency matrix completely changes. Since our likelihood calculations depend on how nodes map into the Kronecker structure, we will have to consider all $N!$ permutations of the graph as shown earlier in Equation 2.7.

To solve this Leskovec et al. (2009) proposes using Metropolis Hastings Markov Chain Monte Carlo (MCMC). Instead of evaluating the log-likelihood over all possible permutations, KronFit approximates this by sampling a limited number of high-quality permutations. The main intuition behind this is that not all permutations align well with the Kronecker structure. The ones that don't align well yield very low likelihood and vice versa Leskovec et al. (2009). So rather than computing all permutations, we focus on what matters the most. To do this, KronFit uses MCMC to sample permutations based on their likelihood.

MCMC Permutation Sampling

As mentioned before, KronFit aims to build a Markov Chain over all possible permutations such that the permutations with higher likelihoods are visited more often. KronFit starts with an initial permutation that is in descending order of their node degrees. This assigns the highest degree nodes to the earlier positions in the permutation. In the paper Kronecker graphs: An Approach to Modeling Networks, Leskovec et al. (2009) has observed that the predicted initiator matrices of real networks they analyzed, tend to have a large top-left entry and a small bottom-right entry. The way Kronecker matrices grow is repeated Kronecker powering of the probabilities. If the top left position has the highest value, then it will also end up creating high degree nodes. Since we are fitting the real graph with a Kronecker model, and real networks tend to yield this structure upon fitting, we expect the high-degree nodes of the real graph to correspond to the top-left positions of the Kronecker matrix.

That is why starting with the degree-sorted permutation places the Markov chain closer to a high-likelihood region of the permutation space. Starting with a random permutation would require a much longer chain to get to the same region.

At each step, two nodes are selected and their position in the current permutation σ is swapped. Once this new permutation σ' is obtained, if the log-likelihood of σ' is greater than σ, then σ' is always accepted. If it's not then it may still be accepted with some probability. KronFit chooses a number α between 0 and 1 from a uniform distribution. If this α is greater than the ratio of the log-likelihood of σ'/σ, σ' is again accepted. This part is important to make sure the algorithm doesn't get stuck in a local optima and get a chance to explore the space Leskovec and Faloutsos (2007). This part is repeated for a number of iterations. This whole process is shown below

Algorithm 1: SamplePermutation(G, Θ)

Input: Kronecker initiator matrix Θ, graph G

Output: Permutation σ

repeat

  Draw j, k uniformly from $(1, \ldots, N)$;

  $\sigma' \leftarrow$ SwapElements(σ, j, k);

  if $\ell(\Theta)_{\sigma'} > \ell(\Theta)_{\sigma}$ **then**

    $\sigma \leftarrow \sigma'$;

  end

  else

    Draw $u \sim U(0, 1)$;

    if $u < \exp(\ell(\Theta)_{\sigma'} - \ell(\Theta)_{\sigma})$ **then**

      $\sigma \leftarrow \sigma'$;

    end

  end

until $\sigma \sim P(\sigma \mid G, \Theta)$;

return σ

Gradient Calculation

Now that we have discussed the mechanism for exploring the permutation space, and we previously discussed how to compute $P(G|\sigma, \Theta)$, now we can discuss how it is used in gradient descent to update the parameters. To calculate the gradients, we just differentiate Equation 2.7

$$\frac{\partial}{\partial \Theta} \ell(\Theta) = \frac{\partial}{\partial \Theta} \log \sum_{\sigma} P(G \mid \sigma, \Theta) \tag{2.16}$$

Since we have some T number of permutations we explored in the earlier section, to compute the log-likelihood and the gradient, we simply average them over all observed samples

$$\ell(\Theta) \approx \frac{1}{T} \sum_{t=1}^{T} \log P(G \mid \sigma^{(t)}, \Theta), \quad \frac{\partial \ell}{\partial \Theta} \approx \frac{1}{T} \sum_{t=1}^{T} \frac{\partial \log P(G \mid \sigma^{(t)}, \Theta)}{\partial \Theta} \tag{2.17}$$

One thing to note is that even when a permutation σ' is rejected, the likelihood and the gradients for the current permutation σ are still considered as it is still a valid sample from the permutation space Leskovec and Faloutsos (2007). This also makes sure that if a permutation has a high likelihood, then the algorithm tries its best to stick with it.

For a single parameter θ_{ab}, where a, and b are the corresponding nodes of an edge, the gradient of the log-likelihood under some permutation σ is obtained by differentiating Equation 2.7 with respect to θ_{ab}:

$$\frac{\partial \log P(G \mid \sigma, \Theta)}{\partial \theta_{ab}} = \sum_{(u,v) \in E} \frac{c_{ab}(u, v, \sigma)}{\theta_{ab}} - \sum_{(u,v) \notin E} \frac{c_{ab}(u, v, \sigma)\, p_{uv}(\sigma)}{(1 - p_{uv}(\sigma))\, \theta_{ab}} \tag{2.18}$$

where $c_{ab}(u, v, \sigma) \in \{0, 1, \ldots, k\}$ counts the number of Kronecker levels at which the recursive partitioning of edge (u, v) maps to position (a, b) in Θ Leskovec et al. (2009). In simple words, this measures how much θ_{ab} contributes in the edge probability p_{uv} given some permutation. The first sum rewards θ_{ab} for the edges that are present, while the second sum penalizes it for non-edges. The non-edge term is approximated using Table 5 as we

discussed earlier. This whole process can be expressed in the following short form:

Algorithm 2: Gradient and log-likelihood estimation

Input: Initiator matrix Θ, graph G, warm-up steps W, sample count T

Output: Estimated $\ell(\Theta)$, estimated $\nabla_\Theta \ell(\Theta)$

Compute approximate log-likelihood ℓ_σ and gradient $\nabla_\Theta \ell_\sigma$ for current σ;

Phase 1 — warm-up (mix the chain, do not accumulate);

for $s = 1$ **to** W **do**

  Propose swap of two nodes u_1, u_2 in σ;

  Compute $\Delta\ell = \text{NodeDelta}(u_1) + \text{NodeDelta}(u_2)$;

  Accept with probability $\min(1, \exp(\Delta\ell))$ and update σ if accepted;

end

Phase 2 — sampling (accumulate statistics);

Set AvgLL $\leftarrow 0$, AvgGrad $\leftarrow 0$;

for $t = 1$ **to** T **do**

  Propose swap u_1, u_2 and apply Metropolis-Hastings accept/reject;

  if *accepted* **then**

    Incrementally update ℓ_σ and $\nabla_\Theta \ell_\sigma$ for swapped nodes;

  end

  AvgLL $\leftarrow$ AvgLL $+ \ell_\sigma$;

  AvgGrad $\leftarrow$ AvgGrad $+ \nabla_\Theta \ell_\sigma$;

end

return *AvgLL*$/T$, *AvgGrad*$/T$;

Before accumulating gradients, the MCMC chain runs for some W number of steps, where the swaps proposed are accepted or rejected the same way we discussed earlier, but nothing is recorded. This warm-up phase allows the algorithm to bring the permutation σ to a high likelihood region from the starting point.

Summary

The Kronecker graph model represents real-world network structure by repeatedly expanding a small initiator matrix Θ in a self-similar way. KronFit estimates the parameters of this model using three main ideas. First, the bit-decomposition trick reduces the cost of computing edge probabilities to $O(k)$. Second, the Taylor approximation simplifies the non-edge log-likelihood from $O(N^2)$ to $O(1)$. Third, Metropolis–Hastings MCMC sampling avoids checking all $N!$ permutations by focusing only on the high-likelihood regions of the permutation space. Together these reduce the overall fitting complexity to $O(E)$ per gradient step. Despite this, KronFit remains bound to sequential single-core execution, which becomes a practical bottleneck for large graphs. MFit directly addresses this limitation by offloading gradient computation to the GPU while retaining the same mathematical foundations established here.

Related Works

KRONFIT Leskovec and Faloutsos (2007) can fit Kronecker graphs by using maximum likelihood estimation and uses Metropolis sampling algorithms to reduce the cost of permuting over $N!$, and uses Taylor approximation to reduce cost for likelihood calculation from $O(N^2)$ to $O(E)$. Although this tool works very well for Kronecker graphs, it runs on a single CPU core which makes it inflexible for very large graphs in addition to working only for Kronecker graphs. MFit addresses these issues by planning to support arbitrary generative models and parallelize computation to make use of GPUs.

KRONEM Kim and Leskovec works very similarly to KRONFIT, but it uses the kronecker graphs with missing nodes and edges, fits the model, and uses the model to generate the missing parts of the graphs. It achieves this by combining Kronecker graph and Expectation Maximizing Framework to design a Metropolis and Gibbs Sampling approach for network estimation. Just like KRONFIT, KRONEM also runs on a single CPU core and does

not leverage hardware like GPU. It again only works for Kronecker graphs. On the other hand MFit can achieve the same result by fitting the network to a model and regenerating the entire graph from scratch on top of being massively parallelizable and will potentially work on general models.

MAGFIT Kim and Leskovec (2011) Multiplicative Attribute Graph (MAG) model extends graph modeling by considering a node with categorical attributes and calculates the probability of an individual edge as the product of the affinity values based on the attributes of the two nodes of that edge. It uses variational expectation maximization framework along with custom update rules which work very well for MAG model structure and is not transferable to other models. Additionally, it only runs on a single core, single thread just like the previous two tools. This gives MFit an advantage over MAGFIT both on its generalizability and scalability.

Chapter 3

MFit

As established in the background, KronFit is a very strong tool for fitting Kronecker graphs to real-world networks, but for massive datasets it faces scalability issues due to being purely single core, which is why we introduce **MFit**. MFit addresses this issue by leveraging modern hardware architecture. Unlike KronFit's hand-derived implementations, we use a hybrid approach that uses PyTorch's automatic differentiation to offload heavy analytical gradient computations to GPU while maintaining high-speed permutation sampling (MCMC) on the CPU using optimized C-compilation.

The main objective is similar to KronFit that is to find the initiator matrix Θ that maximizes $\ell(\Theta) = \log P(G|\Theta)$. The key difference is how the gradients are calculated and the way the parameters are updated. Instead of implementing Equation 2.18 by hand and employing a custom adaptive step-size update rule, MFit formulates the log-likelihood as a differentiable computation graph in PyTorch. Gradients are then computed automatically using autograd and passed to the Adam optimizer for parameter updates.

Before the fitting begins, MFit performs a set of pre-processing steps that are aligned with the initialization strategy of KronFit.

Graph Padding

The Kronecker model assumes that the number of nodes of a graph are exactly N_1^k for some integer k, where N_1 is the dimension of the initiator matrix. In practice, real-world graphs

rarely satisfy this condition. To address this, MFit pads the graph by adding isolated nodes until the total number of nodes in the graph reaches the next power of 2 Leskovec and Faloutsos (2007). The Kronecker depth is then defined as $k = \lceil \log_2 N \rceil$ ensuring compatibility with the recursive structure of Kronecker graphs.

Permutation initialization

As discussed in the background, the initial permutation has a significant effect on how quickly and effectively the Markov Chain reaches a high likelihood region. Following the approach used in KronFit Leskovec and Faloutsos (2007), MFit initializes the permutation σ by sorting the nodes in decreasing order of degree. Nodes with higher degrees are positioned at the top left positions of the Kronecker structure, corresponding to regions where the initiator matrix is expected to place higher probability values.

Parameter Scaling

The initial initiator matrix is first assigned a set of starting values before any scaling is applied. In general, these parameters can be initialized randomly. However, to stay consistent with the original KronFit implementation, we use its default initialization values of

$$\Theta^{(0)} = \begin{bmatrix} 0.9 & 0.7 \\ 0.5 & 0.2 \end{bmatrix}$$

Once initialized, the initial initiator matrix $\Theta^{(0)}$ is adjusted so that the expected number of edges generated by the model is consistent with the observed graph. Given a graph G with E edges, a scaling factor s is applied such that the expected edge count under the Kronecker model matches E Leskovec and Faloutsos (2007). The expected number of edges in a Kronecker model can be calculated using the following way as shown in Equation 2.11

$$\left(\sum_{i,j} \Theta_{ij}\right)^k = E_{expected} \tag{3.1}$$

then the scale factor is calculated as

$$s = \frac{E}{E_{expected}} \tag{3.2}$$

Each entry in Θ_{ij} is then replaced by $s \cdot \Theta_{ij}$ and the results are clipped between 10^{-4} and $1 - 10^{-4}$ to keep the probabilities well defined. Introducing this scaling factor provides a more balanced starting point for the optimization process and prevents the optimizer from spending the early iterations simply rescaling the parameters to a reasonable scale.

MCMC on CPU via Numba

The MCMC-based permutation sampling described earlier is executed on the CPU using a Numba-compiled kernel. The reason for this design is primarily due to the inherently sequential nature of the Metropolis–Hastings algorithm. Each proposed swap depends on the current state of the permutation, making it difficult to parallelize efficiently on a GPU without introducing complex synchronization and dependency issues. In contrast, compiling the loop with Numba allows us to achieve near C-level performance on the CPU while preserving a simple and reliable implementation.

To support efficient computation, the graph is stored in Compressed Sparse Row (CSR) format. Rather than using a dense adjacency matrix or padded edge structures, the graph is represented using two flat integer arrays: one encoding node offsets and the other storing adjacency lists. This representation enables constant-time neighborhood lookups while maintaining $O(E)$ memory usage. Such efficiency is critical for large sparse graphs, where dense representations would be very expensive.

At each MCMC step, a node swap is proposed using a biased sampling strategy. With 80 percent probability, a random edge is selected and its two endpoints are swapped. With the

remaining 20 percent probability, two nodes are chosen uniformly at random Leskovec and Faloutsos (2007). This edge-biased proposal focuses updates on structurally relevant node pairs, improving acceptance rates and helping the Markov chain mix more efficiently.

The change in log-likelihood for a proposed swap is computed incrementally. Only the edges incident to the two swapped nodes contribute to the update, as established earlier. This avoids recomputing the full likelihood over all edges, which would require $O(k \cdot E)$ time, and instead reduces the cost of each step to $O(k \cdot (deg(u) + deg(v)))$.

GPU Gradient Calculation

After every C MCMC steps (i.e., after processing a chunk), the current permutation is transferred from the CPU to the GPU, where the full log-likelihood is evaluated as a differentiable expression in PyTorch. This formulation directly follows from Equation 2.15.

The edge probabilities p_{uv} are computed using the same bit-decomposition method described in Equation 2.6, with the level counts n_{ab} corresponding directly to $c_{ab}(u, v, \sigma)$ from Equation 2.18, computed here via GPU-vectorized bit-wise operations across all edges simultaneously rather than sequentially per edge. The results are then cast to the appropriate floating-point type to form the log-probability. This approach prevents the repeated floating-point accumulation across k levels and helps prevent numerical underflow when working with small values.

Finally, calling `loss.backward()` on the negative log-likelihood automatically propagates gradients to the parameters θ_{00}, θ_{01}, θ_{10}, θ_{11} using PyTorch's autograd mechanism.

GPU Ensembling

Instead of executing all T MCMC steps and computing a single gradient at the end, MFit partitions these T steps into C equal chunks. After each chunk, the current permutation is transferred to the GPU, where the log-likelihood is calculated and the corresponding gradients are accumulated. Once all the chunks are done processing, the accumulated gradients

are divided by C to get their average before performing the optimizer update.

$$\nabla_\Theta \ell \approx \frac{1}{C} \sum_{c=1}^{C} \nabla_\Theta \ell \,\big|_{\sigma^{(c)}} \tag{3.3}$$

This follows directly from Equation 2.17, where the gradient used to update the Θ is formed by averaging over multiple permutations sampled across the Markov Chain. Averaging over C segments rather than a single one reduces the variance of the gradient estimate, which is especially helpful during the early stages when the chain has not yet fully mixed.

Adam Optimizer

Once the averaged gradient has been computed, MFit updates the initiator parameters using the Adam optimizer Kingma and Ba (2017). While the original KronFit implementation uses a custom adaptive step-size rule, Adam maintains an exponential moving averages of both the first moment (momentum) and the second moment (squared gradients) for each parameter, and uses these estimates to scale the update Kingma and Ba (2017).

$$m_p^{(i)} = \beta_1 m_p^{(i-1)} + (1-\beta_1) g_p, \qquad v_p^{(i)} = \beta_2 v_p^{(i-1)} + (1-\beta_2) g_p^2 \tag{3.4}$$

$$\theta_p^{(i+1)} = \theta_p^{(i)} + \alpha \cdot \frac{\hat{m}_p^{(i)}}{\sqrt{\hat{v}_p^{(i)}} + \epsilon} \tag{3.5}$$

Here $\hat{m}_p$ and $\hat{v}_p$ are the bias-corrected moment estimates where, α is the learning rate, and $g_p = \frac{\partial \ell}{\partial \theta_p}$ is the gradient of the log-likelihood with respect to the parameter θ_p. For MFit, β_1 is set to 0.5 instead of the more common value 0.9, while β_2 is set to 0.9. A smaller β_1 reduces the effect of momentum during the early stages of optimization, where gradients obtained from MCMC sampling tend to be noisiest, allowing the updates to respond better to new information. The β_2 value provides a stable estimate of the second moment while still adapting to changes in gradient magnitude. The learning rate is further decayed by a factor of 0.98 at each iteration to promote a stable convergence.

After each update, the parameters are clipped to the interval $[10^{-4}, 1 - 10^{-4}]$. This makes sure that all entries remain valid probabilities and avoids numerical issues in the log-likelihood computations near the boundaries. The full MFit fitting loop is summarized in algorithm 3.

Algorithm 3: MFit — Full Fitting Loop

Input: Graph G, initiator $\Theta^{(0)}$, iterations I, warm-up W, MCMC steps per iteration T, chunks C, learning rate α

Output: Fitted initiator $\hat{\Theta}$

Pad G to 2^k nodes; initialize σ by degree sort; scale $\Theta^{(0)}$;

Run W warm-up MCMC steps on CPU (burn-in, no gradient accumulation);

for $i = 1$ **to** I **do**

  Zero gradients;

  for $c = 1$ **to** C **do**

    Run T/C MCMC steps on CPU; transfer σ to GPU;

    Compute $\ell(\Theta)$ analytically on GPU; accumulate $\nabla_\Theta \ell$;

  end

  Average accumulated gradients by C;

  Perform Adam step on $\theta_{00}, \theta_{01}, \theta_{10}, \theta_{11}$; decay learning rate by 0.98;

  Clip all parameters to $[10^{-4}, 1 - 10^{-4}]$;

end

return $\hat{\Theta}$ *with highest observed* ℓ;

Summary

MFit updates the KronFit approach by taking advantage of modern CPU and GPU hardware. The fitting process is split into two parts based on what each stage does best. The MCMC permutation sampling runs on the CPU using a Numba-compiled kernel, since the sequential nature of Metropolis–Hastings works well on a single core with near C-level perfor-

mance. The gradient computation is moved to the GPU, where PyTorch's autograd replaces the manual gradient formulas used in KronFit, and uses vectorized bit-wise operations to compute edge probabilities across all edges simultaneously rather than sequentially. In addition to this, several improvements are made. Gradient variance is reduced by averaging over C permutation samples in each optimizer step instead of using just one. The Adam optimizer is used with a smaller momentum term $\beta_1 = 0.5$ which helps handle the noisy gradients from MCMC better than KronFit's custom update rule. The graph is stored in CSR format to support $O(E)$ memory usage and constant-time neighborhood lookups during MCMC. Together these design choices produce a fitting pipeline that is faster than KronFit on large graphs and lays the foundation for the general GSDL fitting framework described in the following section.

Chapter 4

High-Performance Parallel Synthetic Graph Generation

To evaluate MFit across 20 datasets, 20 values of k, and 30 samples per setup, we needed to generate a total of 12,000 synthetic graphs. Some of these graphs are extremely large, with hundreds of millions of nodes. Existing graph generators are either sequential or rely on specialized parallel tools, which makes them hard to use at this scale. To solve this, we built a parallel graph generator based on the R-MAT model Chakrabarti et al. (2004) that can run across multiple CPU cores without needing any external libraries.

R-MAT is a recursive model for generating graphs with realistic properties like power-law degree distributions, small diameters, and community structure. It uses four parameters a, b, c, d. For a given value of k, it creates a graph with $N = 2^k$ nodes. Each edge is placed by repeatedly dividing the adjacency matrix into four quadrants and choosing one based on these probabilities until a single position is reached. The quadrants a and d capture densely connected communities, while b and c model cross-community interactions, as illustrated in Figure 4.1.

The key contribution of our implementation is exploiting the recursive structure of R-MAT itself to achieve parallelism. Let's say we have some initiator Θ and we need to generate a Kronecker graph with some k value.

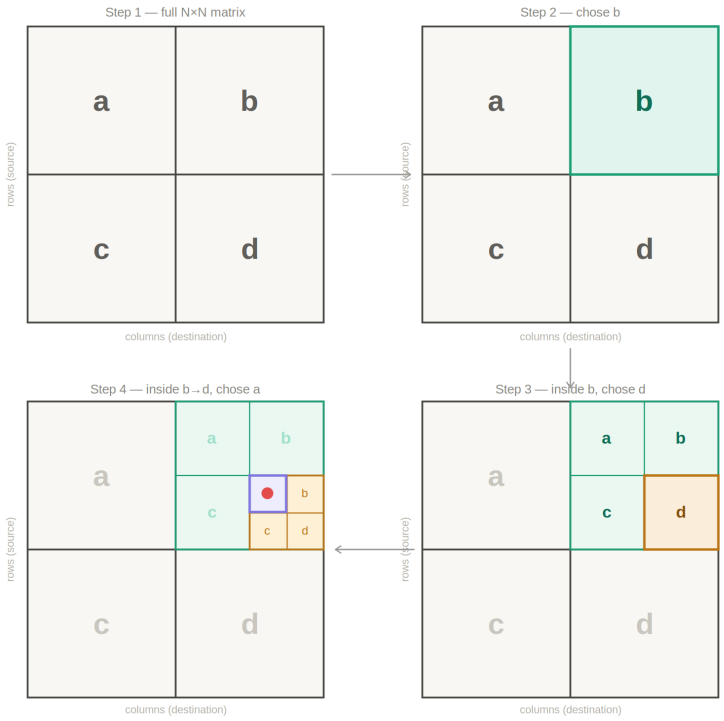


Figure 4.1: Recursive edge placement in R-MAT (Stochastic Kronecker Graphs)

$$\Theta = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

We already discussed that the expected number of edges E_{exp} in a Kronecker graph is just the $(a+b+c+d)^k$ as shown in Equation 3.1. We then normalize the 4 parameters so they satisfy the condition $a_n + b_n + c_n + d_n = 1$. We then multiply E_{exp} with all the four quadrants to get

$$\Theta = \begin{bmatrix} a \cdot E_{exp} & b \cdot E_{exp} \\ c \cdot E_{exp} & d \cdot E_{exp} \end{bmatrix}$$

When sufficient cores are available, these four quadrants can be considered as four sub-problems and can be dispatched as independent parallel tasks via Python's `ProcessPoolExecutor`. Each worker recursively applies the same logic within its assigned subregion, further subdividing as long as additional cores are available. Once parallelism

is exhausted, the remaining work is completed using a sequential R-MAT loop within the assigned sub-matrix defined by (u_{offset}, v_{offset}).

To reduce I/O overhead, each worker writes edges to a local temporary file using buffered output, flushing only after accumulating a large number of edges like 50,000. After all workers finish, the partial outputs are combined into a single edge list using an OS-level merge operation, which is efficient for large files. This implementation takes the 4 parameters (a, b, c, d), the scale k, and a random seed as an input, making it easy to generate large-scale graph experiments.

Experimental Setup

To generate synthetic graphs for the purpose of this experiment, we use the KronFit parameters reported by Leskovec at Leskovec et al. (2009) for 20 real-world datasets. These graphs cover a diversity of network types, such as online social networks, internet and peer-to-peer networks, and email networks. The estimated parameters for these 20 graphs are summarized below in Table 4.1

For each dataset, synthetic graphs were generated at scales $k = 1$ to $k = 20$, with node counts ranging from $N = 2$ up to $N = 877,324,679$ for the `Email-Inside` dataset at $k = 20$. Thirty independent samples were generated per $(dataset, k)$ pair using different random seeds, yielding $20 \times 20 \times 30 = 12{,}000$ graph instances in total. This level of replication provides stable estimates of fitted parameter distributions at each scale and allows us to characterize variability as a function of k. Graphs were also generated up to $k = 29$ during development, reaching 62 billion nodes for `CA-Hep-Ph`, confirming that the generator scales well beyond the range used in the experiments.

Summary

Generating 12,000 graph instances at large k would not be practical using a sequential generator. By using the recursive structure of R-MAT, our parallel implementation spreads

Table 4.1: Estimated parameters for each network

Network	a	b	c	d
AS-RouteViews	0.987	0.571	0.571	0.049
ATP-GR-QC	0.902	0.253	0.221	0.582
Bio-Proteins	0.847	0.641	0.641	0.072
Email-Inside	0.999	0.772	0.772	0.257
CA-GR-QC	0.999	0.245	0.245	0.691
AS-Newman	0.954	0.594	0.594	0.019
Blog-Nat05-6m	0.999	0.569	0.502	0.221
Blog-Nat06all	0.999	0.578	0.517	0.221
CA-Hep-Ph	0.999	0.437	0.437	0.484
CA-Hep-Th	0.999	0.271	0.271	0.587
Cit-Hep-Ph	0.994	0.439	0.355	0.526
Cit-Hep-Th	0.990	0.440	0.347	0.538
Epinions	0.999	0.532	0.480	0.129
Gnutella-25	0.746	0.496	0.654	0.183
Gnutella-30	0.753	0.489	0.632	0.178
Delicious	0.999	0.327	0.348	0.391
Answers	0.994	0.384	0.414	0.249
CA-DBLP	0.999	0.307	0.307	0.574
Flickr	0.999	0.474	0.485	0.144
Web-NotreDame	0.999	0.414	0.453	0.229

the work across multiple CPU cores without needing any external libraries. This significantly reduces generation time as more cores are available because it distributes the workload across CPU cores with no external dependencies. The final dataset, which includes 20 network types, 20 values of k, and 30 samples each, serves as the basis for all experiments in this chapter.

Chapter 5

Graph Structure Descriptor Language

While Stochastic Kronecker Graphs can be a very powerful tool for modeling real-world networks, they are severely limited by the structure of the networks. Real-world graphs can exhibit patterns that no single model like a simple Kronecker initiator can fully capture. To address this, we introduce Graph Structure Descriptor Language (GSDL) which is a domain-specific language designed to describe complex graph structures as compositions of simpler matrix operations. This makes it possible to not only generate sophisticated synthetic data but also be able to fit generalized models to real-world datasets. This section introduces the GSDL language, outlines the matrix classes it provides and describes how MFit is adapted into a general fitting framework called GSDLFit.

The GSDL Language

GSDL is designed as a compositional language that represents graph structures using a combination of structured matrices Veras (2025). The idea is similar to how certain compilers in signal processing like SPIRAL break down complex transforms into smaller, structured components Puschel et al. (2005). In a very similar way GSDL expresses an adjacency matrix as a composition of simpler blocks. This allows it to better exploit structures when selecting optimizations.

GSDL has two main layers: *Terminal operators* and *Higher-order operators*. *Terminal operators* are the basic matrix types which become the building blocks for a more complex

model. Then we have the *Higher-order operators* which combine these primitive models into a more expressive model. Table 5.1 lists the terminals and Table 5.2 the higher-order operators; both are reproduced from the GSDL specification Veras (2025).

Terminal	**Description**
$I(m)$	Identity matrix of size m
Dense($[a_{00} \ldots]$)	Dense matrix with explicit values
$R(\rho)$	Random matrix with entries drawn from f
$K^{(k)}(\theta)$	Kronecker graph $K(\theta)^{\otimes k}$
Mesh$(k, \ldots)$	k-dimensional mesh
Btrfly(θ), Monarch(θ)	Butterfly and Monarch matrices

Table 5.1: GSDL terminal operators Veras (2025).

Operator	**Description**
Sum(A, B)	Probabilistic union $A + B - A \cdot B$
Tensor(A, B)	Kronecker product $A \otimes B$
Mask(A, f)	Stochastic masking to produce a sparse graph
Model(A)	Designate A as an approximation of a real graph

Table 5.2: GSDL higher-order operators Veras (2025).

Matrix Classes in the Implementation

The implementation of GSDL follows an object-oriented design based on inheritance and abstraction. A base class is defined to introduce a key method for computing edge probabilities between node pairs. This design enables polymorphism, allowing different matrix classes that extend `BaseMatrix` to be used interchangeably during model fitting.

```
class BaseMatrix(MatrixModel, np.ndarray):

    def get_edge_probability(self, i, j):

        return self[i][j]
```

Several concrete matrix classes are built on top of this base. For instance, a random matrix assigns the same edge probability across all entries, making it useful as a simple

baseline or noise component. It is a uniform random graph where every entry takes the same probability ρ. It can be useful for introducing randomness into a model or acting as a baseline.

```
class RandomMatrix(BaseMatrix):
    def __new__(cls, shape, rho):
        symbols = np.full(shape, rho)
        ...
```

This is the class for the Kronecker matrix which is central to both MFit and its extensions.

```
class KroneckerMatrix(BaseMatrix):
    def __new__(cls, initiator, k):
        if k == 1:
            symbols = initiator
        else:
            symbols = np.kron(initiator, KroneckerMatrix(initiator, k-1))
        ...
```

MAGMatrix or the multilinear attribute graph is a more flexible variant which allows different matrices to be used at each level of the Kronecker construction, enabling more complex structures.

```
class MAGMatrix(BaseMatrix):
    def __new__(cls, initiator_list):
        if len(initiator_list) == 1:
            symbols = initiator_list[0]
        else:
            symbols = np.kron(initiator_list[0],
                              MAGMatrix(initiator_list[1:]))
        ...
```

Beyond these, additional operators allow combining matrices in meaningful ways. For example, one operator computes the probabilistic union of two matrices, while another represents their intersection. There is also a permutation operator that reorders nodes, which is particularly important since permutation plays a central role in the MCMC sampling process used during fitting.

In addition to these there are more operators that allow combining matrices in meaningful ways. For example `OpKroneckerProduct` takes two different matrices and combines them using a Kronecker product in a single step, instead of repeatedly using the same matrix. This allows different types of structures to be mixed together. The `OpUnion` operation that combines two matrices by computing the probability that an edge exists in at least one of them. Given two independent edge probability matrices A and B, the union probability is:

$$\text{Union}(A, B)_{ij} = A_{ij} + B_{ij} - A_{ij} \cdot B_{ij}$$

```
class OpUnion(BaseMatrix):
    def __new__(cls, lhs, rhs):
        symbols = lhs + rhs - (lhs * rhs)
        ...
```

We also have the following: `OpIntersection` computes the probability that an edge exists in both matrices by multiplying their values, $A_{ij} \cdot B_{ij}$. This corresponds to combining models in a conjunctive manner, where an edge is likely only if both matrices assign it a high probability.

`OpPermuteMatrix` applies a permutation σ to reorder the nodes of a matrix, represented as $PAP^\top$, where P is the corresponding permutation matrix. This is the same type of permutation explored by the MCMC sampler in MFit, now expressed as an explicit GSDL operation.

`Composite Models:` By combining the above operations, GSDL can represent a wide

range of graph models. This gives it the flexibility to go beyond traditional Kronecker-only approaches and better capture the diversity seen in real-world networks. The models currently implemented in the wrapper layer and used in GSDLFit are summarized in Table 5.3.

Model	**GSDL Expression**	**Parameters**
Kronecker	$K^{(k)}(\Theta)$	4
Kron (fixed p_{00})	$K^{(k)}(\Theta)$, p_{00} fixed	3
Kron + Kron (1 perm)	$\mathrm{Sum}(K^{(k)}(\Theta_1), K^{(k)}(\Theta_2))$	8
Kron + Kron (2 perm)	$\mathrm{Sum}(P_1 K^{(k)}(\Theta_1) P_1^\top,\ P_2 K^{(k)}(\Theta_2) P_2^\top)$	8
Kron + Random	$\mathrm{Sum}(K^{(k)}(\Theta), R(\rho))$	5

Table 5.3: Composite graph models expressible in GSDL and implemented in the wrapper layer.

The Wrapper Interface

To connect these model descriptions with the fitting algorithm, a lightweight *wrapper* interface is used. This wrapper is a Python object that has the following three methods:

- `get_num_parameters()`: returns the number of scalar parameters the model requires.
- `get_perm_sizes()`: returns a list of permutation lengths, one per MCMC chain. Standard Kronecker returns `[`2^k`]`; Kron+Kron will have two independent permutations and returns `[`2^k`, `2^k`]`.
- `create_from_parameter_vector(vect, perm_list)` : Builds the full GSDL probability matrix from a parameter vector and a set of permutations, returning a `BaseMatrix` that can be used to compute edge probabilities using `get_edge_probability(u,v)`.

An optional fourth method, `get_est_ll_empty_graph(vect)`, is also implemented, which is used to improve performance. For models like pure Kronecker models, where the log-likelihood of the empty graph has a closed-form expression, it is computed directly using a Taylor approximation.

$$\ell_{empty} \approx -\left(\sum \mathcal{P}\right)^k - \frac{1}{2}\left(\sum \mathcal{P}^2\right)^k$$

For models where this approximation does not apply, such as Kron+Random, the method returns `None` and the fitting engine falls back to explicit non-edge sampling. This single boolean dispatch is what makes GSDLFit model-agnostic without paying the $O(N^2)$ enumeration cost for the common Kronecker case.

The implementation of the Taylor approximation trick for Kronecker Graphs is shown below:

```
class GSDLWrapperKron:
    def __init__(self, k, shape):
        self.k, self.shape = k, shape

    def get_num_parameters(self):
        return self.shape[0] * self.shape[1]          # 4

    def get_perm_sizes(self):
        return [pow(self.shape[0], self.k)]            # [2^k]

    def create_from_parameter_vector(self, vect, perm_list):
        initiator = vect.reshape(*self.shape)
        mat = KroneckerMatrix(initiator, self.k)
        return OpPermuteMatrix(perm_list[0], mat)

    def get_est_ll_empty_graph(self, vect):
        sum_P  = np.sum(vect)
        sum_P2 = np.sum(vect ** 2)
```

```
return -(sum_P ** self.k) - 0.5 * (sum_P2 ** self.k)
```

GSDLFit: MFit as a General Fitting Framework

GSDLFit extends MFit by replacing its hardcoded assumptions about Kronecker graphs with this wrapper-based design. Instead of relying on specialized logic tied to a specific model, the fitting process now interacts with the wrapper to evaluate probabilities and compute likelihoods. This allows the same optimization pipeline to be reused across different models with minimal changes.

What Changed from MFit to GSDLFit

Several key modifications were needed to generalize MFit into GSDLFit.

1. The MCMC kernel is wrapper-driven. In MFit, the Metropolis–Hastings step computes $\Delta\ell$ using `get_kron_log_prob()`, a Numba function that exploits Kronecker structure.

In GSDLFit, $\Delta\ell$ is computed using `wrapper.create_from_parameter_vector()`, followed by `get_edge_probability(u,v)` on the resulting matrix. For Kronecker models, the Taylor-based shortcut is preserved via `get_est_ll_empty_graph` even with the wrapper. For general models, the method falls back to explicit non-edge sampling using

`get_est_loglikelihood_nonedge`.

2. Multiple permutation chains. MFit maintains a single permutation vector. GSDLFit reads `wrapper.get_perm_sizes()` at initialization and allocates one chain per entry. For example, the Kron+Kron-2perm model requires two independent chains of length 2^k, each sampling a distinct node-to-position mapping. MCMC then runs over all chains in each iteration.

3. Parameter count is dynamic. MFit hardcodes four parameters since it works only for Kronecker Graphs, where GSDLFit calls `wrapper.get_num_parameters()` and builds a `torch.nn.ParameterList` of that length. This lets the optimizer adapt automatically to models with multiple parameters.

4. Gradient computation is Kronecker-specific. The GPU likelihood function `calc_ll_torch` in GSDLFit still uses the bit-decomposition approach so it produces gradients only for the first four parameters. For models with more than four parameters, such as Kron+Kron, which has eight parameters, the additional parameters are updated only through the MCMC sampling of the permutation. Implementing support for full autograd for general GSDL models is left as future work (Section 5).

Running GSDLFit

The following is all someone needs to do to fit their graph to some model. The user can select a model name from the command line. A wrapper is created internally from the model name and k, and then used to create the `GSDLFit` object.

```
fitter = GSDLFit(
        graph_file_path = args.graph_file,
        model_name = args.model,
        fixed_param = args.fixed_param,
        iterations = args.iterations,
        warmup_mcmc = args.warmup,
        mcmc_per_iter = args.mcmc_per_iter,
        learning_rate = args.lr,
        non_edge_samples = args.non_edge_samples
    )
    best_params, best_ll = fitter.fit()
```

The user just needs to change the `model_name` to switch to a more complex model.

Generating Synthetic Graphs

Once the parameters are fitted, the GSDL expression can be used to generate synthetic graphs by applying `RandomMask` to the probability matrix:

```
initiator = BaseMatrix([[0.9, 0.7],
                        [0.5, 0.1]])
prob_matrix = KroneckerMatrix(initiator, k)
adj_matrix  = RandomMask(prob_matrix, seed=42)
```

The resulting `.mtx` file is in Matrix Market format, which is a standard sparse matrix representation supported by many graph analysis tools. Now you can easily generate synthetic graphs and use them with other tools.

Limitations and Future Work

GSDLFit demonstrates that the CPU–GPU pipeline used in MFit can be extended to support general GSDL models with relatively small changes, hence expanding its application to all sorts of complex models. However, there are still a few important limitations.

First, `create_from_parameter_vector` builds the full $N \times N$ probability matrix via `np.kron`, which is practical only for small graphs. For higher k values at production scale, the MCMC kernel calls this function once per proposed swap, making it a significant bottleneck for large N. A more efficient approach would avoid constructing the full $N \times N$ matrix and instead compute edge probabilities $P_\sigma(u, v)$ directly from the parameter vector whenever they are needed. This strategy would eliminate the need to materialize the entire matrix and is left as future work.

Second, the gradient computation in `calc_ll_torch` still relies on the Kronecker bit-decomposition trick. Because of this, gradients are only computed for the first four param-

eters. For models like Kron+Kron, which have eight parameters, only the parameters of the first Kronecker component are updated using gradients. The remaining parameters are adjusted only through permutation sampling in the MCMC step. To fix this, the Union log-likelihood would need to be expressed in a fully differentiable way, either by writing a custom autograd function or by reformulating the Union operation in PyTorch. This is a natural next step for improving GSDLFit.

Third, the current wrapper interface assumes that all initiator matrices are of size 2×2. While this simplifies the implementation, it limits the types of models that can be expressed. Allowing larger or more general initiator shapes could make it possible to capture even more complex patterns found in real-world graphs.

Summary

GSDL introduces a way to describe graph structure using combinations of structured matrix building blocks. GSDLFit builds on this by extending MFit's CPU–GPU pipeline so it can work with any model that can be expressed in this language. The main idea that makes this possible is the wrapper interface. By hiding model-specific details behind a small set of methods, the same MCMC sampling and gradient optimization process used in MFit can be reused across different models. This means the core algorithm does not need to change whether we are fitting a Kronecker graph, a combination of multiple Kronecker graphs, or a Kronecker graph with added randomness. As a result, MFit can be seen not just as a tool for Kronecker graphs, but as the starting point of a more general fitting framework. In this framework, the Kronecker model acts as the baseline, while GSDL allows more complex graph structures to be built and fitted in a consistent way.

Chapter 6

Experiments

In this chapter, we will do rigorous experiments on MFit on major areas like accuracy, computational performance and efficiency, and scalability. We will do stress testing on how much we can push in some aspects. We will do the following set of experiments:

- **MFit vs KronFit:** In this section we will do head-to-head comparisons using the synthetic datasets generated from parameters mentioned in *Kronecker graphs: An Approach to Modeling Networks* Leskovec et al. (2009). We will evaluate the performance and the accuracy to ensure that MFit's speed does not compromise the model's ability to accurately fit the parameters.

- **Optimizers:** We explore three different optimizers including Adam, AdamW, and RMSProp for MFit. This experiment will analyze their impact on convergence behavior and the stability of parameter estimates.

- **Threads Scaling:** We will analyze the performance gains with different thread counts and will identify the bottlenecks that bound the further speedup, drawing on Amdahl's law.

- **Data Type Precision:** We will study the impact of reduced precision for the data representation in MFit, and analyze the trade-offs between numerical stability and performance.

Experimental Environment

All MFit vs KronFit experiments were conducted on the Schooner HPC cluster at the University of Oklahoma. KronFit was run on the `32gb_20core` partition, which uses Intel Haswell CPUs with 20 cores per node and 32 GB of system memory. MFit was run on the `sooner_gpu_test_fp64` partition, which uses Intel Sapphire Rapids CPUs with 64 cores per node, 512 GB to 1 TB of system memory, and 2 to 4 NVIDIA H100 GPUs per node. All remaining experiments, including the optimizer comparison, thread scaling, and data type experiments, were conducted on a local machine equipped with an AMD Ryzen 7 1700X eight-core processor running at up to 3.4 GHz with 16 logical threads.

MFit vs KronFit

To validate the accuracy and performance of MFit, we compared it with the original C++ KronFit implementation Leskovec and Faloutsos (2007) across the different synthetic datasets we mentioned in the earlier section. While 20 datasets were tested on, we will report only four representative datasets: Blog-Nat06all, CA-GR-QC, and Cit-Hep-Ph.

Blog-Nat06all was chosen because it had the best performance and accuracy where CA-GR-QC had the worst. AS-RouteViews was chosen because it was the primary dataset analyzed by Leskovec in Leskovec et al. (2009). The results and plots for the rest of the datasets will be included in the Appendix.

Blog-Nat06all Dataset

Blog-Nat06all shows has shown very stable results for both KronFit and MFit. It also shows clearly how MFit's GPU-Accelerated engine outperforms the sequential limitations of the original C++ KronFit implementation in both performance and accuracy.

The Bottom left plot in Figure 6.1 shows the total run time of KronFit and MFit across 20 k levels. The run time for MFit can be seen slowly increasing as the k level increases

which is expected since the graph size grows exponentially with k (i.e, $N = 2^k$), leading to more edges and higher computational cost. Even at $k = 20$, where the datasets have more than 19 million edges, MFit takes roughly 1250 seconds. On the other hand, the run time for KronFit seems to be increasing exponentially with k. The gap between KronFit and MFit starts getting wider and significant at roughly k 14. At k 20, the run time for KronFit is around 10 times more than of MFit and this gap will exponentially get bigger as k increases. This experiment demonstrates MFit's superior performance due to its hybrid architecture in contrast to sequential KronFit.

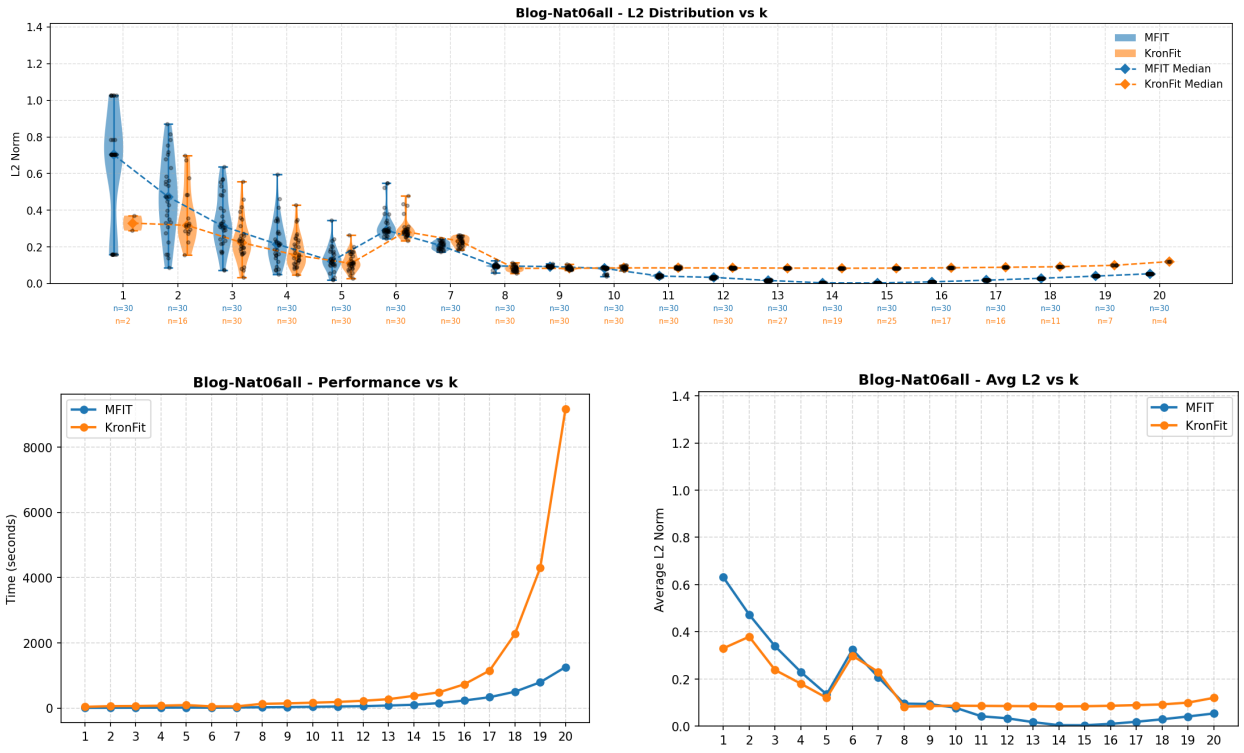


Figure 6.1: Results for the Blog-Nat06all dataset. The top plot shows the distribution of L2 error across 20 runs for MFit and KronFit at each value of k, visualized using violin plots. The bottom-left plot compares total runtime (in seconds) as a function of k for both methods. The bottom-right plot shows the average L2 error versus k illustrating the trend in parameter recovery accuracy across increasing Kronecker scales. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

The Average L_2 vs k plot confirms that the significant performance difference wasn't the only win for MFit. For $K < 5$, both models start with high average L_2. After $k = 5$,

they seem to be identical, but after k 10, MFit starts outperforming KronFit again by a wide margin. At some k levels, MFit seems to have almost zero L_2 norm values showing perfect reconstruction of the initiator matrix that was used to generate this data set. For higher k, average L_2 for both KronFit and MFit seems to be increasing which is expected because due to the exponential increase in size, small changes in Θ get amplified since they lead to larger deviation which makes the optimization very sensitive. When the permutation space becomes very large the Markov chain explores fewer permutations resulting in noisier gradients which worsens convergence. Since the model builds matrices recursively, even small changes in parameters affect massive number of edges increasing L_2 error. But even as the average L_2 increases, MFit still seems to be performing much better than KronFit.

The Violin plot further validates the potential and stability of MFit. At low k values, the distribution for MFit seems to be much taller reflecting higher variance but if you look at the color coded number of samples under the k values axis, MFit has perfectly 30 samples for each k value means it successfully predicted the parameters for all the 30 different graphs generated at that k level. But for KronFit, at lower k and high k values, the number of samples seems to be dropping and some of them are as low as 2. This tells us that KronFit has failed to estimate the parameters on those datasets. With more experiments and analysis, we could have found out why KronFit is failing for some of them but due to the massive number of datasets we are working with, it was not possible to look at individual graphs for further analysis. Additionally with increasing k values the distribution for both models seems to get narrower and almost merging together at higher k values resulting in the dark spots on the plot, which is a sign of stability.

Overall we can safely conclude that MFit was able to outperform Kronfit both in run time and accuracy. Due to some of the heavy lifting being done by GPU for MFit, the performance gap will get even wider for higher k values. In terms of accuracy, even though MFit starts with higher L_2 norms, it very quickly outperforms KronFit for $k > 10$.

CA-GR-QC Dataset

Unlike Blog-Nat06all, CA-GR-QC turned out to be the most challenging dataset for both MFit and KronFit. Both models struggle for low k values in terms of accuracy which worsens as k values increases.

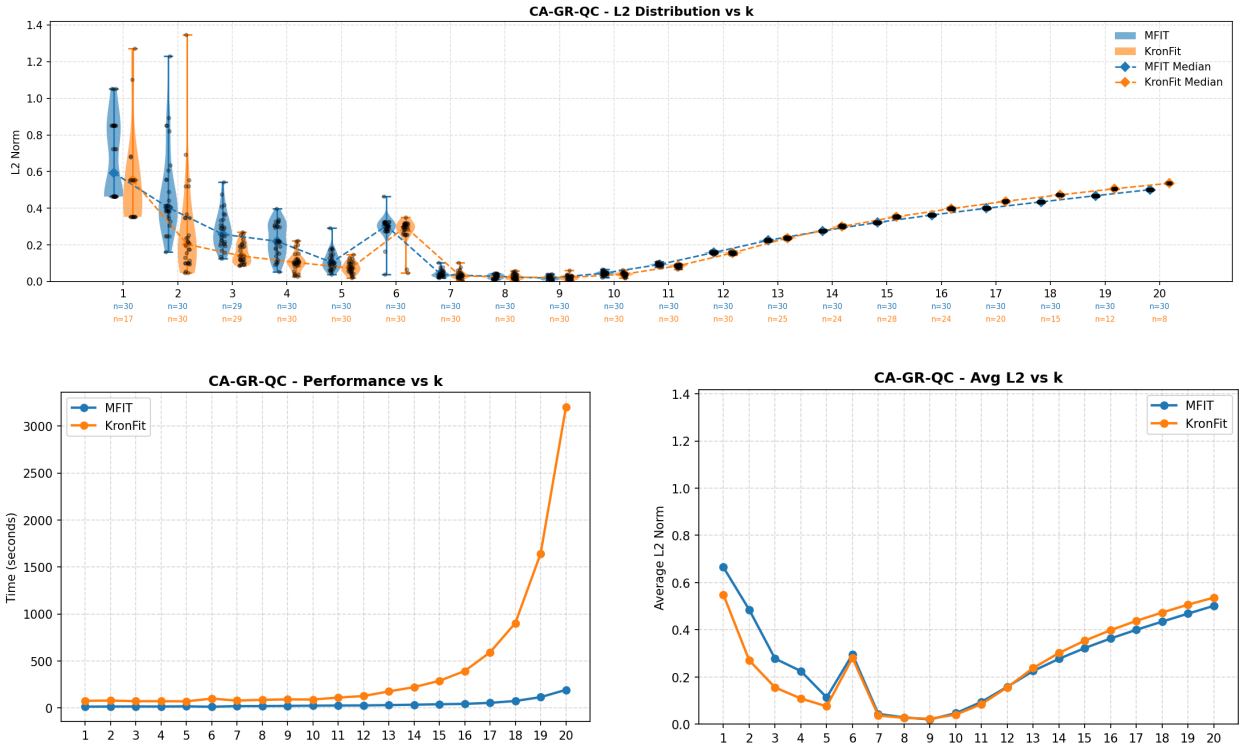


Figure 6.2: Results for the CA-GR-QC dataset. The top plot shows the distribution of L2 error across 20 runs for MFit and KronFit at each value of k, visualized using violin plots. The bottom-left plot compares total runtime (in seconds) as a function of k for both methods. The bottom-right plot shows the average L2 error versus k illustrating the trend in parameter recovery accuracy across increasing Kronecker scales. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

The bottom left graph of Figure 6.2 shows total run-time with respect to k. Similar to the previous dataset, initially the run time for both models is very similar even though MFit seems to be faster than KronFit. But their paths seems to diverge after $k > 12$. MFit seems to increase roughly linearly but KronFit's run-time shoots up exponentially creating a massive gap at $k = 20$. At this k, MFit seems to be approximately 8 times faster than KronFit. This confirms that MFit's GPU-accelerated architecture is well suited for handling

large graphs that a sequential model struggles with.

The average L_2 plot shows an interesting phenomenon. At $k = 6$, both models encounter a sharp spike in L_2 values. This further proves the correctness of MFit because it seems to replicate KronFit's behavior very accurately even in this degraded regime.

The violin plot further validates our point. At higher k values, the distribution gets very small both in height and width confirming the stability of our model and Adam optimizer. The dark clustering tells us that the degradation here is a systematic convergence issue rather than the failure of the individual models since both model seems to reliably find the same solution. Similar to the previous dataset, MFit successfully completes 30 out of 30 samples for every k level. On the other hand, KronFit's sample count fluctuates throughout the k levels even dropping to 8 for $k = 20$. This indicates that KronFit often fails to converge or crashes for some of the graphs where MFit is more robust and is able to predict the parameters for every dataset. When the output and error file was checked for a few of the KronFit samples they did not show anything. This hints towards an infinite loop but further experiments are required to fully grasp what causes KronFit to crash.

Overall just like the Blog-Nat06all dataset, MFit outperforms KronFit in both dimensions, accuracy and performance. MFit's performance advantage is consistent and faithfully mirrors KronFit.

AS-RouteViews Dataset

This dataset serves as a critical validation point because it was one of the primary real-world networks analyzed by Leskovec in his original paper Kronecker graphs: An Approach to Modeling Networks Leskovec et al. (2009). MFit's performance and accuracy on these datasets demonstrates that our GPU-Accelerated framework remains faithful to the foundational logic of Kronecker graphs and its generative logic.

The performance plot on the bottom right of Figure 6.3 again confirms the computational efficiency of MFit as the graph scales. MFit and KronFit perform very closely to each

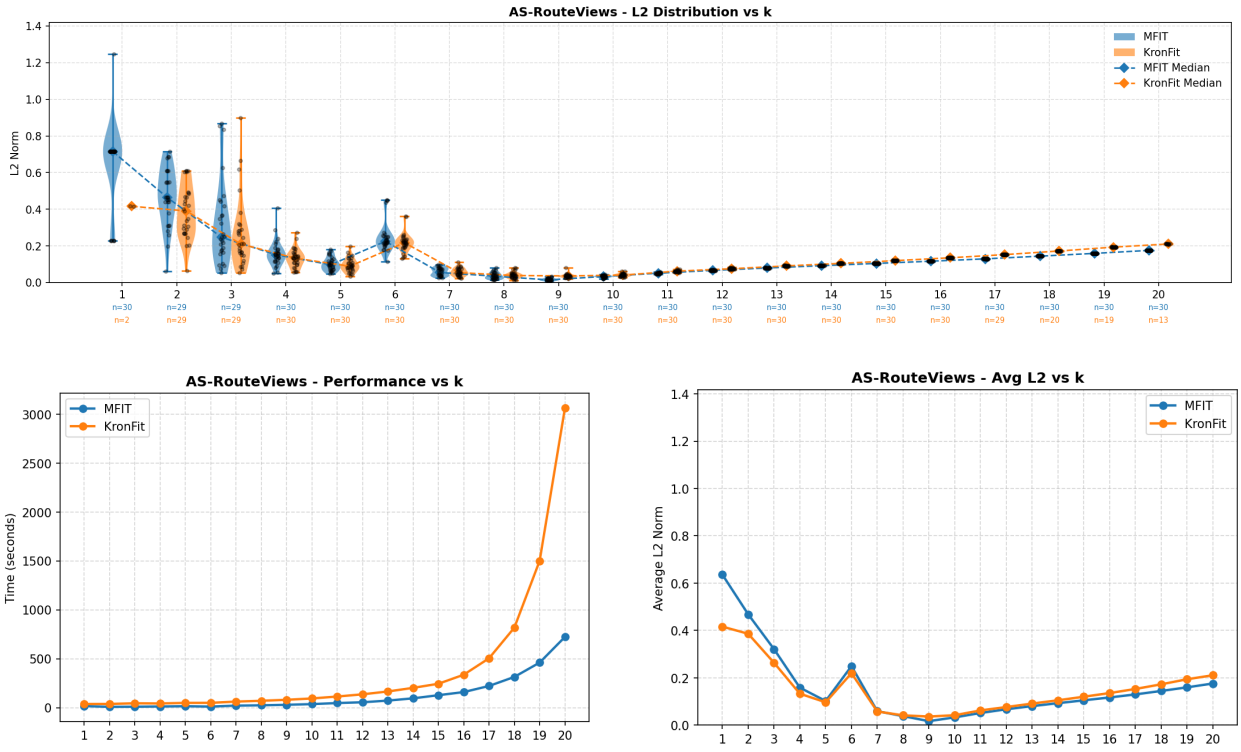


Figure 6.3: Results for the AS-RouteViews dataset. The top plot shows the distribution of L2 error across 20 runs for MFit and KronFit at each value of k, visualized using violin plots. The bottom-left plot compares total runtime (in seconds) as a function of k for both methods. The bottom-right plot shows the average L2 error versus k illustrating the trend in parameter recovery accuracy across increasing Kronecker scales. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

other at lower k values, but after $k = 16$ KronFit's total run-time shoots up exponentially. Performance for MFit also starts degrading at higher k values and it seems to increase more as compared to the previous two datasets we analyzed. But the performance difference between KronFit and MFit at $k = 20$ is approximately 4 which is still significant considering this ratio will increase even more as k increases.

The average L_2 vs k plot show that both models begin with high error values and their errors both seems to decrease steadily until $k = 5$. The L_2 values again seems to spike at $k = 6$ which is not unique to this dataset. We observed the same phenomenon with the previous two datasets. In fact this trend appears consistently among the other datasets we analyzed, including the ones in the Appendix. So we looked more into it and after some digging we found out that the consistency of this issue is not a random glitch but the stochastic nature of R-MAT at small scales.

The root cause of the $k = 6$ becomes clear when we look at the adjacency structure of the generated graphs directly. Figure 6.4 shows spy plots of the generated graphs for AS-RouteViews at $k = 5$, $k = 6$, and $k = 7$. At $k = 5$, the graph has 29 active nodes and 59 edges. At $k = 7$, the graph maintains the trend and has 117 nodes and 223 edges. But at $k = 6$, something interesting happens. The graph has only 34 active nodes and 12 edges. It has even fewer edges than the graph for $k = 5$ despite having a higher k value which should have higher number of expected edges. Nearly all edges are concentrated in column 0 or row 0 meaning that the generated graph is nearly empty and is more sparse than the graph for $k = 5$.

This behavior is most likely due to how the R-MAT generator works. In simple terms, R-MAT generates edges by repeatedly throwing a dart at a dartboard divided into regions, where each region corresponds to a quadrant of the initiator matrix and larger probability values mean bigger regions. For small k values the total number of possible nodes 2^k is small, so even a modest number of darts lands on meaningful nodes and produces a reasonably connected graph. However when k increases from 5 to 6, the dartboard doubles in size

while the number of darts grows only modestly. Combined with the fact that the initiator probability is heavily concentrated in one corner, most of the dartboard represents low-probability empty space and most darts land nowhere meaningful, producing a nearly empty degenerate graph as seen in Figure **??**. For $k > 6$, the expected number of edges grows large enough to compensate for the expanded node space and the generator recovers. This makes $k = 6$ a particularly unstable scale (I call it **The Dead Zone**) for R-MAT generation with these initiators, and the consistent appearance of this spike across all 20 datasets in our benchmark suggests it is a general property of Kronecker graph sampling at this scale rather than anything specific to AS-RouteViews.

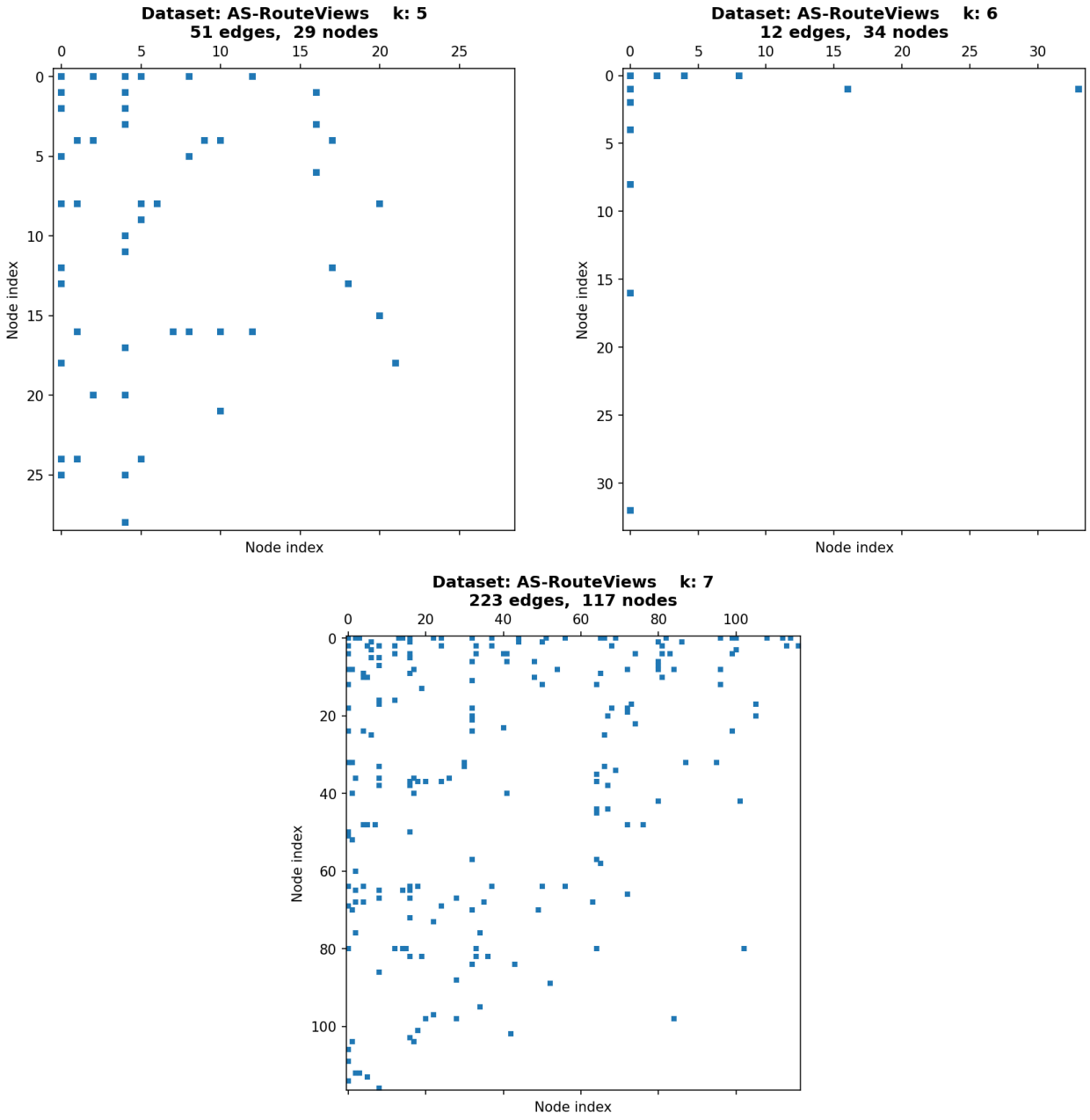


Figure 6.4: AS-RouteViews Spy plots for k=5, k=6, and k=7

Summary

Across all datasets we analyzed, MFit consistently outperforms KronFit in performance while maintaining superior parameter recovery. The performance advantage for MFit grows for high k values and the total run-time difference between KronFit and MFit will continue to widen with the increase in k as shown for the above datasets. In terms of accuracy, both methods track each other closely across most k values, validating that MFit's heterogeneous pipeline faithfully reproduces KronFit's parameter recovery, and has proven itself to be a high-performance alternative to KronFit without sacrificing accuracy. The consistent $k = 6$ spike across all datasets is a systematic property of the R-MAT generator at this particular scale, which we refer to as the **Dead Zone**, and it affects both methods equally. Full results for all 20 datasets are provided in the Appendix.

Optimizers

MFit uses Adam as its default optimizer due to its adaptive step sizes and momentum mechanism which is well-suited for handling noisy gradients like in MCMC. In this section, we evaluate two alternative optimizers, AdamW and RMSprop to see how changing the optimizers affects the convergence and the performance of the model. For the purpose of this experiment, all analyses are done on the Blog-Nat06all and CA-GR-QC datasets. These datasets were chosen because Blog-Nat06all gives very low errors and CA-GR-QC yields very high errors, so we wanted to test them on the extremes. This also helps us remain consistent with the previous sections, to enable direct comparison.

AdamW

AdamW Loshchilov and Hutter (2019) is a version of Adam optimizer that modifies Adam by correcting how weight regularization is applied. In Adam, the L_2 regularization is usually added directly to the loss function. As a result the adaptive learning rate also ends up

getting scaled by it. This causes parameters with larger accumulated gradients to receive weaker regularization than they would under Stochastic Gradient Descent (SGD). AdamW resolves this issue by separating weight decay from the gradient update, directly applying it to the parameters instead: $\theta_t \leftarrow \theta_{t-1} - \eta_t(\cdots + \lambda\theta_{t-1})$. This ensures that the regularization strength λ is independent of the learning rate, simplifying tuning and improving consistency. In the case of MFit, this is very important because gradients from MCMC sampling can vary greatly across parameters.

RMSprop

In addition to AdamW, we also tried MFit with RMSprop Shi et al. (2021). This optimizer differs from Adam in a way that it removes the first moment term, similar to setting $\beta_1 = 0$ in Adam. Instead, it adapts the learning rate based solely on a moving average of squared gradients. Since MFit already reduces Adam's β_1 from 0.9 to 0.5 to limit instability from early iterations, evaluating RMSprop allows us to examine the effect of removing momentum entirely.

Blog-Nat06all Dataset

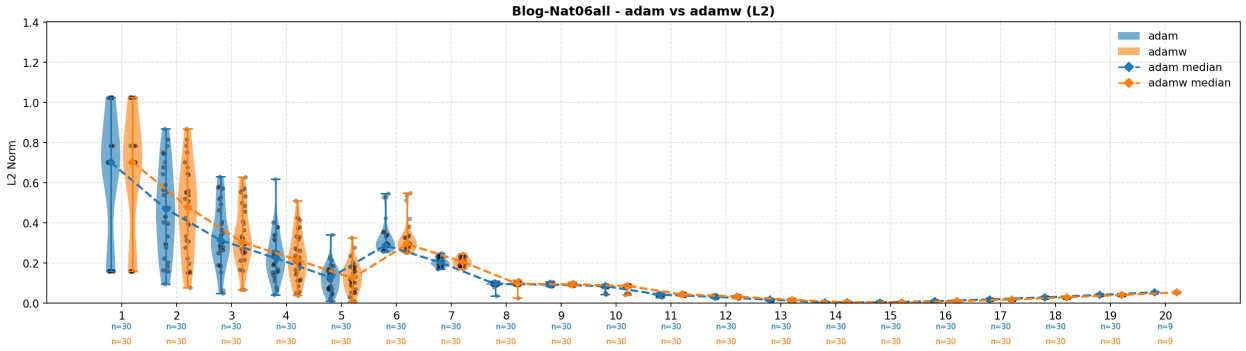


Figure 6.5: Shows the distribution of L2 error for Blog-Nat06all across 30 runs for Adam vs AdamW at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

From Figure 6.5 we can see that Adam and AdamW perform almost identically. You can

see it from similar looking distribution and median line across all k values. Even the width and the height of the distributions seem to be similar for most of the k values.

Both optimizers quickly converge to near-zero L_2 error from $k = 8$ onward, as confirmed by the average L_2 plot Figure 6.7. The spike observed at $k = 6$ appears in both cases, with recovery occurring by $k = 8$. Overall, AdamW does not provide a clear advantage here, likely because the dataset is sufficiently large and dense that regularization has limited influence on optimization.

RMSprop shows noticeably worse performance in terms of accuracy. Both the violin plot in Figure 6.6 and the average L_2 Figure 6.7 curve show slower convergence compared to Adam. While Adam achieves near-zero error by $k = 8$, RMSprop maintains higher error values until approximately $k = 11$ or $k = 12$, with significantly wider distributions. Adam is showing the usual spike at $k = 6$ but interestingly the L_2 for RMSprop drops at that k level. But it's worth noting that it still stays above Adam. RMSprop also shows more fluctuations at lower k levels. It's not until $k = 16$ that it becomes more stable and matches the accuracy of Adam.

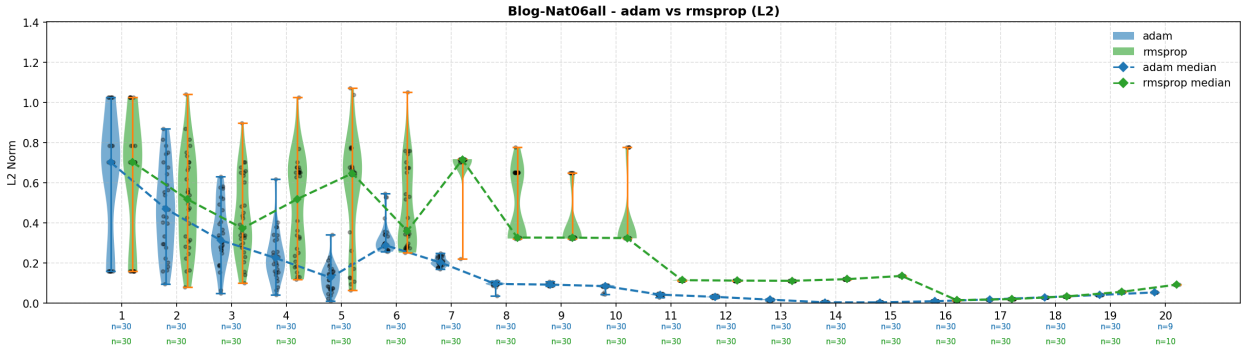
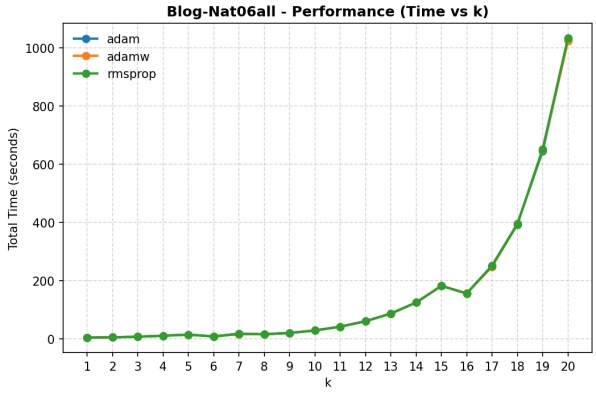
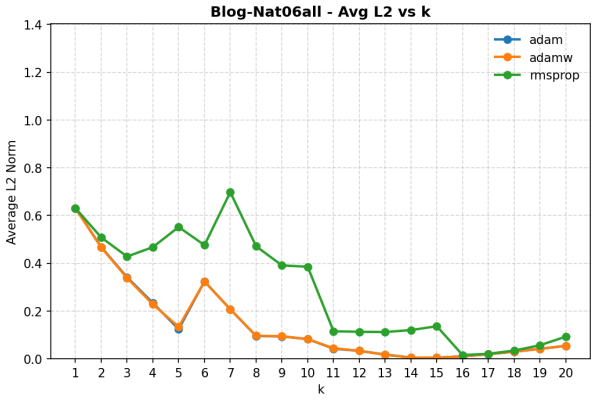


Figure 6.6: Shows the distribution of L2 error for Blog-Nat06all across 30 runs for Adam vs RMSprop at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

Runtime remains exactly the same across all optimizers, as expected. They overlap at every single k level.



(a) Run-time vs k for Adam, AdamW, and RMSprop



(b) L_2 vs k for Adam, AdamW, and RMSprop

Figure 6.7: The left plot compares total runtime (in seconds) for Blog-Nat06all as a function of k for all optimizers. The right plot shows the average L2 error versus k illustrating the trend in parameter recovery accuracy across increasing Kronecker scales.

CA-GR-QC Dataset

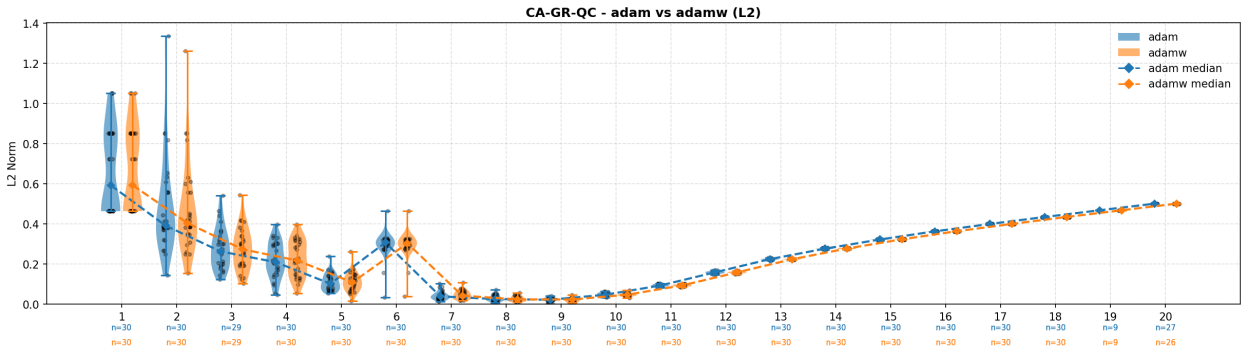


Figure 6.8: Shows the distribution of L2 error for CA-GR-QC across 30 runs for Adam vs AdamW at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

A similar trend is observed on CA-GR-QC. We can see in Figure 6.8 Adam and AdamW closely follow each other across all k values, both decreasing in error up to $k = 5$ with the same spike at $k = 6$ and the average L_2 slowly increasing for higher k values. The average L_2 curves are nearly indistinguishable, and the distribution shapes remain the same as well. This suggests that the high-k degradation is driven by the dataset rather than optimizer choice.

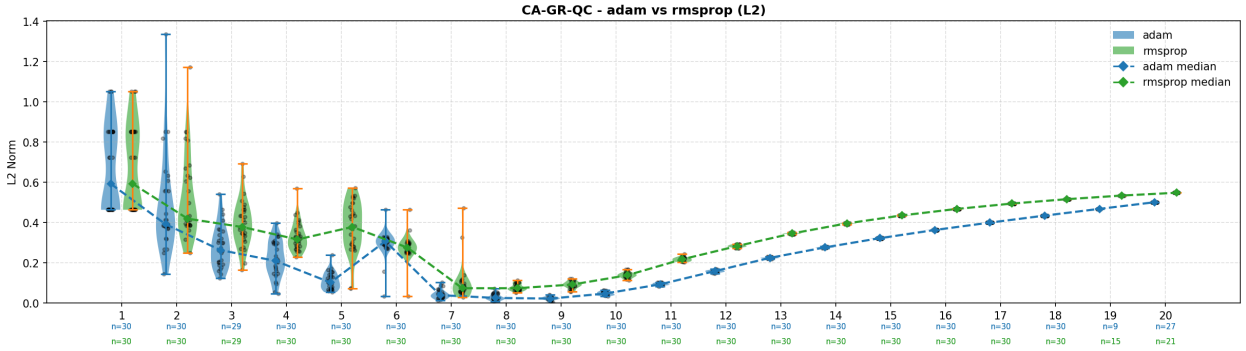
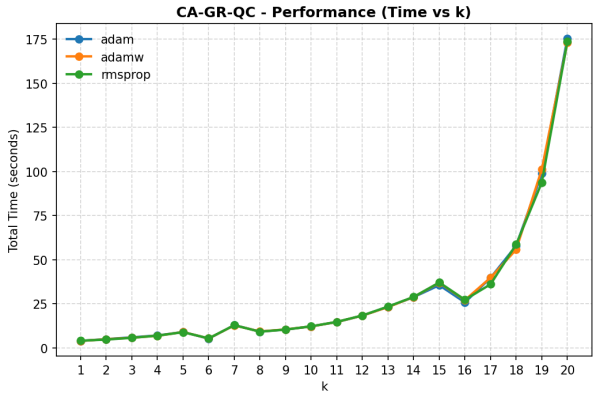


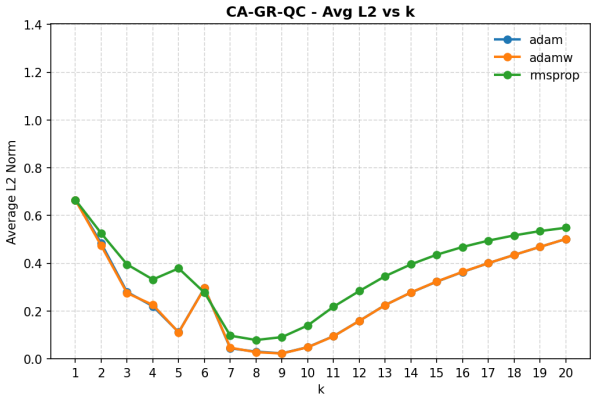
Figure 6.9: Shows the distribution of L2 error for CA-GR-QC across 30 runs for Adam vs RMSprop at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

Figure 6.9 shows that on CA-GR-QC, RMSprop consistently produces higher L_2 error than Adam across most values of k. The average error curve remains above Adam's from $k = 3$ onward, and the gap stays for increasing k but slowly seems to be decreasing towards the end as seen in Figure 6.10. The distributions are also wider, with more outliers, particularly in the mid-range. The absence of momentum appears to negatively impact performance, as RMSprop is less effective at smoothing noisy gradient updates.

Similar to Blog-Nat06all, Figure 6.10 shows that both AdamW and RMSprop perform similarly to Adam when it comes to runtime.



(a) Run-time vs k for Adam, AdamW, and RMSprop



(b) L_2 vs k for Adam, AdamW, and RMSprop

Figure 6.10: The left plot compares total runtime (in seconds) for CA-GR-QC as a function of k for all optimizers. The right plot shows the average L2 error versus k illustrating the trend in parameter recovery accuracy across increasing Kronecker scales.

Summary

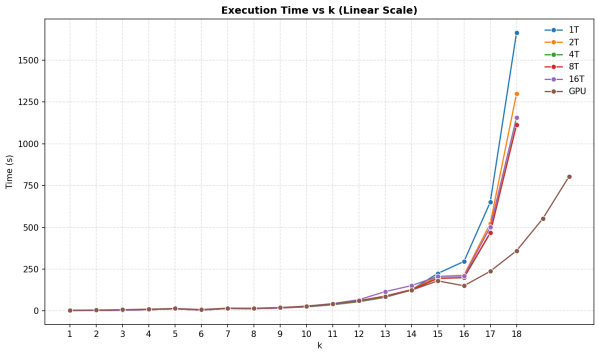
Across both datasets, AdamW performs nearly identically to Adam, with no significant differences in accuracy or runtime. This indicates that decoupled weight decay neither improves nor harms performance in this setting. In contrast, RMSprop consistently underperforms Adam, particularly on the larger dataset, where the lack of momentum results in less stable convergence. Overall, these findings support the use of Adam as the default optimizer for MFit, as its momentum component plays a key role in handling the stochastic gradients produced by MCMC sampling.

Threads Scaling

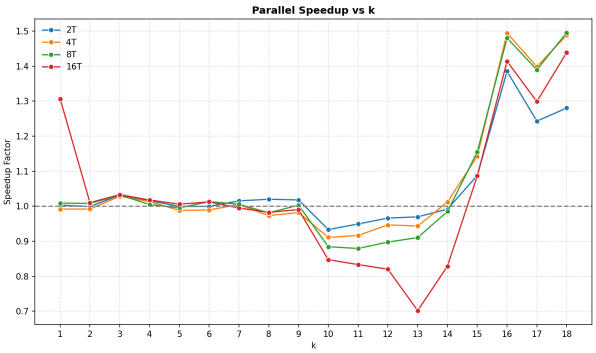
MFit consists of two primary stages, first is the MCMC-based permutation sampling phase that is fully executed on the CPU using Numba, and the second is the gradient computation along with the optimizer update, which runs on the GPU. In the absence of a GPU, the entire pipeline executes on the CPU, making the number of threads a key factor in performance. By varying the number of threads assigned to the process, we can identify the scaling limits and the point of diminishing returns for parallel CPU execution. That is why in this experiment, we analyze how runtime and accuracy vary as the CPU thread count increases from 1 to 16, while also including a GPU configuration as a baseline.

Runtime Performance

The Execution Time vs k plot in Figure 6.11 compares run-times across 1, 2, 4, 8, and 16 threads, in addition to the GPU timings. For small values of k, all configurations perform very similarly, typically completing within 10 seconds. However, as k increases, differences become more obvious. The single-thread configuration (1T) scales poorly, as you can see it shooting up at $k = 18$. In contrast, the version with GPU enabled significantly outperforms all CPU-based runs, demonstrating the advantage of GPU acceleration over CPU parallelism.



(a) Run-time for all threads + GPU vs k



(b) Speed-up for all threads

Figure 6.11: (a) Comparison of execution time for CPU thread configurations (1T to 16T) and GPU implementation on a linear scale. (b) Speedup factor for multi-threaded executions relative to the single-thread (1T) baseline, illustrating the ratio of performance gain or overhead across the parameter range.

The Parallel Speedup vs k plot in Figure 6.11 provides deeper insight into scaling behavior. Speedup is measured as a ratio of the run-time compared to the run-time of 1T setup. At smaller k, all configurations show speedup values close to 1, indicating minimal benefit from parallelism. But after ($k \approx 9$ to $k \approx 15$), something interesting happens. All the multi-threaded setups start performing worse than the single-threaded setup. It can be seen that the performance for the setup with 16T drops almost by a factor of 0.7. This could potentially be caused by expensive start-up cost. In this range, the actual work needed to be done is so small that the time required to manage multiple threads like assigning tasks, syncing memory, joining threads exceeds it. Once the k value crosses 14, the actual work that needs to be done becomes more than the setup time cost, then the multi-threaded setups are able to properly utilize their potential. One thing to notice in the speed-up plot is that the maximum speed-up is achieved by the setup with 4 and 8 threads instead of 16 threads. This limitation is consistent with Amdahl's LawAmdahl (1967), which states that the presence of sequential components bounds the maximum achievable speedup for multi-threaded or parallel systems.

Accuracy

Figure 6.12 shows that all thread configurations yield very similar L_2 error values across the entire range of k. The curves for different thread counts overlap almost perfectly, and the GPU results align closely as well at least for the higher k values. The setup with GPU enabled seems to perform better compared to pure CPU setups until $k = 6$. This phenomenon could be due to the fact that when gradient computations are offloaded to the GPU, the calculations are performed with higher numerical precision compared to CPU-only runs for smaller k. When k is small, a multi-threaded CPU setup can cause floating-point summation drift due to things like rounding and order of operations. But at higher k, the magnitude of the parameters and the gradients grow and the tiny rounding errors are no longer significant enough. This indicates that for applications focused on smaller k values, GPU execution offers advantages in both speed and accuracy. At larger k, the accuracy differences diminish, and the primary benefit of using a GPU becomes its superior runtime performance. Overall, we can say that parallelism affects execution time only, not the accuracy.

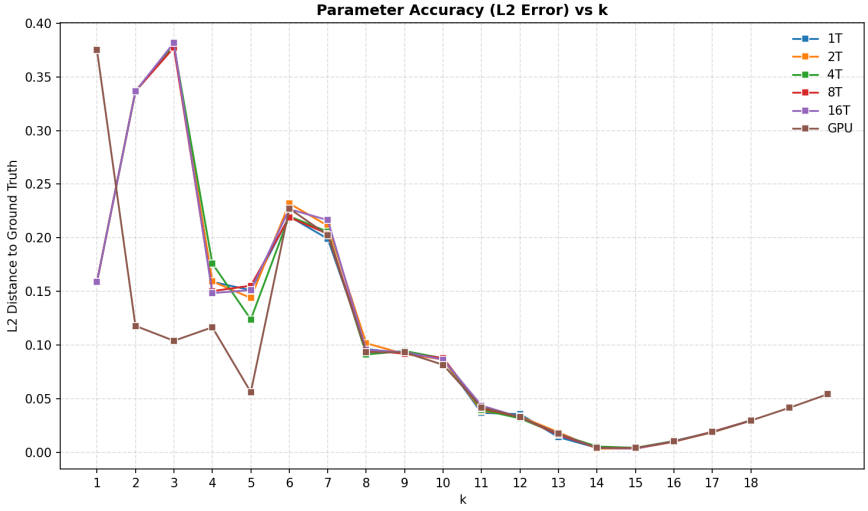


Figure 6.12: Accuracy vs k across multiple threads + GPU

Execution Time Breakdown

Figure 6.13 shows how time is distributed across different stages of the model during fitting. The plot shown is for Blog-Nat06all at $k = 18$. More such plots were generated and are attached in the Appendix but we chose the biggest one for analysis since it has the biggest changes in time for the different parts of the fitting process.

The red blocks represent time taken by MCMC sampling, the blue blocks correspond to gradient computation and optimization, and the gray segment indicates overhead which includes time taken for model initialization and transfer of data between GPU and CPU. The first thing that catches the eye is the fact that the time block for MCMC part remains nearly identical across all thread counts and the GPU setup. This indicates that the MCMC phase does not benefit from multi-threaded systems. This is expected since MCMC is purely sequential in MFit as each step of the Markov Chain depends on the previous one.

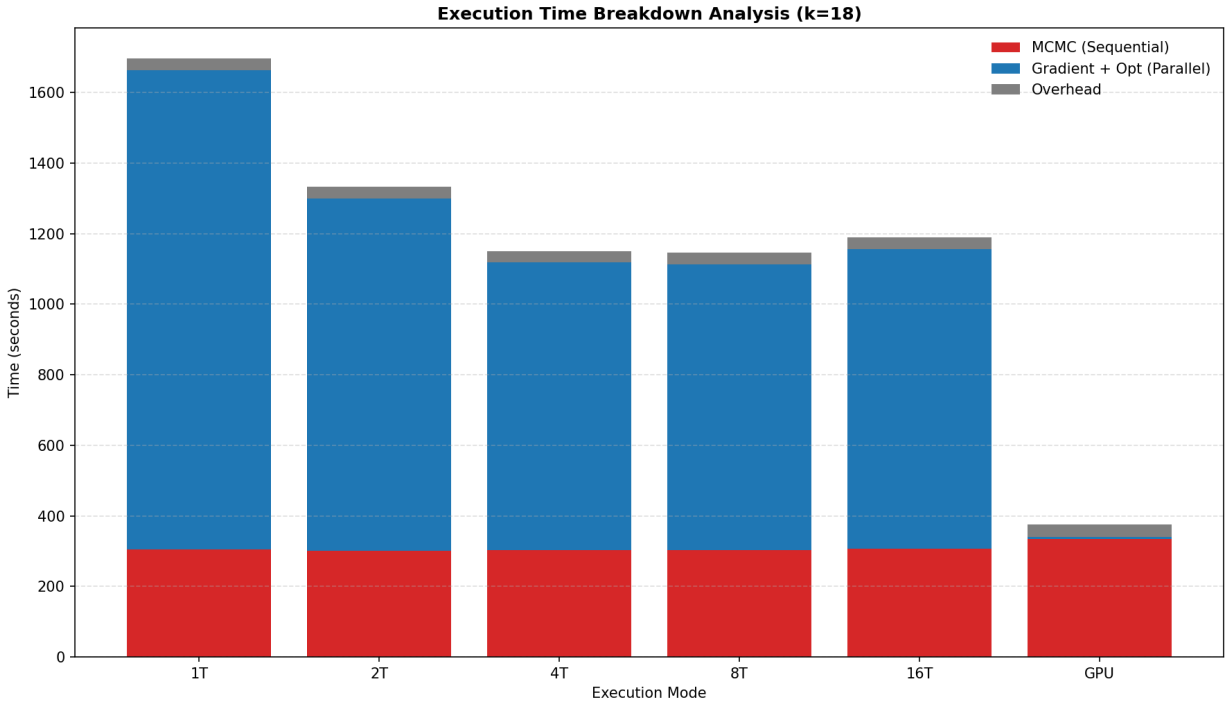


Figure 6.13: Blog-Nat06all Execution Time Breakdown

On the other hand, the gradient and the optimization stage show a decent drop in time blocks across increasing thread level. There is a good drop in the time blocks from single

threaded to double threaded and some additional improvement for 4 threaded systems. But after that there is no significant change suggesting diminishing returns. However there is a massive drop in the setup that has GPU enabled. This setup lets the GPU do all the heavy lifting of the gradient computations, which explains the minimal blue block for the GPU version to a point that it almost cannot be seen.

From this experiment, it becomes clear that the MCMC phase acts as the primary bottleneck. At $k = 18$ with 4 threads, it accounts for roughly two-thirds of the total runtime. According to Amdahl's Law Amdahl (1967), this implies a theoretical upper bound for speedup, assuming perfect parallelization of the remaining workload. This limitation cannot be overcome unless MCMC can somehow be implemented in a non-sequential manner.

Figure 6.14 shows how the total execution time is split for the single-threaded setup across different values of k. The numbers above each bar show the ratio of gradient and optimization time compared to MCMC time.For smaller values of k, this ratio is much smaller than 1. For example at $k = 11$, it's only $0.05\times$, which means MCMC takes up most of the run time and the part that can be parallelized is very small.

However, as k grows, the gradient and optimization stage expands rapidly, reaching $1.38\times$ at $k = 16$, $2.24\times$ at $k = 17$, and $4.45\times$ at $k = 18$. In other words, the parallelizable portion of the total workload grows much faster than the sequential portion as the problem scales up.

This trend aligns better with Gustafson's Law Gustafson (1988) than with Amdahl's Law. Amdahl's Law assumes the problem size is fixed and suggests that the sequential part will always limit the overall speedup. In contrast, Gustafson's Law reflects what happens here: as the problem size increases, the parallel portion of the work also increases, while the sequential portion does not grow as much.

In our case, the MCMC chain length grows very slowly with respect to k. On the other hand, the gradient computation grows with the number of edges and the size of the Kronecker expansion. Because of this, GPU acceleration becomes more effective at larger k, since it

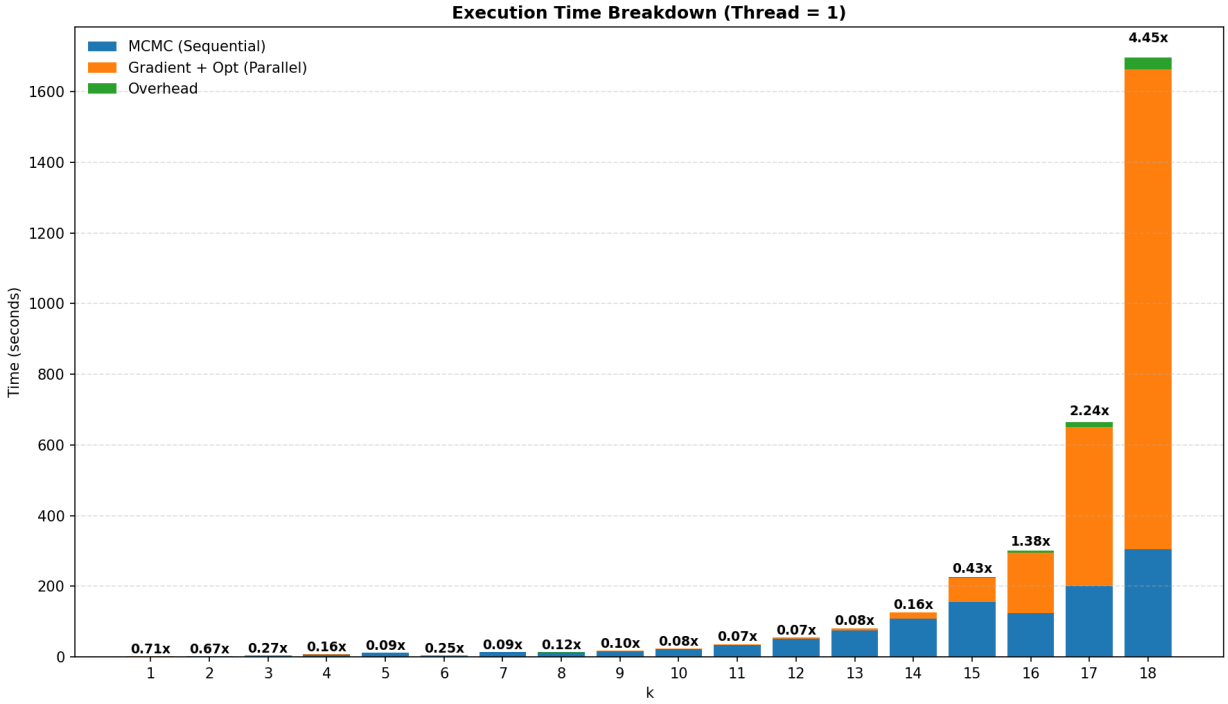


Figure 6.14: Execution time breakdown for the single-threaded configuration across all k values

speeds up the part of the computation that is growing the most.

This is also supported by Figure 6.13, where the GPU version reduces the total runtime to around 375 seconds at $k = 18$. This improvement comes mainly from speeding up the gradient and optimization stage, while the MCMC time remains almost the same across all configurations.

Summary

Thread scaling experiments showed that MFit consists of two stages with different behavior. The MCMC phase is sequential and remains a bottleneck, while the gradient and optimization phase is parallelizable and dominates runtime at larger k. As a result, GPU acceleration becomes more effective as the problem scales, consistent with Gustafson's Law. We also saw that parallelism does not affect parameter estimation accuracy. The MCMC bottleneck is the primary limitation of the current design, and parallelizing it remains an open problem.

However, recent work on parallelizing MCMC across the sequence length Zoltowski et al. (2025) represents a promising direction that could be adapted in future versions of MFit to eliminate this bottleneck entirely.

Data Type

Unlike KronFit, MFit introduces an additional design choice when running on GPUs: the numerical precision used for storing and computing tensors. PyTorch allows multiple data types, and using lower precision can reduce memory usage and potentially improve performance for memory-bound operations. However, this comes at a cost and could possibly cause instability. Since MFit relies on gradients derived from a log-likelihood objective, it is essential to understand that lowering precision can negatively affect results.

That is why to test this, we did an experiment with three different setups. We call them baseline, economy and stress. The Baseline setup is the same as what we have been doing until now. It uses 64-bit integers (int64) for graph-related indices such as edge lists and permutations, and 32-bit floating point (float32) for parameters and likelihood computations. The Economy setup changes only the index tensors from int64 to int32, reducing memory usage for graph storage by half while leaving everything else unchanged. The Stress setup adds to the Economy setup by converting all major computations, including parameters and likelihood values, to float16. This setup serves as an extreme test of reduced precision.

The float16 version is called the stress test because its range is significantly smaller, roughly from 2^{-24} to 2^{16}. During fitting, when log-probabilities become sufficiently negative, exponentiation can underflow to zero, effectively eliminating gradient contributions from those samples. It is also important to note that PyTorch internally keeps optimizer states such as momentum buffers in float32, so the reduced precision primarily affects forward passes and gradient calculations rather than the optimizer update itself. To ensure that the MCMC remains stable, we use float64 across all experiments which runs only on CPU using

numba. We separated the precision changes for GPU and CPU so they don't affect each other.

Table 6.1 summarizes the three experiments we will perform.

Component	Variable	Baseline	Economy	Stress
Graph indices	`idx_dtype`	`int64`	`int32`	`int32`
Math precision	`math_dtype`	`float32`	`float32`	`float16`
Parameter bound	`upper_bound`	0.9999	0.9999	0.999
MCMC (CPU)	—	`float64`	`float64`	`float64`

Table 6.1: Precision setups for the three dtype experiments.

Blog-Nat06all

The Blog-Nat06all dataset has been the one on which MFit and KronFit both had performed very well. The same thing can be seen here in Figure 6.15. The distributions and the median line for the Economy setup follow very closely to Baseline indicating that the reduction in storage of the indices from int64 to int32 did not affect the accuracy. The Economy version fully overlaps Baseline model results further validating our results.

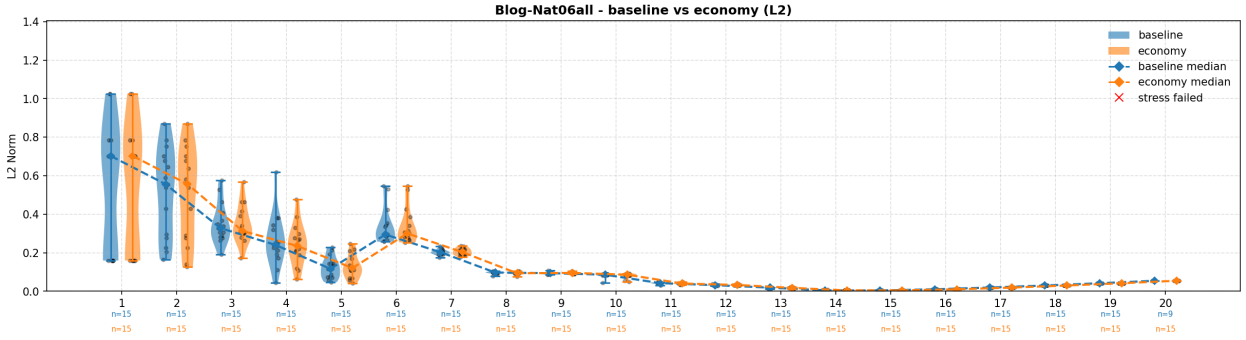


Figure 6.15: Shows the distribution of L2 error for Blog-Nat06all across 30 runs for Baseline vs Economy at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

However the accuracy results for the stress configurations are very concerning. At $k = 11$, it started failing and returned infinity for its log-likelihood. That is why the red crosses are plotted at the k values where the stress experiment failed. Once the experiment failed, there

was no point in running it for higher k values as it would do the same thing. That is why we stopped the experiments for the stress setup on Blog-Nat06all as shown in Figure 6.17.

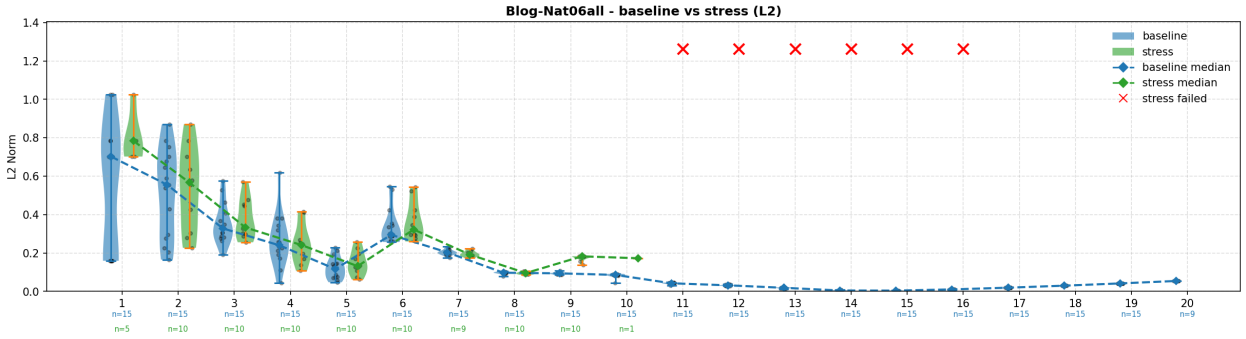
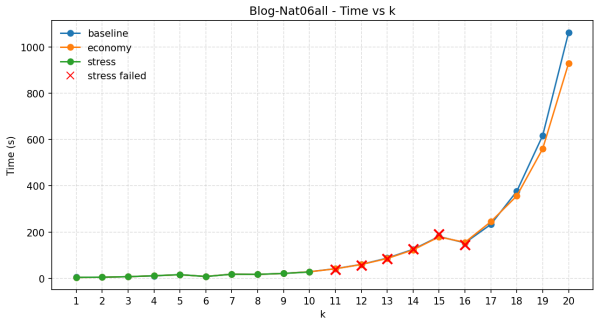
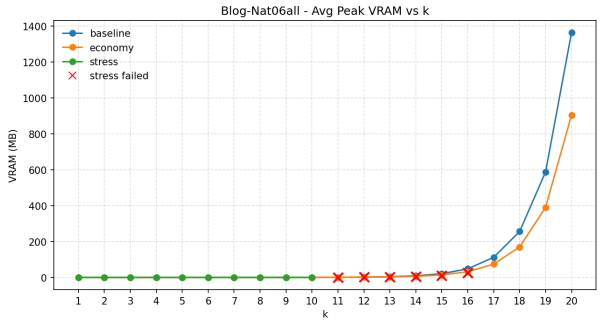


Figure 6.16: Shows the distribution of L2 error for Blog-Nat06all across 30 runs for Baseline vs Stress at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.



(a) Run-time vs k for Baseline and Economy



(b) Avg Peak VRAM vs k for Baseline and Economy

Figure 6.17: The left plot compares total runtime (in seconds) for Blog-Nat06all as a function of k for all 3 data type setups. The right plot shows the VRAM vs k for the different configurations.

When it comes to memory usage, Economy setup significantly reduces VRAM requirements. As shown in the right plot in Figure 6.17, Economy requires roughly 35 percent less VRAM as compared to Baseline. This percentage will increase even more if the k value increases. The stress setup should in theory use even less memory but since it started failing, we stopped further experiments on it, thus there is no data available for its VRAM requirements.

CA-GR-QC

Similar to Blog-Nat06all, Baseline and Economy again produce almost perfectly identical results across all values of k. The L_2 error decreases steadily until around $k = 9$ or 10, where the model best matches the graph size, and then begins to increase for larger k values as we saw happen in earlier experiments for this dataset.

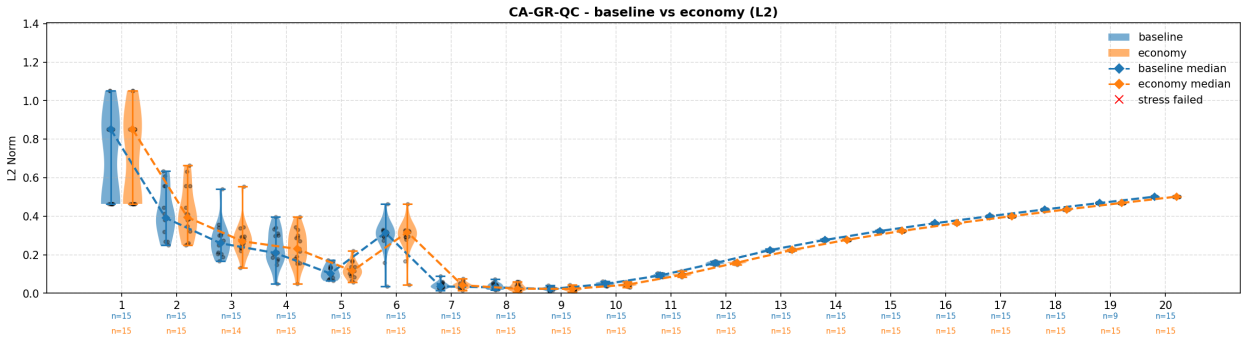


Figure 6.18: Shows the distribution of L2 error for CA-GR-QC across 30 runs for Baseline vs Economy at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

The Stress setup fails even earlier here, beginning at $k = 9$ as shown in Figure 6.19. This earlier breakdown is consistent since the initial parameters for CA-GR-QC are much smaller than Blog-Nat06all which results in worsening the numerical stability.

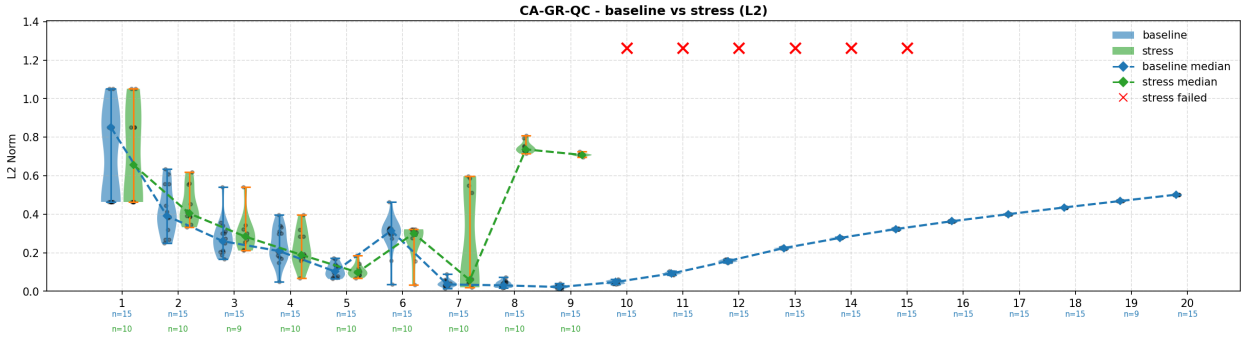
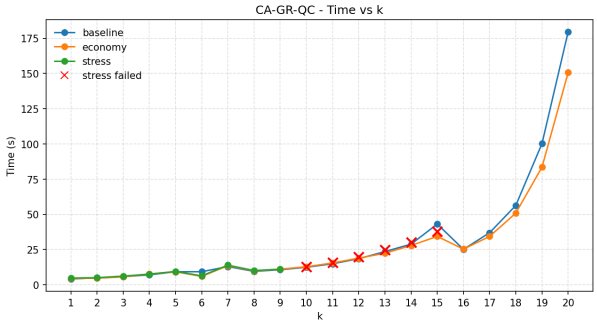


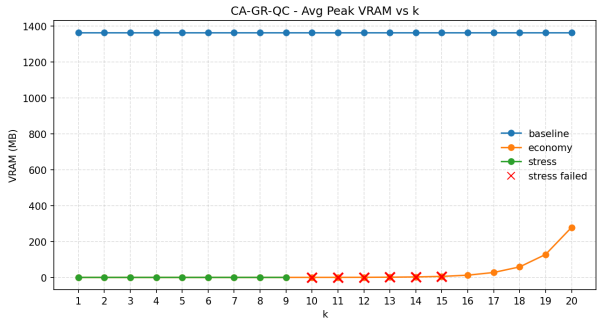
Figure 6.19: Shows the distribution of L2 error for CA-GR-QC across 30 runs for Baseline vs Stress at each value of k, visualized using violin plots. The color-coded numbers beneath the k-axis indicate the number of samples available for each value of k.

The VRAM behavior in this dataset is particularly notable especially for Baseline as shown in Figure 6.20. The Peak VRAM values for the Economy setup act similarly as the

previous dataset. They start small and slowly increase towards the end. But the Baseline configuration shows a constant memory usage across all k, suggesting that large index tensors dominate memory allocation from the beginning. This demonstrates that reducing index precision can have substantial benefits even at small problem sizes when graph storage dominates memory usage.



(a) Run-time vs k for Baseline and Economy



(b) Avg Peak VRAM vs k for Baseline and Economy

Figure 6.20: The left plot compares total runtime (in seconds) for CA-GR-QC as a function of k for all 3 data type setups. The right plot shows the VRAM vs k for the different configurations.

Summary

From these experiments we can safely conclude that switching index tensors from int64 to int32 provides consistent memory savings without affecting accuracy. Also the stress experiment shows us that reducing computation precision to float16 is only viable for small problem sizes; beyond a certain point, numerical underflow causes gradients to disappear and prevents convergence. We also saw that the maximum useful value of k is determined by the size of the graph rather than numerical precision, as even high-precision configurations degrade when the model becomes overly large relative to the data. This conclusion was further verified using the plots in the Appendix.

Summary

This chapter presented a comprehensive evaluation of MFit across four experimental dimensions.

First, the MFit vs KronFit comparison confirmed that MFit consistently outperforms KronFit in both runtime and parameter recovery accuracy across all tested datasets, with the performance gap widening at higher k values as the GPU acceleration becomes increasingly dominant over sequential CPU execution.

We also observe a consistent accuracy spike at $k = 6$ across all datasets. This is not due to a problem with the fitting methods, but instead comes from a structural property of the R-MAT generator, which we refer to as the Dead Zone.

Third, the optimizer comparison shows that AdamW performs almost the same as Adam, while RMSprop performs worse because it lacks momentum. This supports using Adam as the default optimizer for the noisy gradients produced by MCMC.

Fourth, the thread scaling experiments show that the two stages of MFit behave differently: the MCMC phase is sequential and does not benefit from more threads. On the other hand the gradient stage is parallelizable, which grows with k, and benefits strongly from GPU acceleration. This behavior is consistent with Gustafson's Law, where the parallel portion of the workload increases with problem size.

Finally, the data type experiments established that reducing index precision from `int64` to `int32` delivers consistent VRAM savings with no accuracy cost, while `float16` arithmetic introduces numerical instability beyond a model-dependent threshold due to log-probability underflow. Taken together, these results position MFit as a faster, more robust, and hardware-efficient alternative to KronFit, with well-characterized tradeoffs that inform practical deployment decisions.

Chapter 7

Conclusion

This thesis introduced MFit, a GPU-accelerated framework for fitting Kronecker graph models. It improves on KronFit by reformulating the problem as a differentiable optimization task that takes advantage of modern hardware. The main idea is to split the work between CPU and GPU: the Metropolis–Hastings MCMC permutation sampling runs on the CPU using Numba, while the log-likelihood and gradient computation run on the GPU using PyTorch's autograd. This keeps the same mathematical foundation as KronFit but replaces its slow, sequential gradient computation with a faster, parallel approach.

Empirical evaluation across 20 real-world graph datasets demonstrated that MFit consistently outperforms KronFit in both runtime and parameter recovery accuracy. The speedup becomes larger at higher k, reaching over $10\times$ in some cases, since GPU acceleration is more effective on larger problems. Additional experiments show that Adam is the best optimizer for handling the noisy gradients from MCMC. Thread scaling experiments showed that the MCMC step remains a sequential bottleneck as explained by Gustafson's Law, and that using `int32` reduces memory usage without affecting accuracy, while `float16` can cause numerical issues. We also observed a consistent error spike at $k = 6$, which was traced to a structural property of the R-MAT generator, called the Dead Zone.

Beyond Kronecker graphs, this thesis introduced GSDLFit, which extends MFit into a more general framework using a wrapper interface. This allows the same fitting pipeline to work with any model that can be expressed in the Graph Structure Descriptor Language.

As a result, MFit is not just an improved version of KronFit, but also a starting point for a more general and flexible graph fitting framework, establishing the groundwork for the broader GSDL project.

Future Work

There are several directions that remain open for future investigation.

Parallelizing MCMC. The Metropolis–Hastings MCMC step is currently sequential and remains the main bottleneck in MFit. It cannot be improved simply by adding more threads or GPUs. Recent work on parallel MCMC across the sequence length Zoltowski et al. (2025) suggests a promising direction that could remove this bottleneck and improve overall scalability.

Lazy evaluation in GSDLFit. The current GSDLFit implementation builds the full $N \times N$ probability matrix using `np.kron` at every MCMC step, which is not practical for large graphs. A better approach would be to compute edge probabilities $P_\sigma(u, v)$ only when needed, directly from the parameters, without constructing the full matrix. This would make GSDLFit scale to larger graphs.

Full autograd for general GSDL models. GSDLFit currently propagates gradients only through the first four Kronecker parameters due to its reliance on the bit-decomposition trick. Models with more parameters, such as Kron+Kron with eight parameters, are partially fit through MCMC alone. Expressing the Union log-likelihood as a fully differentiable PyTorch computation would unlock gradient-based optimization for the complete parameter vector of any GSDL model.

Larger initiator matrices. Both MFit and GSDLFit currently use a 2×2 initiator matrix. Allowing larger initiators could help capture more complex graph structures, but would also

increase the number of parameters and the complexity of computation.

Chapter 8

Appendix

Synthetic datasets

Answers

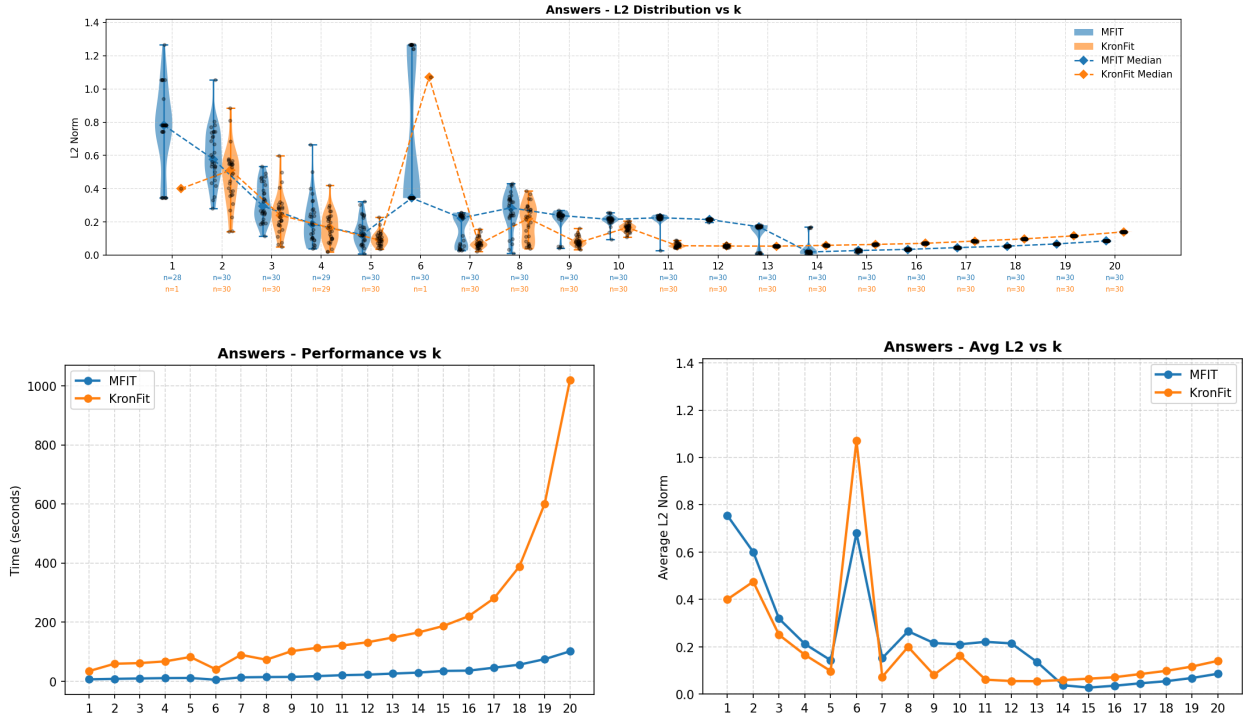


Figure 8.1: Plots for Answers dataset

AS-Newman

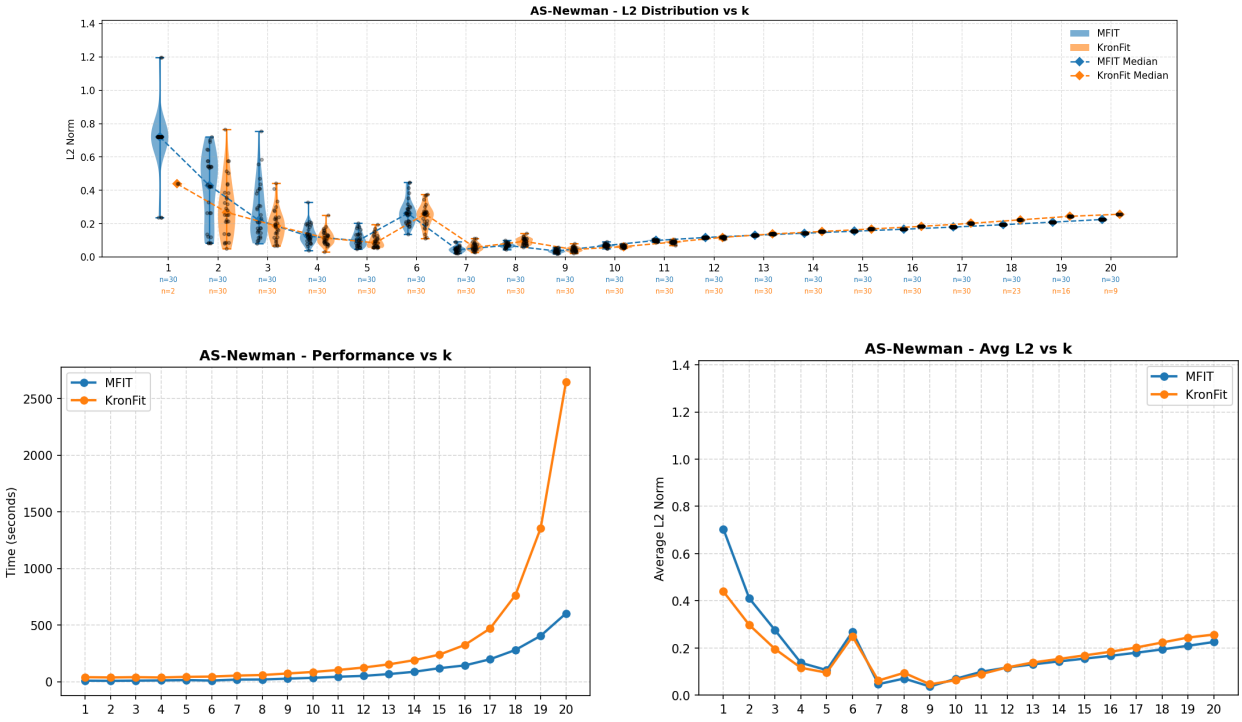
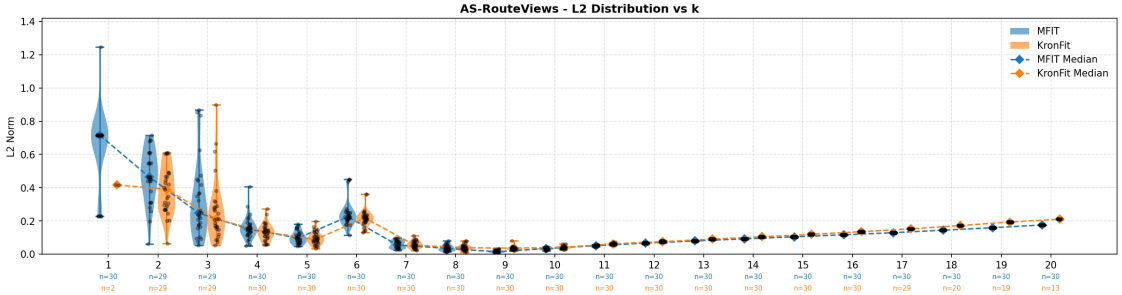


Figure 8.2: Plots for AS-Newman dataset

AS-RouteViews



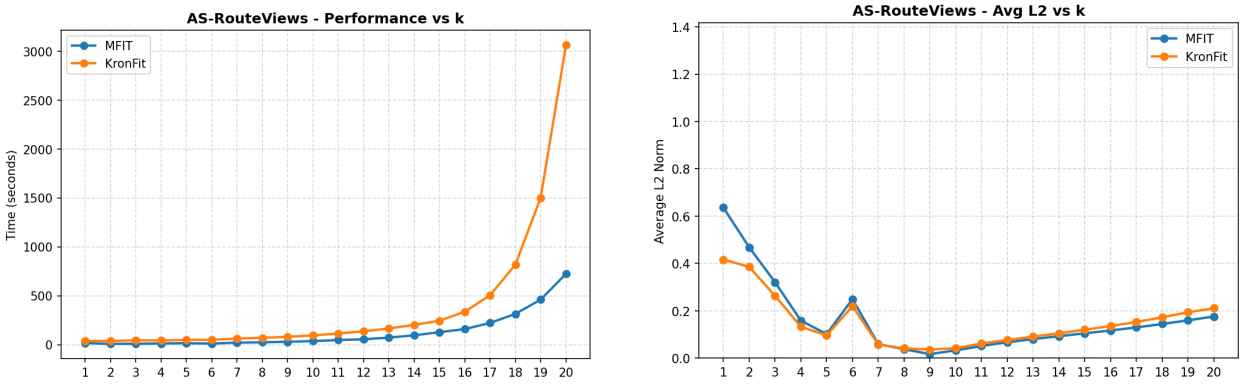


Figure 8.3: Plots for AS-RouteViews dataset

ATP-GR-QC

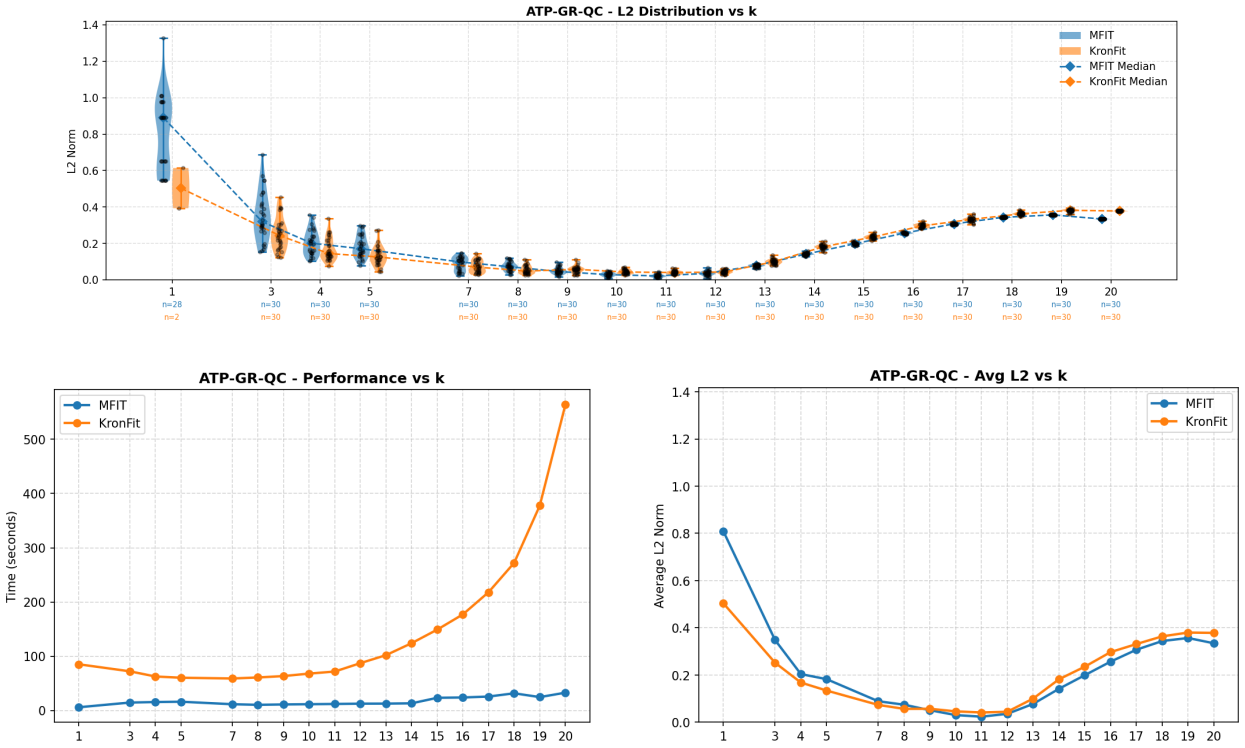


Figure 8.4: Plots for ATP-GR-QC dataset

Bio-Proteins

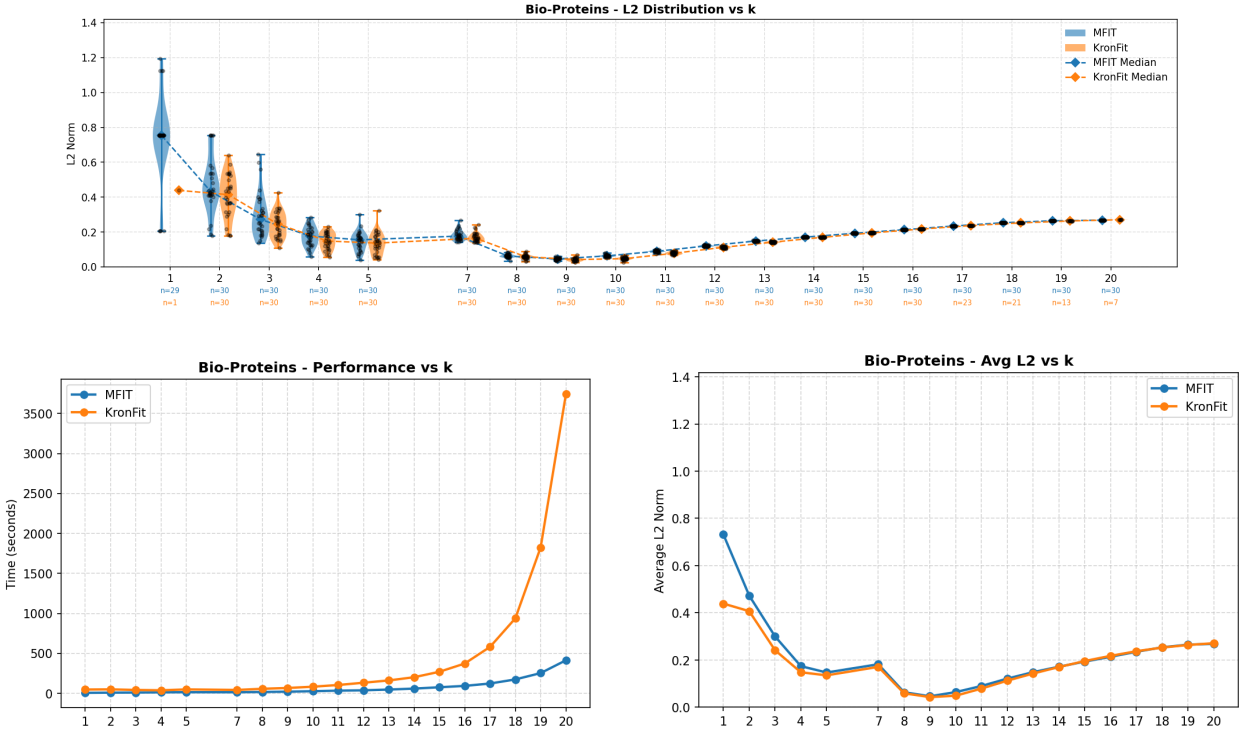
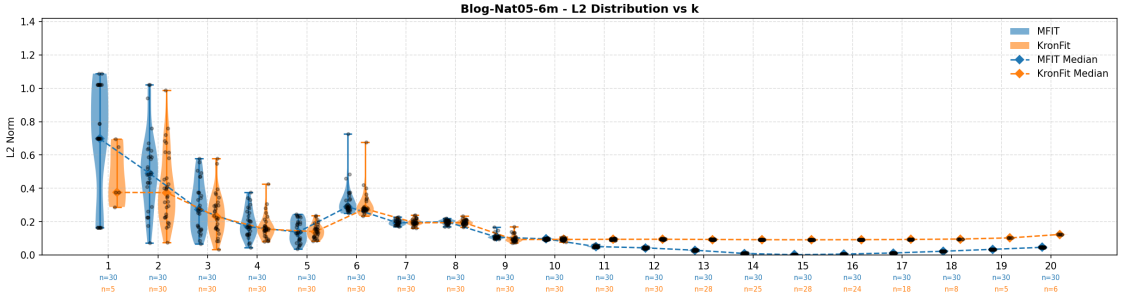


Figure 8.5: Plots for Bio-Proteins dataset

Blog-Nat05-6m



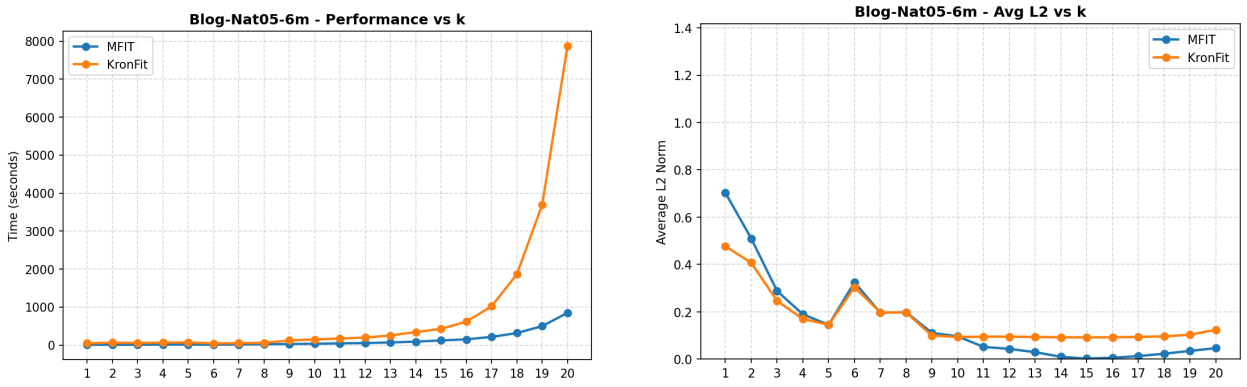


Figure 8.6: Plots for Blog-Nat05-6m dataset

Blog-Nat06all

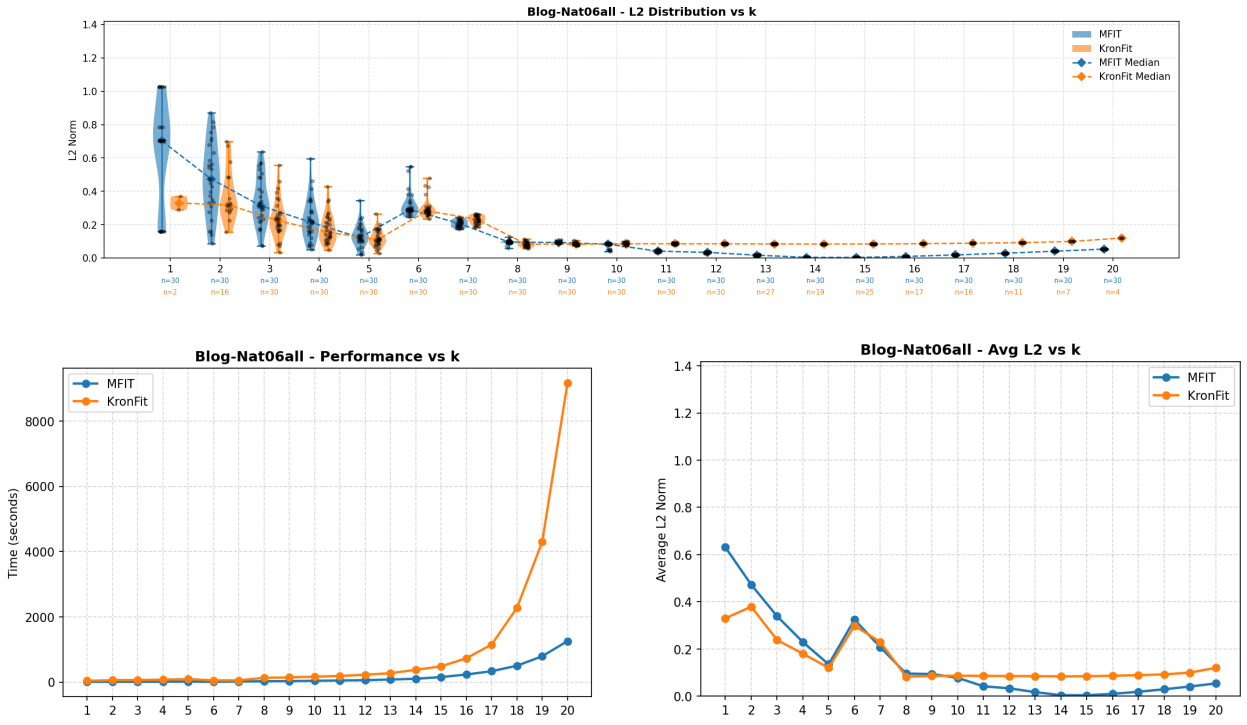


Figure 8.7: Plots for Blog-Nat06all dataset

CA-DBLP

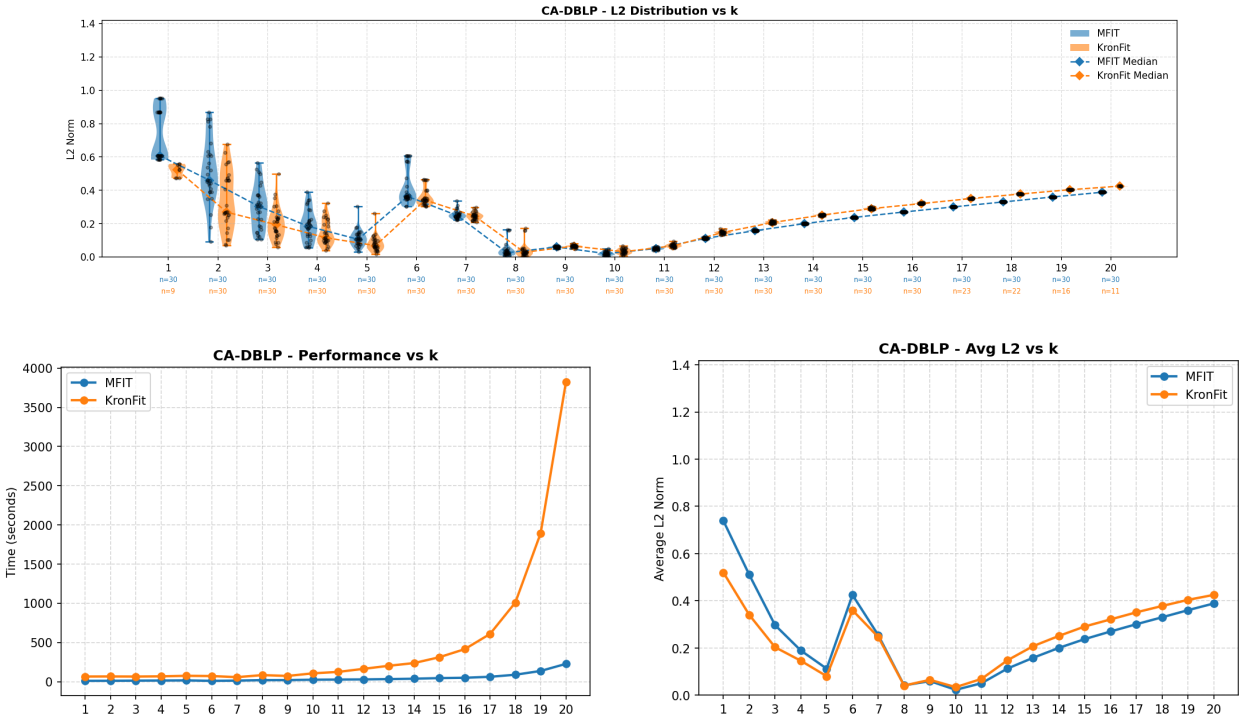
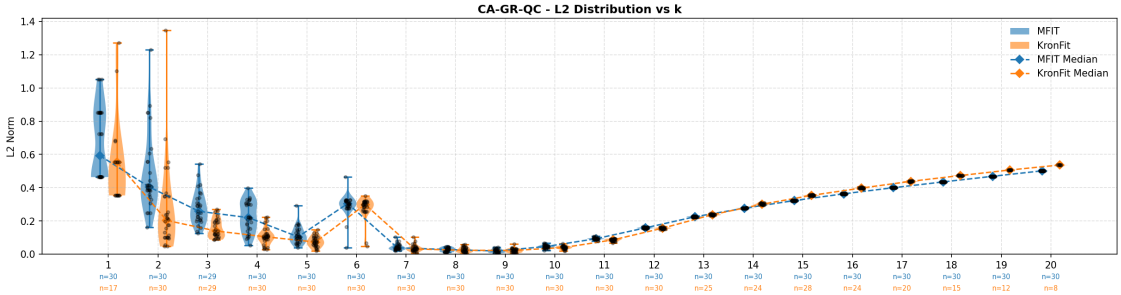


Figure 8.8: Plots for CA-DBLP dataset

CA-GR-QC



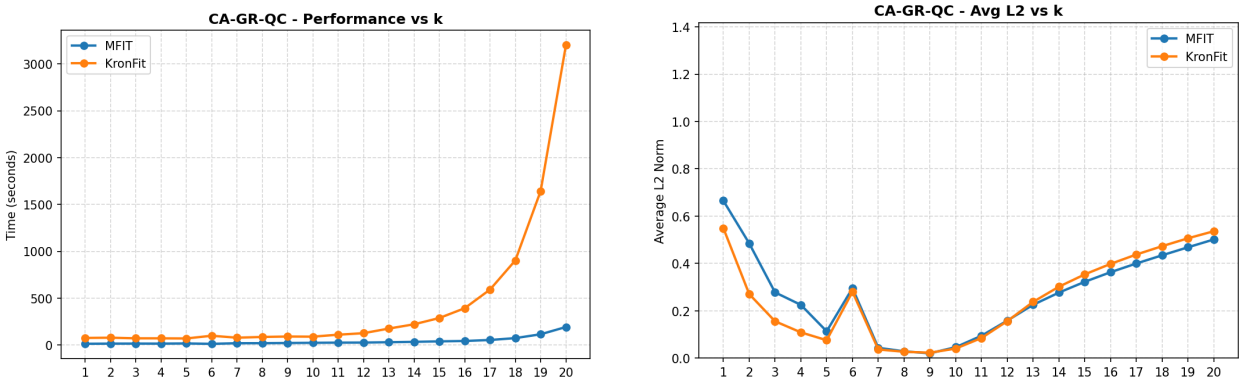


Figure 8.9: Plots for CA-GR-QC dataset

CA-Hep-Ph

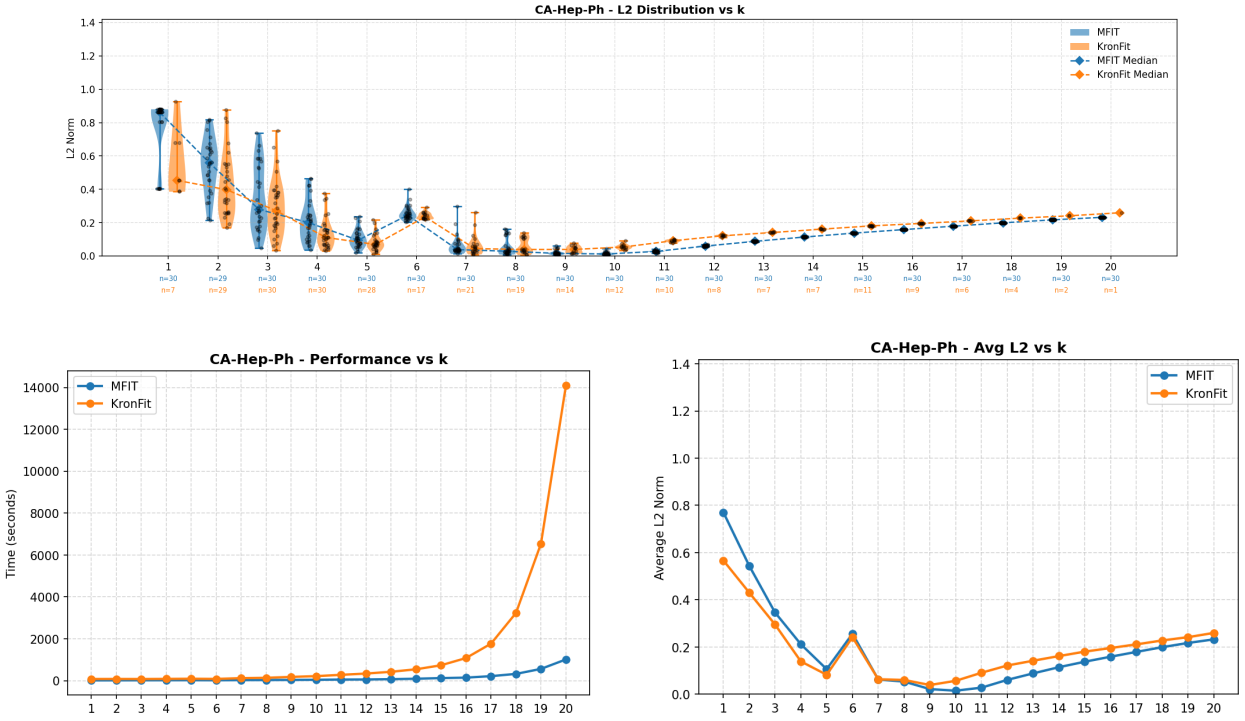


Figure 8.10: Plots for CA-Hep-Ph dataset

CA-Hep-Th

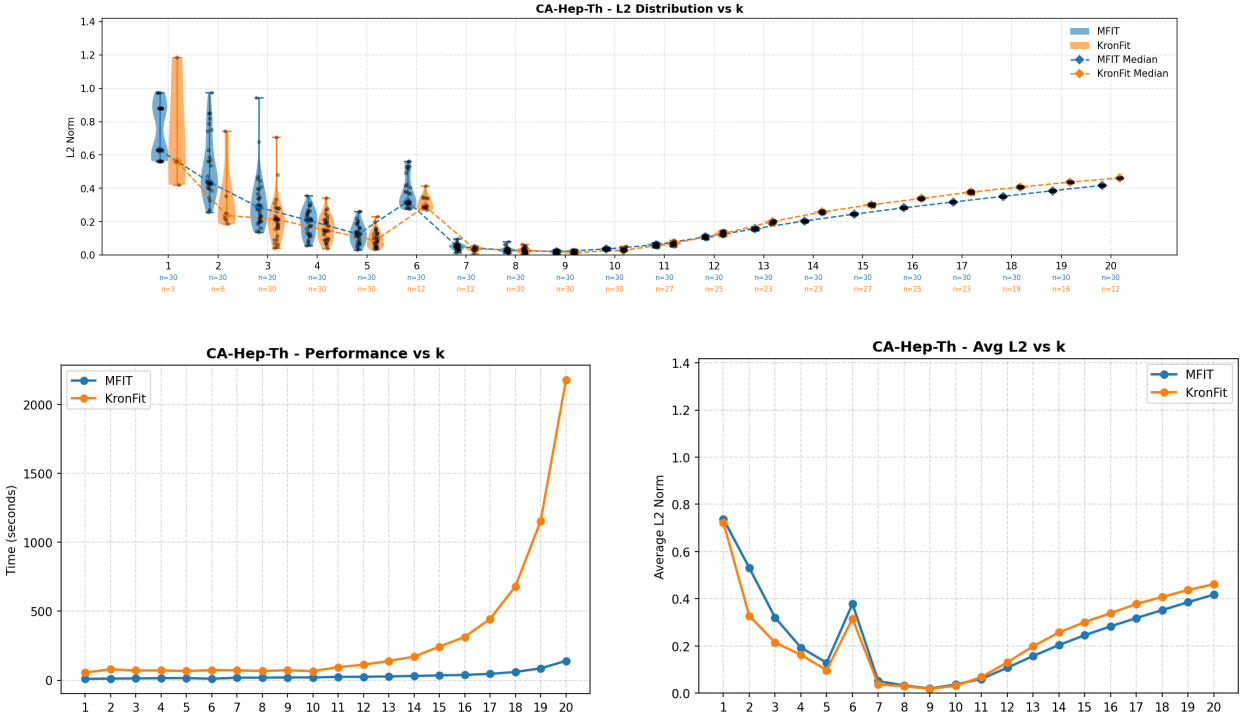
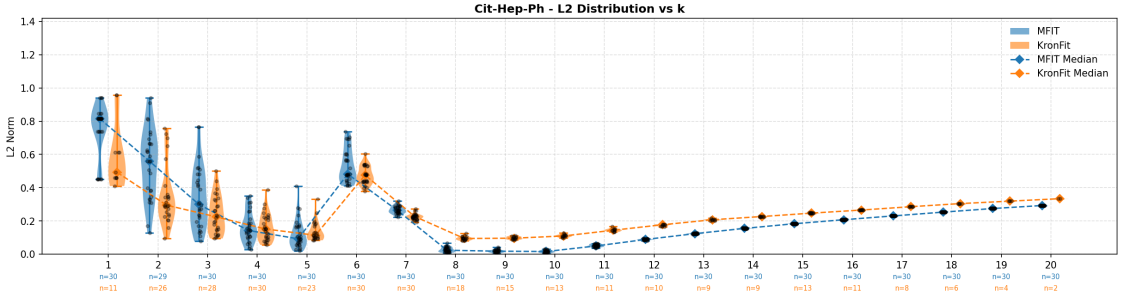


Figure 8.11: Plots for CA-Hep-Th dataset

Cit-Hep-Ph



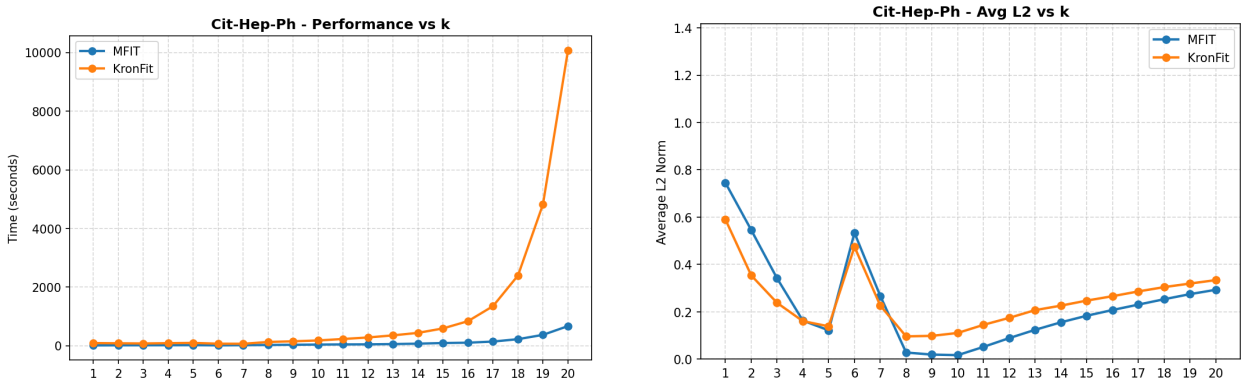


Figure 8.12: Plots for Cit-Hep-Ph dataset

Cit-Hep-Th

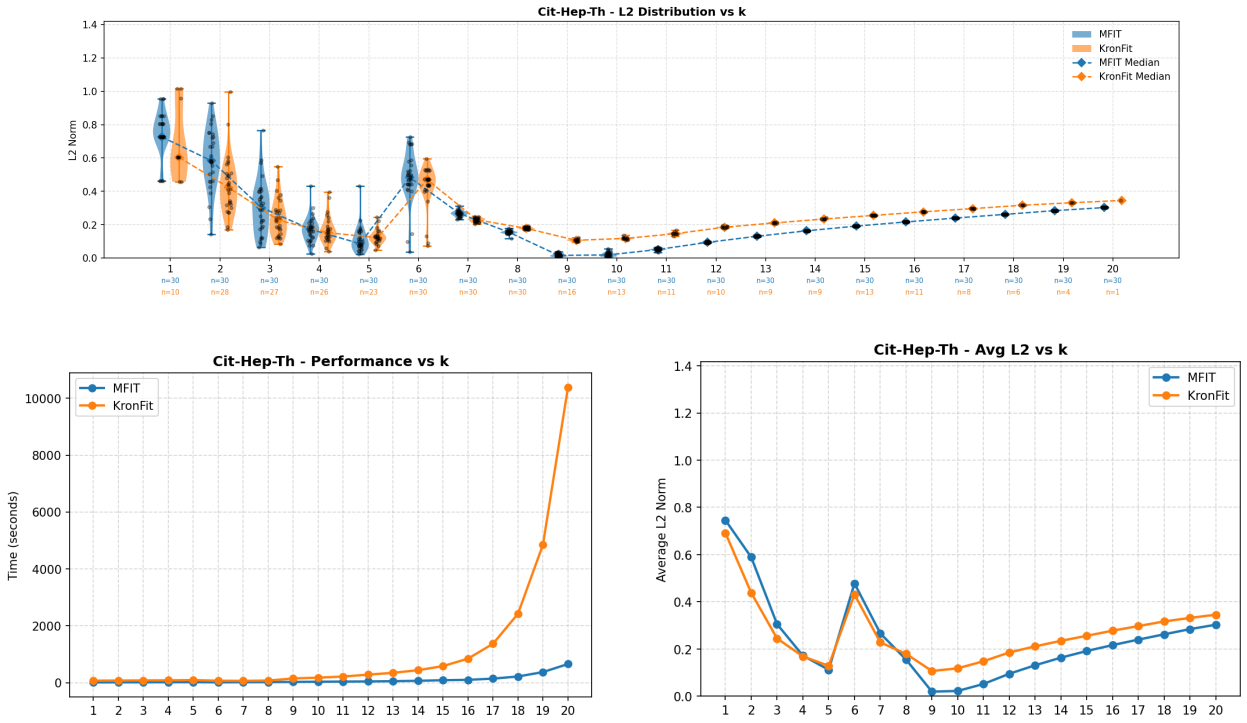


Figure 8.13: Plots for Cit-Hep-Th dataset

Delicious

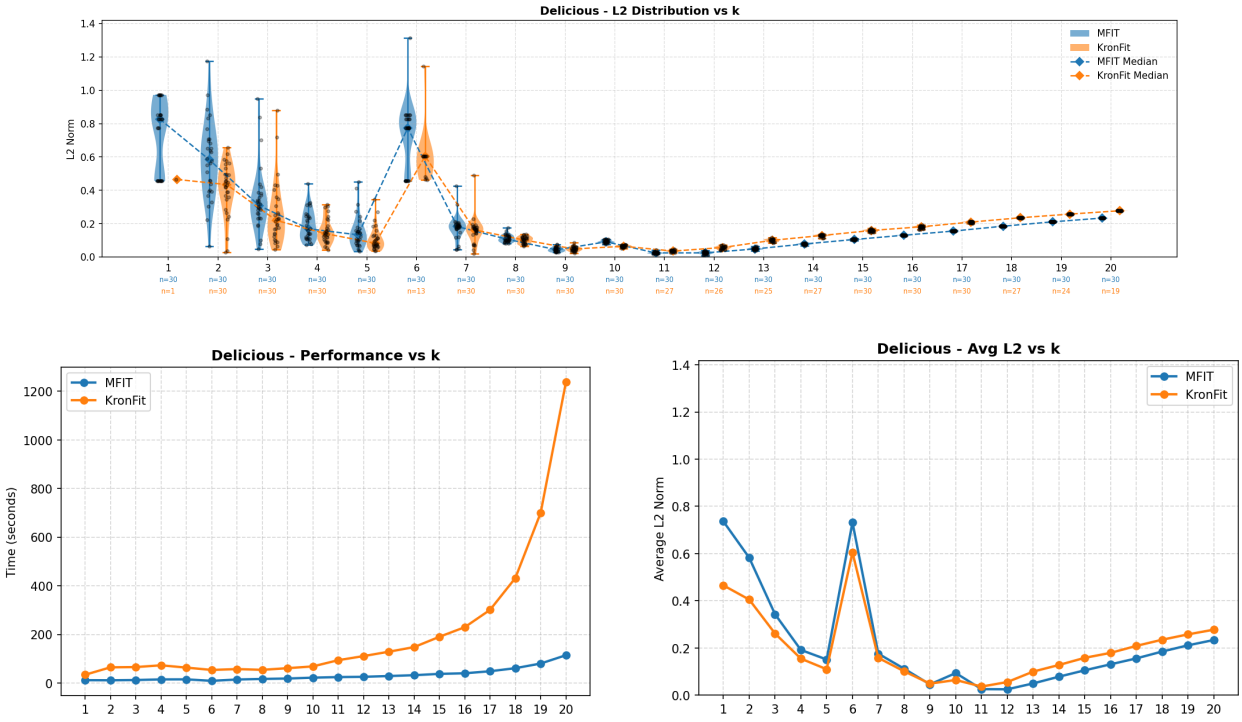
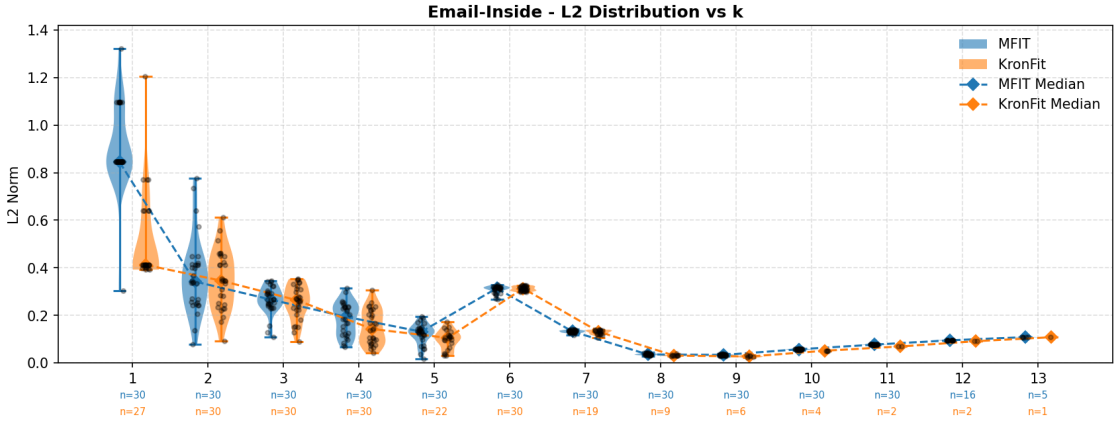


Figure 8.14: Plots for Delicious dataset

Email-Inside



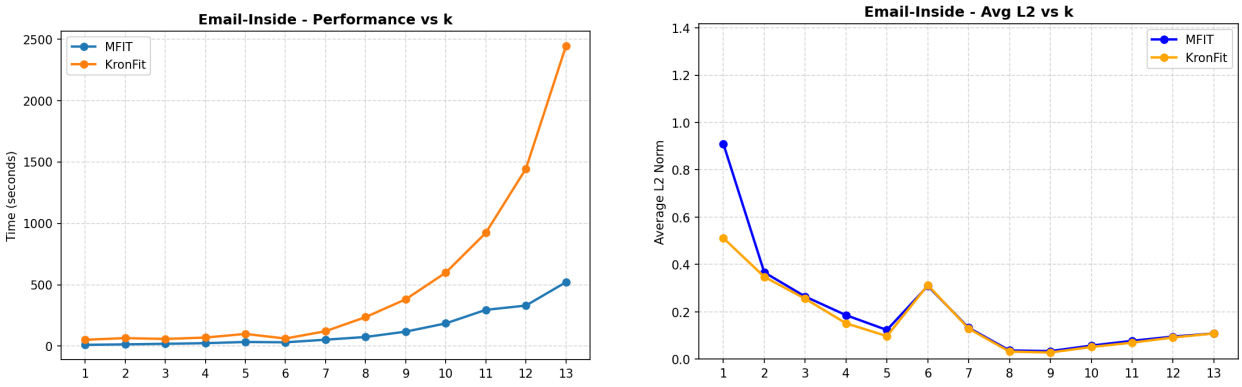


Figure 8.15: Plots for Email-Inside dataset

Epinions

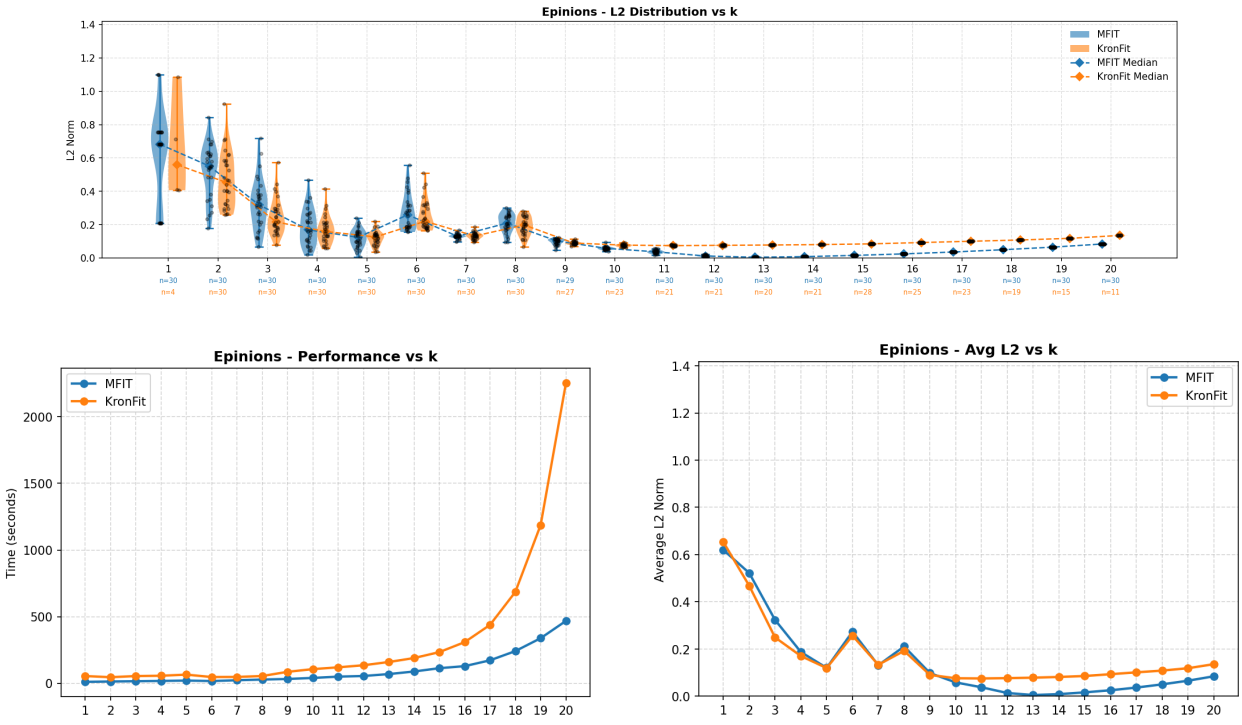


Figure 8.16: Plots for Epinions dataset

Flickr

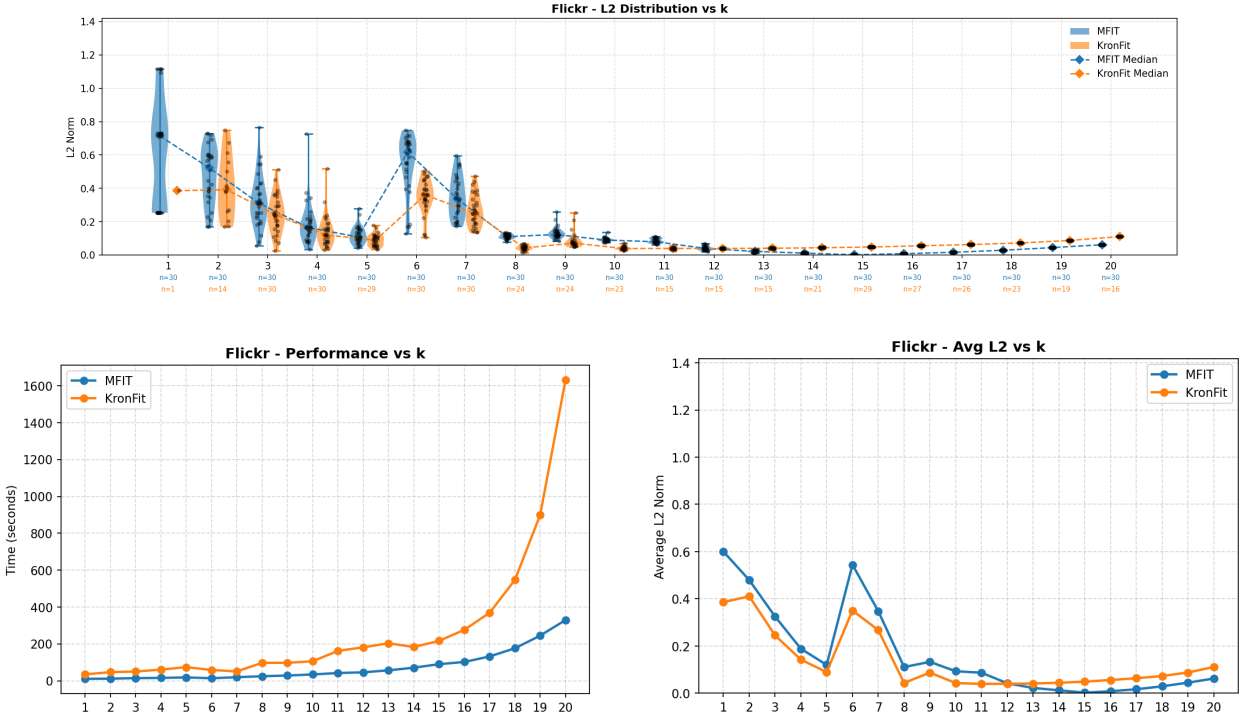
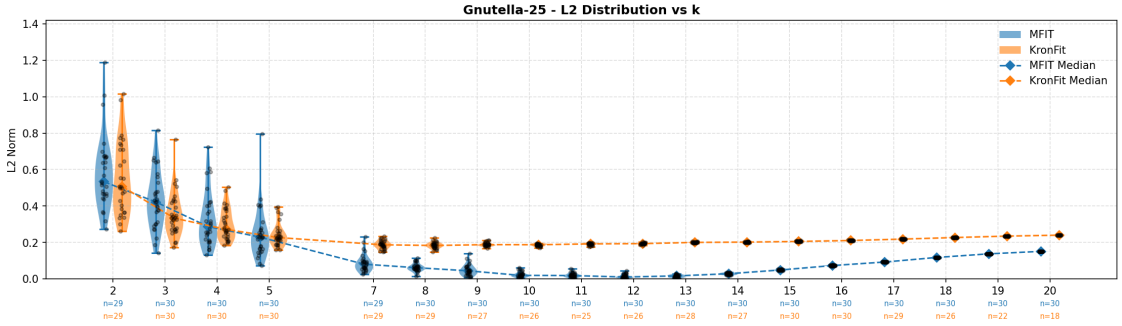


Figure 8.17: Plots for Flickr dataset

Gnutella-25



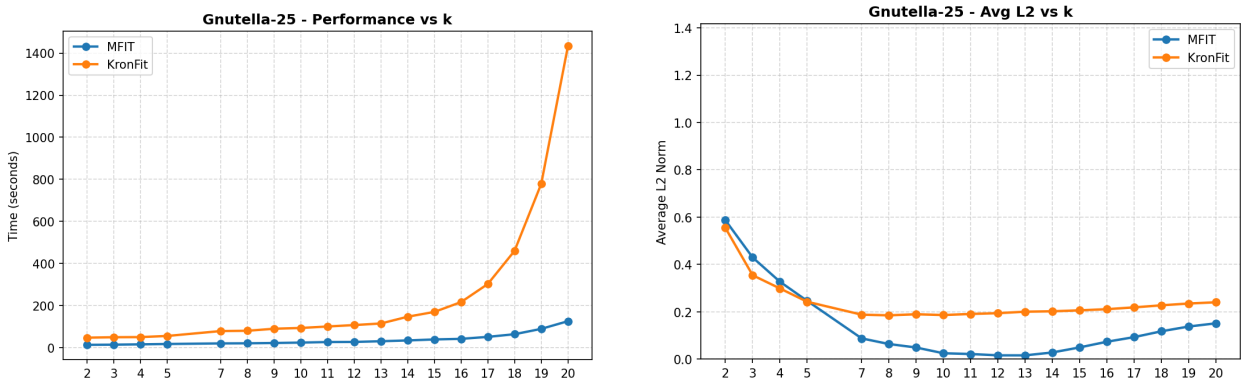


Figure 8.18: Plots for Gnutella-25 dataset

Gnutella-30

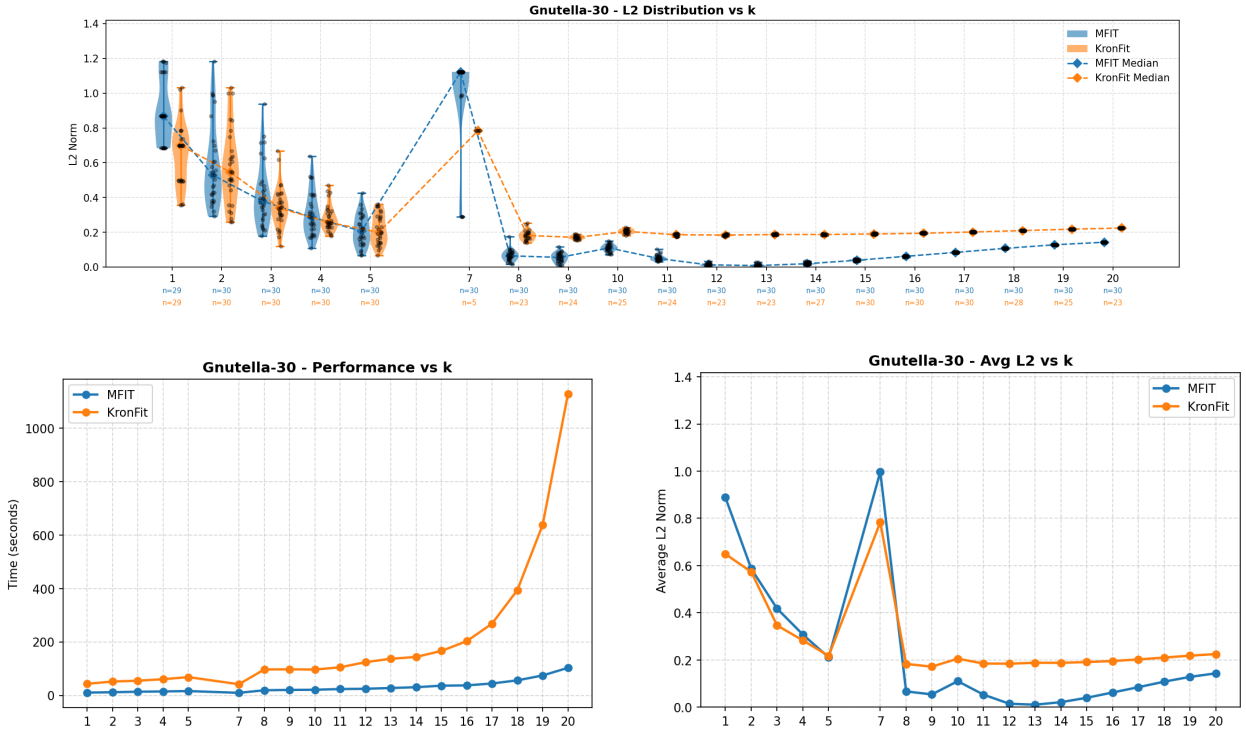


Figure 8.19: Plots for Gnutella-30 dataset

Web-NotreDame

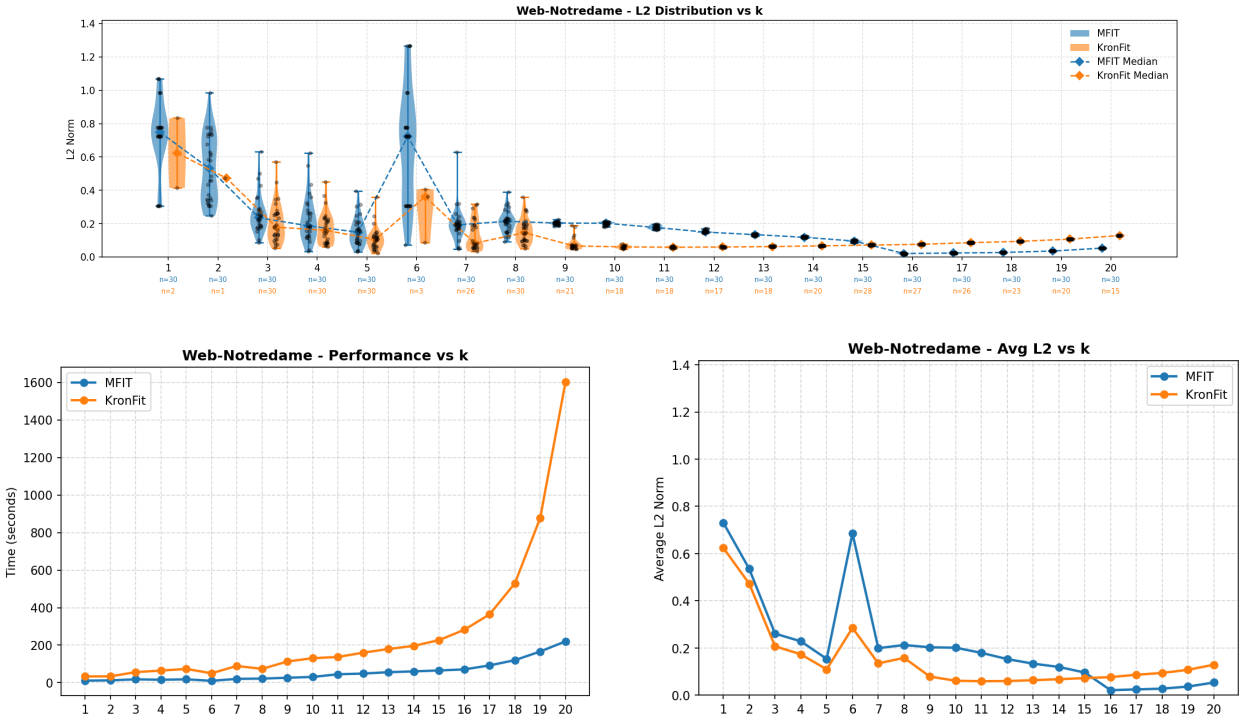


Figure 8.20: Plots for Web-NotreDame dataset

Blog-Nat06all Execution Time Breakdown

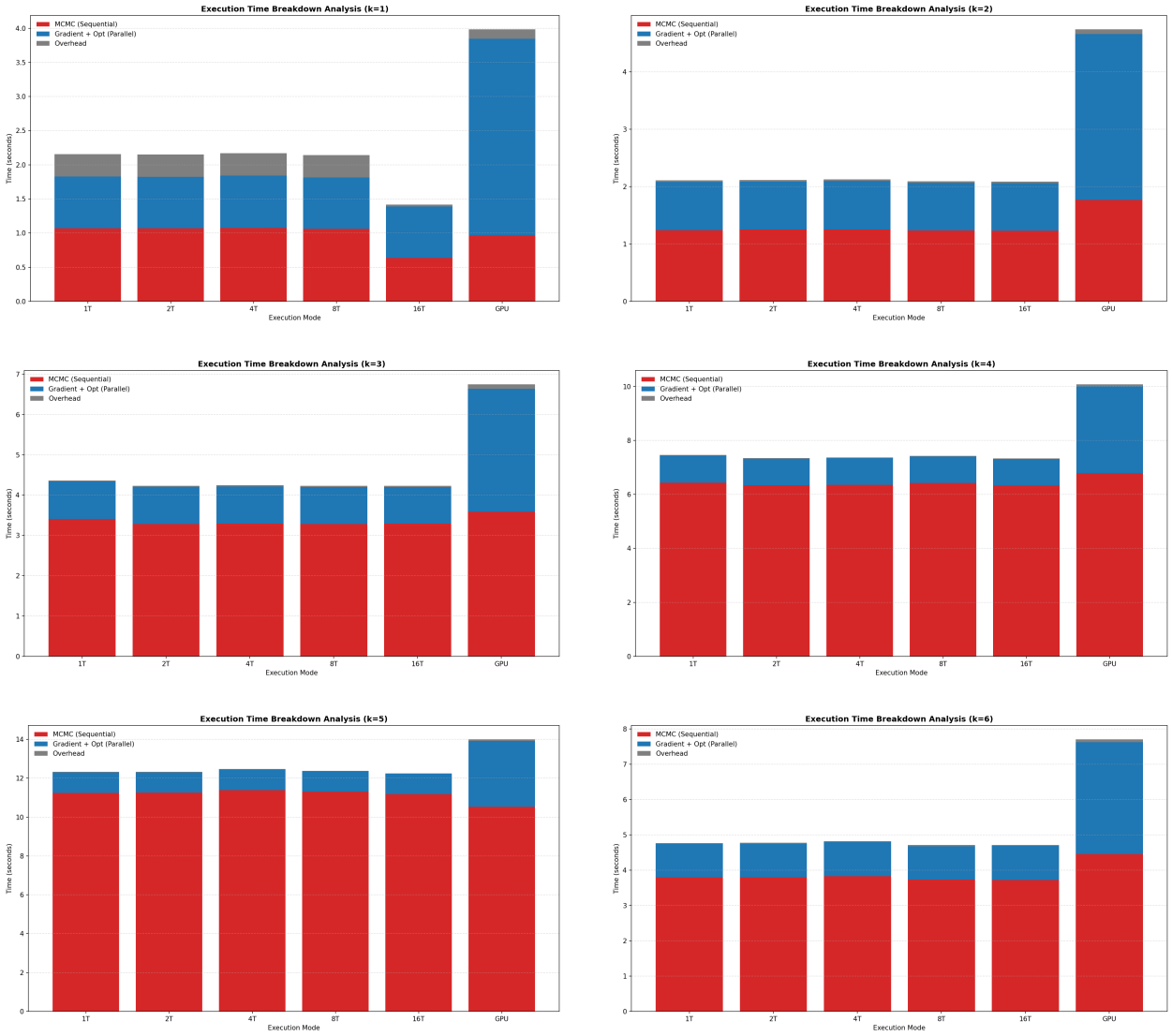


Figure 8.21: Breakdown plots for $k = 1$ to $k = 6$

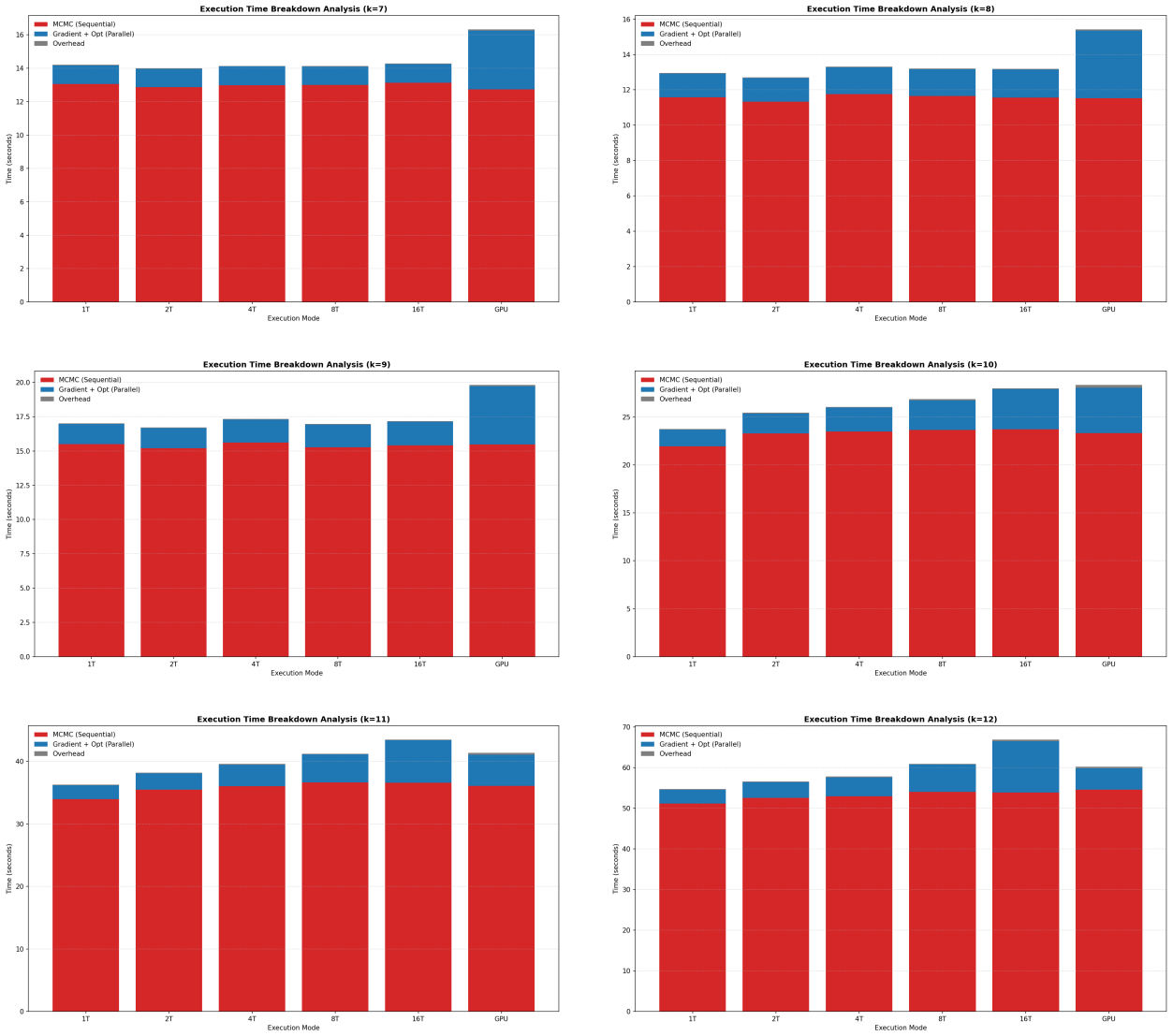


Figure 8.22: Breakdown plots for $k = 7$ to $k = 12$

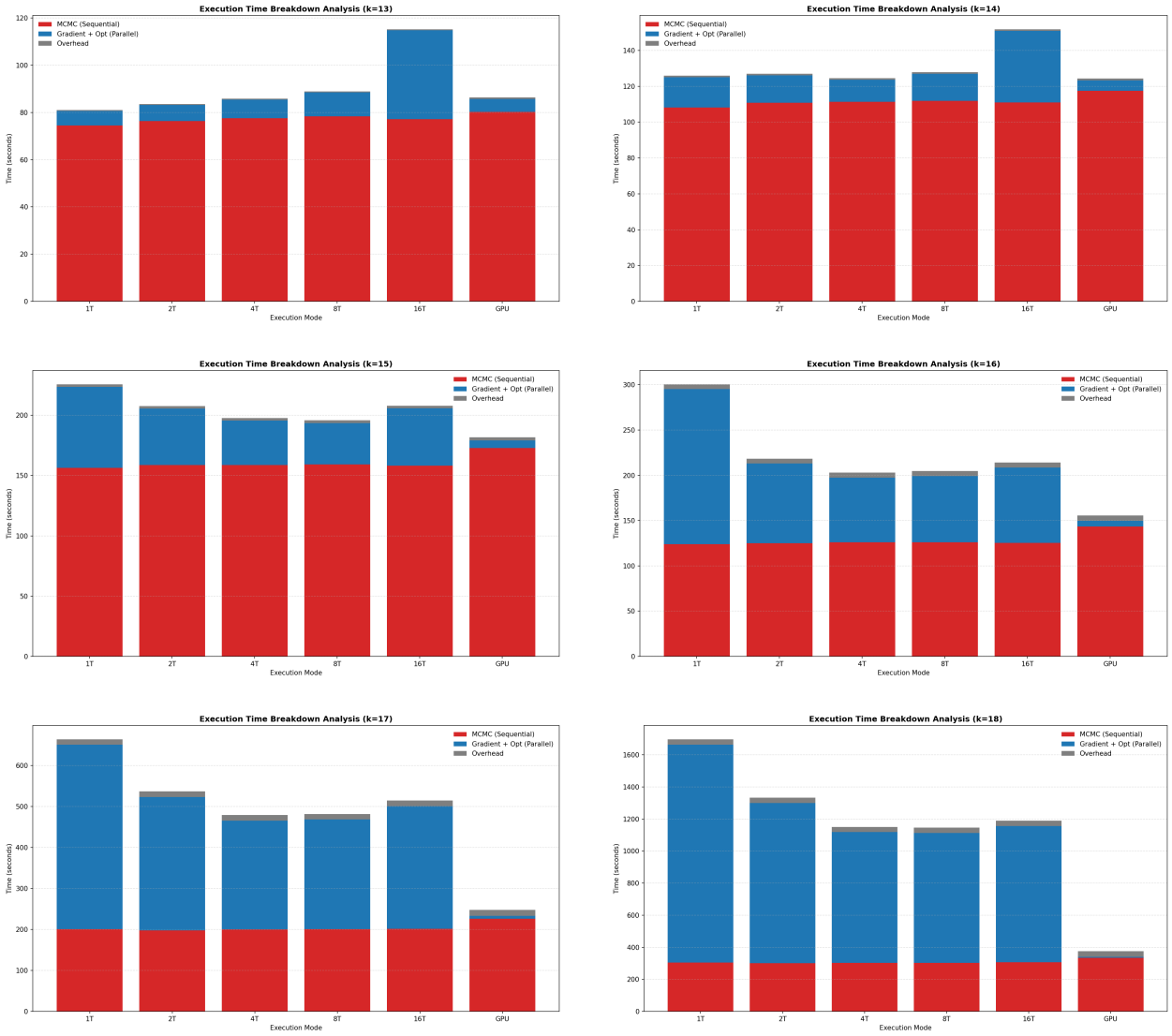


Figure 8.23: Breakdown plots for $k = 13$ to $k = 18$

Spy Plots

AS-Route Views

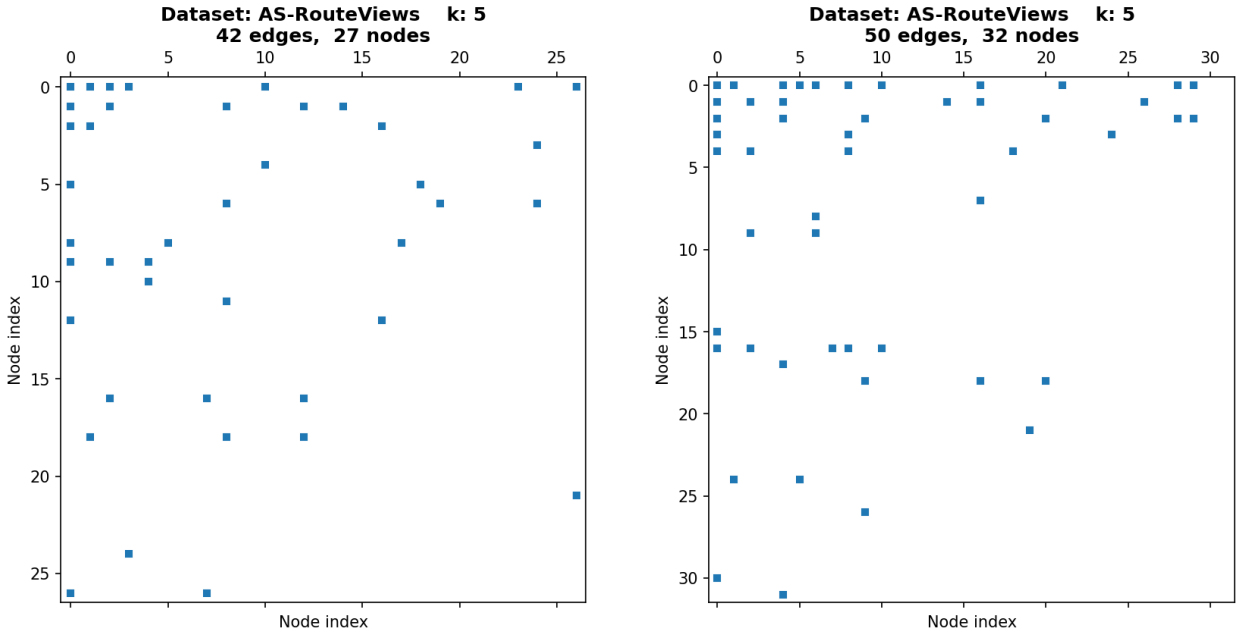


Figure 8.24: Spy Plots

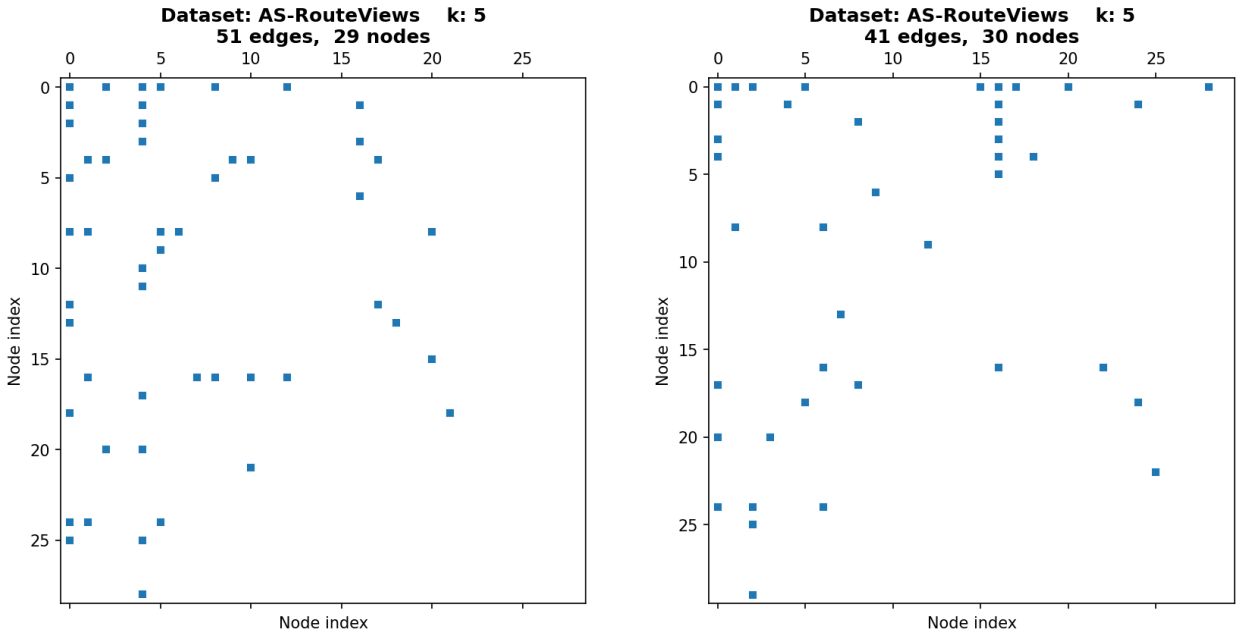


Figure 8.25: Spy Plots

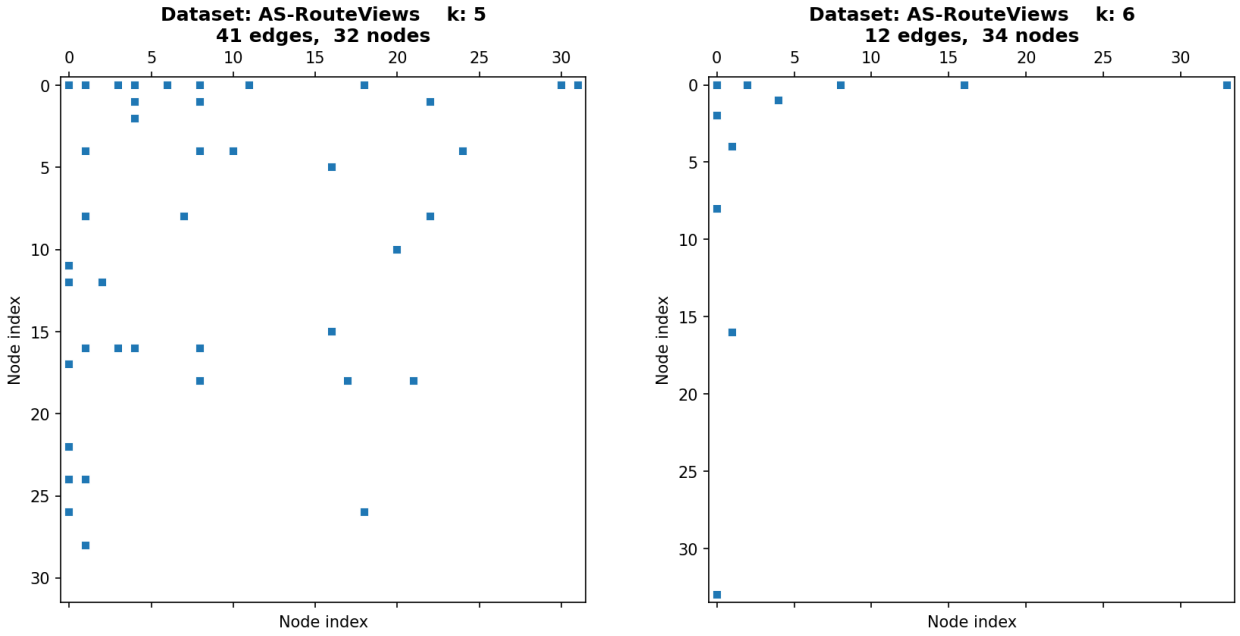


Figure 8.26: Spy Plots

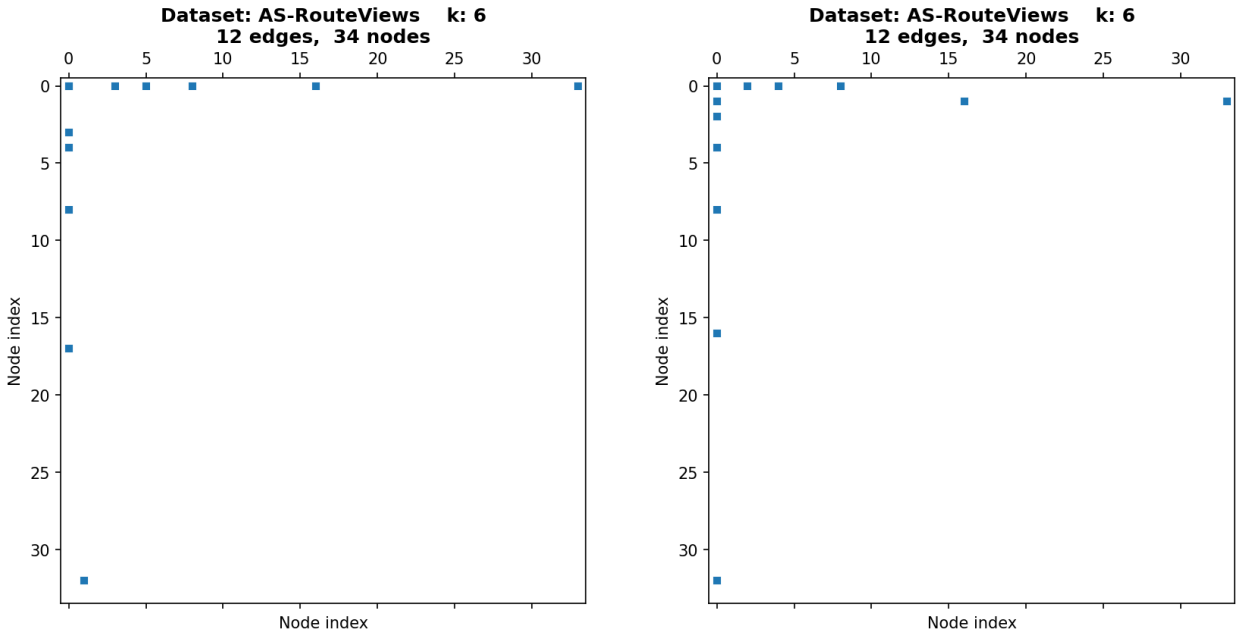


Figure 8.27: Spy Plots

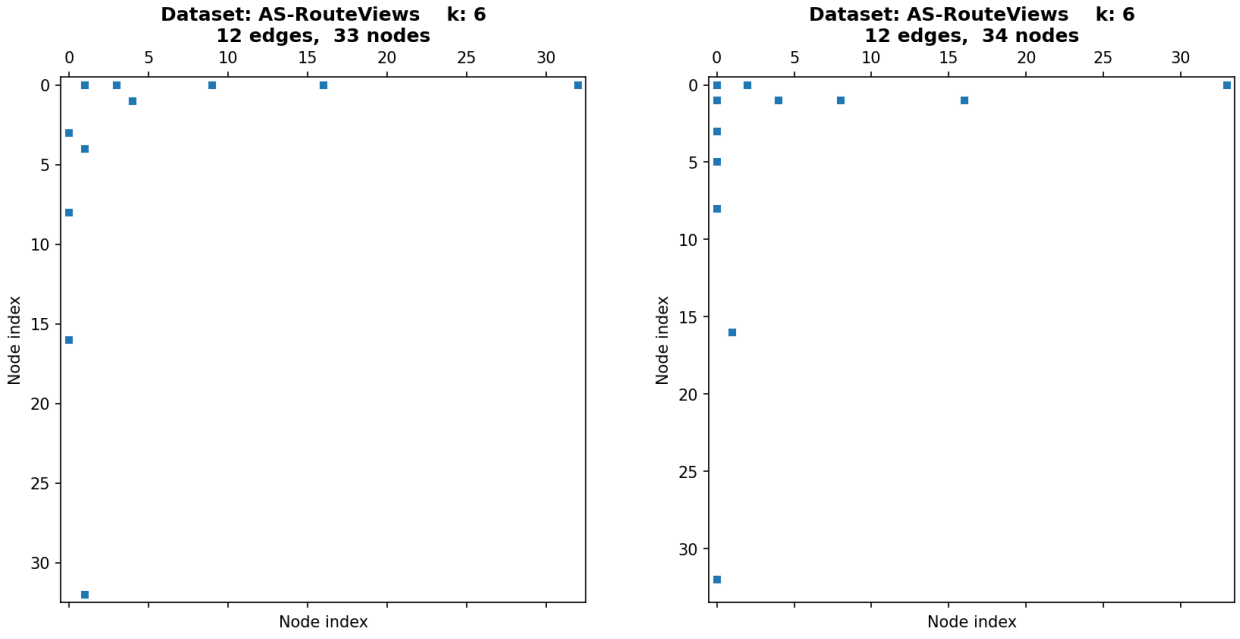


Figure 8.28: Spy Plots

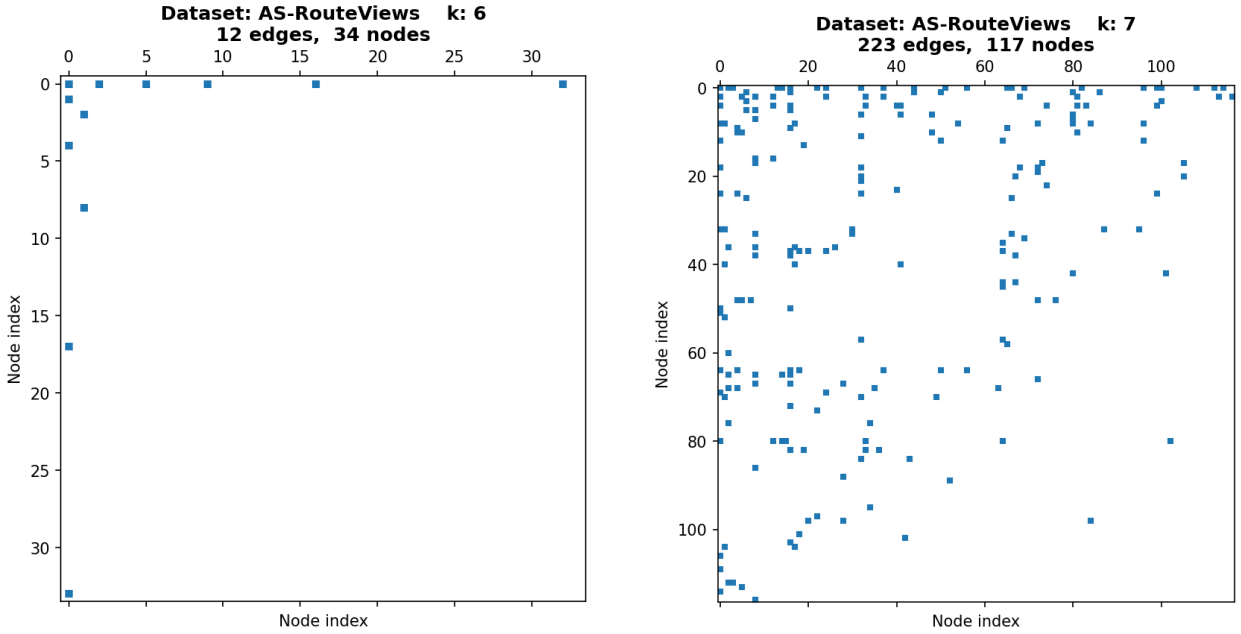


Figure 8.29: Spy Plots

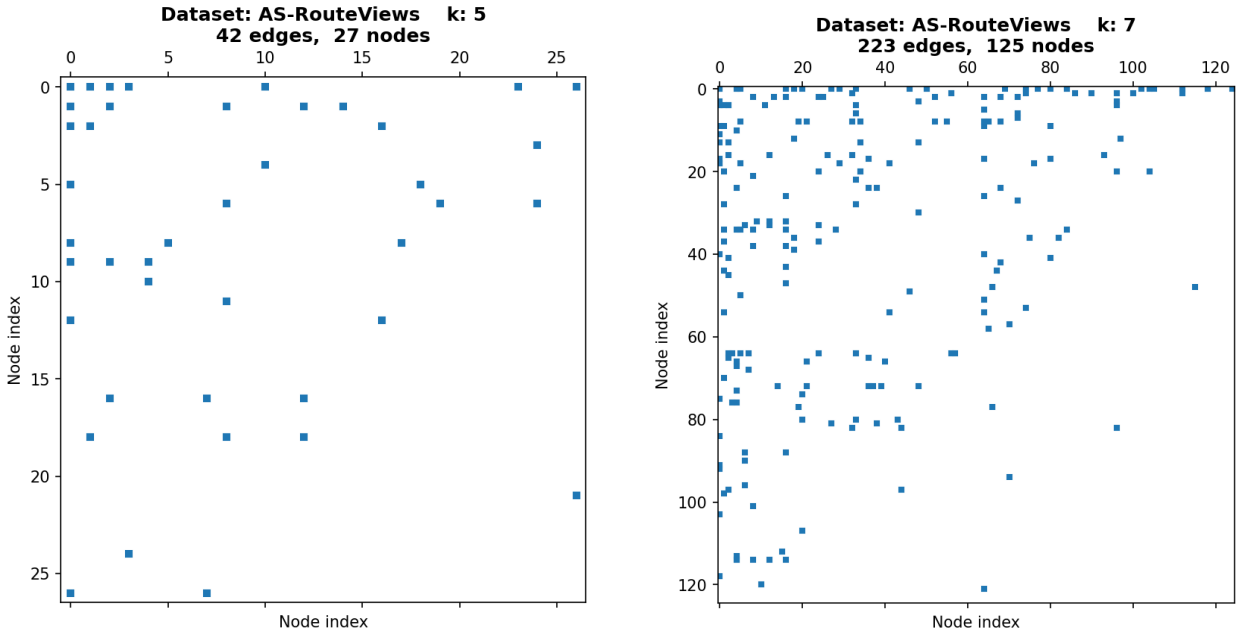


Figure 8.30: Spy Plots

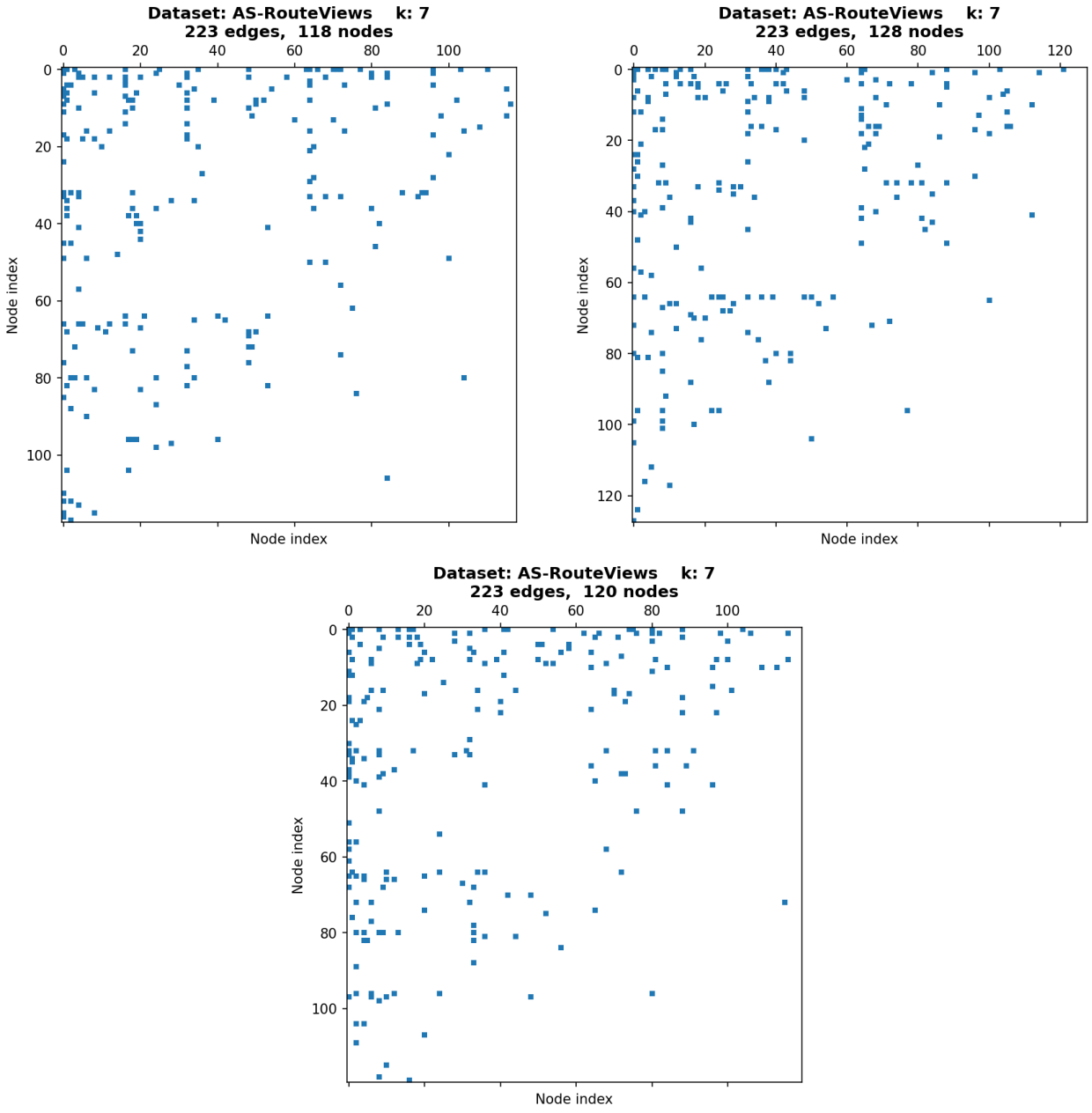


Figure 8.31: Spy Plots

Bibliography

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. ISBN 9781450378956. doi: 10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. volume 6, 04 2004. doi: 10.1137/1.9781611972740.43.

John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415. URL https://doi.org/10.1145/42411.42415.

Myunghwan Kim and Jure Leskovec. *The Network Completion Problem: Inferring Missing Nodes and Edges in Networks*, pages 47–58. doi: 10.1137/1.9781611972818.5. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611972818.5.

Myunghwan Kim and Jure Leskovec. Modeling social networks with node attributes using the multiplicative attribute graph model, 2011. URL https://arxiv.org/abs/1106.5053.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

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. ISBN 9781595937933. doi: 10.1145/1273496.1273559. URL https://doi.org/10.1145/1273496.1273559.

Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks, 2009. URL https://arxiv.org/abs/0812.4905.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Mamatha N, Sunitha S.S, and Shivakumar M D. Graph theory and its role in social network analysis. *World Journal of Advanced Research and Reviews*, 21(2):2088–2093, February 2024. ISSN 2581-9615. doi: 10.30574/wjarr.2024.21.2.0249. URL http://dx.doi.org/10.30574/wjarr.2024.21.2.0249.

Ayushi Patil, Shreya Mahajan, Jinal Menpara, Shivali Wagle, Preksha Pareek, and Ketan Kotecha. Enhancing fraud detection in banking by integration of graph databases with machine learning. *MethodsX*, 12:102683, 2024. ISSN 2215-0161. doi: https://doi.org/10.1016/j.mex.2024.102683. URL https://www.sciencedirect.com/science/article/pii/S2215016124001377.

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. doi: 10.1109/JPROC.2004.840306.

Naichen Shi, Dawei Li, Mingyi Hong, and Ruoyu Sun. RMSprop converges with proper hyper-parameter. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=3UDSdyIcBDA.

Richard M Veras. CAREER:High Performance Code Generation for Irregular Computation via a Graph Structure Descriptor Language. NSF Award Number 2440646. Directorate for Computer and Information Science and Engineering, Division of Computing and Communication Foundations. 2025., January 2025.

David M. Zoltowski, Skyler Wu, Xavier Gonzalez, Leo Kozachkov, and Scott W. Linderman. Parallelizing mcmc across the sequence length, 2025. URL https://arxiv.org/abs/2508.18413.