UNIVERSITY
OF OSLO

Master's thesis

Performance Modeling and Reordering Selection for Sparse Matrix-Vector Multiplication Using Machine Learning

Mads S. Balto

Computational Science
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences

Spring 2025

Mads S. Balto

Performance Modeling and Reordering Selection for Sparse Matrix-Vector Multiplication Using Machine Learning

Supervisors:
Xing Cai
James D. Trotter
Morten Hjorth-Jensen

Abstract

Sparse matrix-vector multiplication (SpMV) is a performance-critical kernel in scientific computing, particularly in simulations that solve large, sparse linear systems. Its performance is typically limited not by arithmetic throughput, but by memory access patterns shaped by the matrix structure.

This thesis investigates the use of machine learning (ML) to predict SpMV execution time and to identify performance-enhancing reordering algorithms. All experiments assume the Compressed Sparse Row (CSR) format for storing sparse matrices. Reordering algorithms permute the rows and/or columns of a matrix to improve memory access locality. A wide range of such algorithms exist, including Reverse Cuthill-McKee (RCM), Approximate Minimum Degree (AMD), and various graph- and hypergraph-based strategies. A dataset of 486 matrices and their reordered variants is used to train and evaluate several regression models, including least squares, ridge regression, and tree-based methods such as XGBoost. Execution time prediction is framed as a regression task using matrix-level features, while reordering selection is treated as a classification task.

XGBoost achieves strong performance in single-threaded SpMV time prediction ($R^2 = 0.95$, NRMSE = 86.2%, NMAE = 9.8%), and maintains moderate accuracy under multi-threaded execution ($R^2 = 0.85$, NRMSE = 145.3%, NMAE = 34.9%). Feature preprocessing offers negligible benefit, and hyperparameter optimization yields modest gains in the multi-threaded setting.

Classification-based reordering selection achieves 51–65% accuracy depending on architecture, with high top-3 coverage (typically >90%). On average, the selector yields a 1.10× speedup over the original ordering, peaking at 1.26×, while staying within approximately 1.16× of oracle performance.

To our knowledge, this is the first work to apply supervised learning to both SpMV time prediction and reordering selection using only matrix-level features and a fixed CSR representation. Prior auto-tuning systems such as SMAT [27], DIESEL [30], and WISE [44] explore broader design spaces including format and kernel selection, achieving 2–4× speedups. Our results show that meaningful, portable performance gains are achievable even within the narrower domain of reordering-only optimization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 SpMV in Physics

Sparse matrix-vector multiplication (SpMV) is a fundamental kernel in scientific computing, underpinning iterative solvers used in numerical simulation, optimization, and machine learning. In high-performance computing (HPC) settings, it often accounts for **50–90% of total runtime** in solvers for partial differential equations, structural mechanics, reservoir simulation, and more [10, 16, 33]. At scale, this translates to *millions of core-hours* annually devoted to a single operation. Any improvement, even at the percent level, directly impacts simulation turnaround, power usage, and scientific throughput.

In many iterative solvers for large sparse systems, the cost of SpMV dominates total runtime—particularly in high-performance computing contexts—making its performance a key factor in the efficiency of large-scale computational pipelines. Krylov subspace methods, such as Conjugate Gradient (CG) and GMRES, are the dominant iterative solvers in this setting and rely heavily on repeated SpMV [37, 38].

One prominent example is the Finite Element Method (FEM), widely used in structural mechanics and multiphysics simulations. Many open-source FEM frameworks—such as FEniCS, deal.II, and Elmer FEM—as well as proprietary solvers, assemble large sparse matrices and rely on iterative solvers in which SpMV accounts for a substantial portion of the runtime [42].

Finite Difference Methods (FDM) also give rise to large sparse linear systems, particularly in implicit formulations. These systems are typically solved using iterative algorithms such as CG or GMRES, both of which depend heavily on repeated SpMV operations [31]. As a result, SpMV is a performance-critical kernel in FDM-based solvers applied in domains such as astrophysics, seismology, and climate modeling [33].

Finite Volume Methods (FVM) are widely used in fluid dynamics, climate modeling, and subsurface flow simulation. Like FEM and FDM, they often result in sparse linear systems that must be solved at each time step, particularly in implicit formulations. In the energy and geoscience domain, the Open Porous Media (OPM) Flow simulator employs iterative solvers such as BiCGStab and GMRES to solve pressure and coupling equations. For some reservoir models, the linear solver phase alone has been reported to account for 90% or more of total simulation time [32].

These examples not only illustrate the ubiquity of SpMV, but also its computational significance. In many of these simulations, SpMV emerges as the dominant cost within iterative solvers, often accounting for the majority of the runtime [32, 33]. This bottleneck effect makes SpMV a prime target for optimization efforts in high-performance computing.

1.2 Addressing the Bottleneck

Optimizing SpMV performance requires addressing its sensitivity to memory access patterns and sparsity structure. While format selection plays a role, this thesis assumes a fixed storage format—Compressed Sparse Row (CSR)—which is widely used and compatible with the reordering algorithms explored. Instead, we focus on matrix reordering: permuting the rows and columns of a matrix to improve memory locality and reduce stalls. By reshaping the sparsity pattern prior to SpMV execution, reordering can significantly influence runtime, especially in bandwidth-limited settings [43].

Reordering effectiveness is highly matrix-dependent. The same algorithm may accelerate one matrix while degrading performance on another [43]. This variability complicates manual tuning and motivates a data-driven approach. In this thesis, we evaluate two complementary machine learning tasks: regression, to predict SpMV execution time from matrix-level features; and classification, to identify the reordering algorithm that yields the best runtime.

1.2.1 Motivating Reordering Selection

Figures 1.1 and 1.2 illustrate this matrix-specific variability in reordering performance. They show SpMV slowdowns relative to the best-performing reordering for three representative matrices, under both single-thread and 64-thread execution.

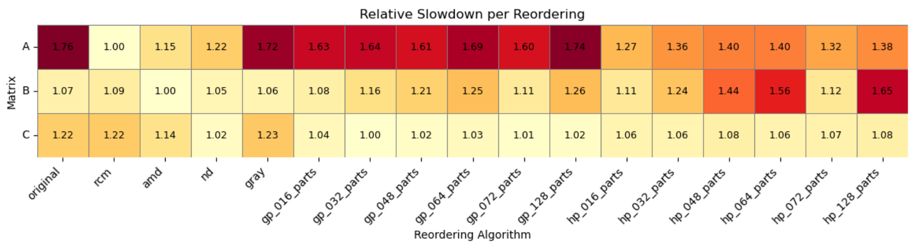

Figure 1.1: Slowdown of *single-thread* SpMV execution times for three SuiteSparse matrices across six reordering algorithms (see Section 2.1.3), relative to the best-performing reordering.

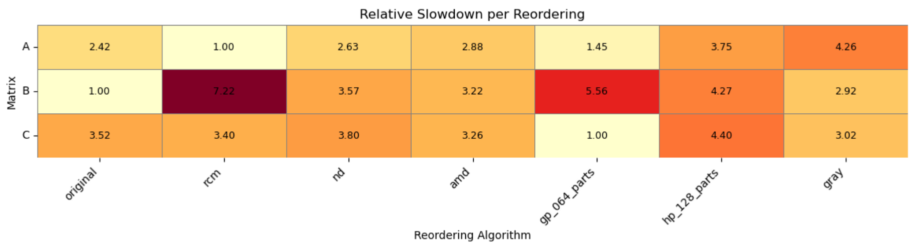

Figure 1.2: Slowdown of *64-thread* SpMV execution times for three SuiteSparse matrices across six reordering algorithms (see Section 2.1.3), relative to the best-performing reordering.

As the figures show, the best reordering differs per matrix, and poor choices can incur significant performance penalties. In some cases, reordering boosts throughput by

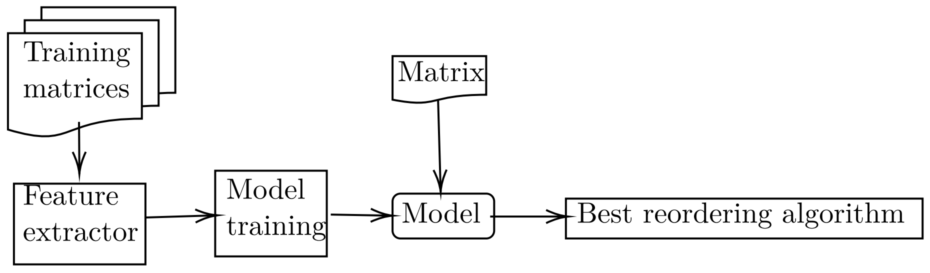

Figure 1.3: High-level overview of the machine learning pipeline used to predict the most performance-enhancing reordering algorithm for SpMV.

up to 40×; in others, it degrades performance by an order of magnitude. This motivates the need for learned models that go beyond static heuristics.

1.3 Pipeline

Figure 1.3 shows a high-level machine learning pipeline for predicting the most performance-enhancing reordering algorithm. A similar pipeline is used for regression, adapted to predict execution time.

1.4 Research Questions

This thesis explores the feasibility of using machine learning to improve SpMV performance by modeling structural characteristics of sparse matrices. We focus on the following questions:

- **Prediction Accuracy:** How accurately can machine learning models predict SpMV execution time from matrix-level features?
- **Generalization Across Execution Modes:** To what extent do models trained on single-threaded SpMV generalize to multi-threaded execution?
- **Reordering Selection:** How effectively can classification models identify the optimal reordering algorithm for a given matrix?

Taken together, our findings suggest that learning-based reordering decisions can be integrated into solver pipelines, delivering near-optimal SpMV performance with minimal overhead—without requiring exhaustive trial-and-error or domain-specific heuristics.

Chapter 2

Background

This chapter introduces the technical foundations necessary to understand the models and experiments presented in this thesis. We begin with a detailed overview of sparse matrix storage formats and matrix-vector multiplication, including the Compressed Sparse Row (CSR) format and performance characteristics of SpMV. We then cover key machine learning concepts such as regression, classification, model selection, and evaluation procedures. Finally, we define the metrics used to evaluate model performance.

2.1 Sparse Matrix Fundamentals

A matrix is typically considered sparse when most of its elements are zero, and when specialized data structures and algorithms can be used to exploit that sparsity. From a numerical perspective, sparsity often implies that only the nonzero values are stored, using formats that avoid representing zeros explicitly [1, 37]. Sparse matrices are ubiquitous in scientific computing, arising in both physics-based simulations and a range of non-PDE domains such as circuit design and economics [4]. They warrant techniques beyond those designed for dense matrices, as applying dense methods to sparse systems leads to unnecessary computation and memory use [1]. The most important of these techniques are sparse storage formats, which store only the non-zero values, implicitly representing the zeros. There are several formats such as COO [13], Compressed Sparse Row (CSR) [13], ELL [13], HYB[3], and DIA [13]. Some are specialized and others more generalized. Our analysis implements CSR as it remains the de facto standard sparse format [13].

2.1.1 Compressed Sparse Row (CSR) Format

Although many sparse storage formats exist, such as COO, ELL, HYB, and DIA, this thesis focuses exclusively on CSR. This choice reflects widespread CSR use in numerical software libraries and its compatibility with the reordering techniques and performance modeling tasks investigated in this thesis [13].

For understanding CSR, it is helpful to first define the COO storage format. COO consists of three arrays: `RowIdx`, `ColIdx`, and `Vals`, storing the corresponding row-indices, column-indices, and values of the nonzero entries of a sparse matrix, respectively. Gao et al. [13] express the storage size of COO in bytes, S_{COO}:

$$S_{\text{COO}} = \text{nnz}(2S_{\text{idx}} + S_{\text{val}}), \tag{2.1}$$

where nnz is the number of non-zeroes, and S_{idx} (typically 4 bytes) and S_{val} (typically 8 bytes) are the storage size of indices and values.

CSR ensures that all non-zeros belonging to the same row are stored contiguously in memory. The format further compresses the storage size by altering the row indexing of the non-zeros. Where COO explicitly maps each value to its corresponding row-index, CSR does this implicitly. This is achieved by introducing an array `RowPtr` of length $m+1$, where m is the number of rows in the matrix. The i-th entry in `RowPtr` contains the index in the `ColIdx` and `Vals` arrays where the non-zero entries for row i begin. The entries for row i therefore span the interval $\texttt{RowPtr[}i\texttt{]}$ to $\texttt{RowPtr[}i+1\texttt{]}-1$, inclusive. The final element $\texttt{RowPtr[}m\texttt{]}$ is always equal to nnz, marking the end of the last row.

Gao et al. also express the storage size of CSR in bytes, S_{CSR}:

$$S_{\text{CSR}} = (m+1)S_{\text{idx}} + \text{nnz}(S_{\text{idx}} + S_{\text{val}}). \tag{2.2}$$

Simple algebra shows that S_{CSR} is less than S_{COO} when nnz is greater than $m+1$, which overwhelmingly often is the case. This clearly establishes CSR as preferable to COO in otherwise similar implementation details.

2.1.2 Sparse Matrix-Vector Multiplication (SpMV)

For a dense vector $\mathbf{x}$ stored as `Vec` and a sparse matrix A stored in the CSR-format, the multiplication $\mathbf{y} \leftarrow A\mathbf{x}$ amounts to computing:

$$y_i = \sum_{k=\texttt{RowPtr}[i]}^{\texttt{RowPtr}[i+1]-1} \texttt{Vals}[k] \times \texttt{Vec}[\texttt{ColIdx}[k]]. \tag{2.3}$$

SpMV is a memory-bound operation [17, 30, 43]. Its performance is rarely limited by floating-point throughput but rather by how effectively data is moved through the memory hierarchy [17, 30, 43]. This hierarchy consists of multiple levels: CPU registers (fastest and smallest), L1, L2, and L3 caches (increasing in size and latency), and main memory (largest but slowest) [17]. Each level is progressively further from the processor core and more expensive to access in terms of latency and energy cost [17].

The effectiveness of memory usage is governed by the concepts of cache hits and cache misses [17]. A cache hit occurs when the requested data resides in a faster memory level (e.g. L1), while a miss results in costly fetches from slower memory levels [17]. For SpMV, an irregular access pattern to the input vector `Vec`, driven by the column indices stored in `ColIdx`, can occur. This limits cache reuse and increases the miss rate, thus becoming a performance bottleneck [4, 13, 17].

To describe the theoretical span in performance for SpMV, we introduce the concept of attainable performance, which represents the upper bound on practical execution performance. Let:

$$P_{\text{attainable}} = \min\left(P_{\text{peak}}, \frac{b_{\text{peak}}}{B_c}\right), \tag{2.4}$$

where P_{peak} denote the theoretical peak floating-point throughput of the hardware [GFlops/s], b_{peak} the possible memory bandwidth [GWords/s], and B_c the code balance, defined as the ratio of data traffic to floating point ops [words/flops]. Note that P_{peak} and b_{peak} are both fully hardware dependent, whereas B_c is problem dependent. For memory-bound operations such as SpMV, we can maximize $P_{\text{attainable}}$ by minimizing B_c [17].

For a sparse matrix A with m rows, n columns, and nnz nonzero elements in the CSR-format, a corresponding $\mathbf{x}$ vector with n elements, and their product $\mathbf{y}$ with m elements, we represent them as arrays:

- `Vals` with nnz elements of byte size 8, representing the non-zero values.
- `ColIdx` with nnz elements of byte size 4, representing the corresponding column indices of `Vals`.
- `RowPtr` with $m+1$ elements of byte size 4, representing the indices of `Vals` and `ColIdx` that correspond to the first non-zero element per row.
- `Vec`, n elements of byte size 8, representing the elements of $\mathbf{x}$.
- `y`, m elements of byte size 8, representing the elements of $\mathbf{y}$.

We assume that all arrays are stored contiguously in memory.

We model the best- and worst-case memory traffic. `Vals` and `ColIdx` elements are all accessed once contiguously, incurring a combined $4+8 = 12 \times nnz$ bytes contribution to the traffic. `RowPtr` values are accessed mostly twice per row. Since these accesses are contiguous, they typically do not cause extra cache misses. The total memory traffic from `RowPtr` is therefore $(m+1) \times 4$ bytes. `y` is written to m times incurring a $8m$ bytes contribution. We assume all n elements of `Vec` are loaded from main memory only once, and remain in cache when reused. In the best-case scenario, each access to `Vec` is a cache hit, corresponding to a contribution of $8n$ bytes. This represents a total traffic in bytes of

$$12nnz + 12m + 8n + 4. \tag{2.5}$$

We furher assume that each cacheline is 64 bytes. In the worst case, each access to `Vec` is a cache-miss, meaning a whole cache line is loaded each time, incurring a contribution of $64nnz$ bytes. This represents a total traffic in bytes of

$$76nnz + 12m + 4. \tag{2.6}$$

Assuming $nnz \gg m, n$, the ratio of the worst- and best-case memory traffic is approximately 6.33. Recall that code balance is the ratio of data traffic to floating point ops. Because the number of floating point ops remains constant, the ratio of worst- and best-case memory traffic equals the ratio of of worst- and best-case code balance. We express the worst-case code balance $B_{c,\text{worst}} = 6.3B_{c,best}$. The direct implication is that attainable performance can in the best case can be $5.33\times$ faster relative to the worst case, as shown by:

$$\frac{P_{\text{attainbable,best}} - P_{\text{attainable,worst}}}{P_{\text{attainable,worst}}} = \frac{\frac{b_{\text{peak}}}{B_{c,\text{best}}} - \frac{b_{\text{peak}}}{B_{c,\text{worst}}}}{\frac{b_{\text{peak}}}{B_{c,\text{worst}}}} \approx 5.33. \tag{2.7}$$

(Recall also that SpMV is memory bound).

SpMV performance on multi-threaded CPU contexts may also be affected by load balance, which arises when rows of the matrix have widely varying numbers of nonzeros [17, 42]. In such cases, uniform distribution of rows to threads can lead to some threads completing early while others remain active, reducing overall efficiency [42]. A higher imbalance factor indicates poor resource utilization and directly translates into degraded performance [17, 42].

Reordering techniques, explored in Section 2.1.3, are primarily designed to improve memory access locality, and may also impact computational load distribution [42].

2.1.3 Reordering Algorithms

Reordering algorithms apply permutations to the rows, columns, or both dimensions of a sparse matrix in order to modify its structure [43]. These permutations preserve all numerical information and allow recovery of the original ordering [43]. They are typically designed to optimize structural objectives such as matrix bandwidth, fill-in, edge-cut, and cut-net, which serve as proxies for SpMV performance [43]. In this context, an edge-cut is the number of edges that cross the k clusters produced by the partitioner; the reordering seeks to minimize that number. Net-cut is the hypergraph analogue of edge-cut: a net (hyperedge) spans all non-zeros that reference the same column; minimizing cut nets keeps those references inside one block.

Here we use the classical *row–net* model: each matrix row becomes a vertex and an (undirected) edge connects two vertices when the corresponding rows share at least one column index *(for non-symmetric matrices we simply apply the test to the pattern of $A+A^{\mathsf{T}}$ so the graph is undirected).*

These objectives are only indirectly related to execution time of SpMV, and the effectiveness of a given reordering depends strongly on the matrix structure. Empirical studies consistently show that no single algorithm dominates across all matrices [13, 43].

Performance outcomes from reordering are highly variable, with the same algorithm sometimes yielding speedups and other times degrading performance, depending on the matrix structure [43]. This variability stems from how reorderings affect features that govern memory locality and load balance in non-uniform ways [43].

Consequently, selecting an optimal reordering for a given matrix remains a challenging and often unpredictable task, motivating data-driven approaches supported by the concepts outlined in Section 2.2.

We focus on the following widely studied reordering algorithms in Table 2.1, adapted from [43].

Table 2.1: Sparse matrix reordering algorithms evaluated in this study. Descriptions adapted from [43].

Short Name	Reordering Algorithm	Description
RCM [28]	Reverse Cuthill–McKee	Bandwidth reduction via breadth-first graph traversal
AMD [2]	Approximate Minimum Degree	Local greedy strategy to reduce fill by selecting sparsest pivot row
ND [25]	Nested Dissection	Recursive divide-and-conquer using vertex separators to reduce fill
GP [25]	Graph Partitioning (METIS)	Multi-level recursive graph partitioning with METIS using edge-cut objective
HP [6]	Hypergraph Partitioning (PaToH)	Column-net hypergraph partitioning with PaToH using cut-net metric
Gray [46]	Gray Code Ordering	Splitting of sparse and dense rows and Gray code ordering

With the exception of Gray, every reordering considered here (RCM, AMD, ND, GP, HP) produces *one* permutation that we apply to both rows and columns. For non-symmetric matrices we first form the structural pattern of $A + A^{\mathsf{T}}$ so the graph (or hypergraph) is undirected before partitioning. This symmetric application preserves the intended block reuse of x_j values.

Gray ordering is handled differently: it is a row-only scheme that clusters rows with similar sparsity patterns and leaves the column order unchanged.

Note that both GP and HP are not single reordering algorithms, but families of methods. Their behavior depends on configuration parameters, most notably the number of partitions into which the graph or hypergraph is divided. This parameter substantially affects the resulting matrix structure and, by extension, SpMV performance. For example, METIS (GP) and PaToH (HP) may produce different reorderings depending on the number of parts they are configured with, e.g. 32, 64, or 128 parts.

2.2 Machine Learning Fundamentals

Machine learning is the study of algorithms that improve their performance at a task through experience. Mitchell [29] defines it formally: *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

This section introduces core machine learning concepts and techniques that underpin the predictive models developed in this thesis.

2.2.1 Problem Formulation

Supervised learning tasks are commonly categorized into regression and classification [29]. Regression involves predicting a continuous-valued target given a set of input features [29]. In this thesis, we formulate regression problems to predict the execution time of SpMV based on structural properties of the sparse input matrix. Classification, by contrast, refers to predicting a discrete label from input features [29]. Here, it is used to predict the best reordering algorithm for a given matrix that minimizes SpMV execution time.

2.2.2 Linear Models as Baselines: OLS and Ridge Regression

Following standard practice in machine–learning studies, which recommends beginning with simple, interpretable models as a yardstick [9, 14], we adopt linear regression methods as baselines for predicting SpMV execution time. In particular, we use Ordinary Least Squares (OLS) and its ℓ_2 regularized variant, Ridge regression [18, 20]. Their closed-form solutions give negligible training cost and coefficients that map directly to feature influence, making them an ideal reference point for gauging the extra value yielded by more flexible non-linear models such as XGBoost see Section 2.2.4).

Linear regression refers to a class of supervised learning models that assume a linear relationship between input features and a continuous target variable [29]. Given a feature matrix $X \in \mathbb{R}^{m \times n}$ with m samples and n features, and a target vector $y \in \mathbb{R}^m$, the model assumes:

$$\hat{y} = X\mathbf{w}, \tag{2.8}$$

where $\mathbf{w} \in \mathbb{R}^n$ is the vector of coefficients to be learned [29]. The most common fitting method is ordinary least squares (OLS), which minimizes the objective $\mathcal{L}$, i.e. mean squared error (MSE), between predicted and observed values [29]:

$$\mathcal{L}_{\text{OLS}}(\mathbf{w}) = \|X\mathbf{w} - y\|_2^2 = \frac{1}{m}\sum_{i=0}^{n-1}(X\mathbf{w} - y_i)^2. \tag{2.9}$$

The solution is given by the normal equation:

$$\mathbf{w}_{\text{OLS}} = (X^T X)^{-1} X^T y, \tag{2.10}$$

assuming $X^T X$ is invertible [39]. This formulation is simple and interpretable, but it can be unstable in the presence of multicollinearity or when the number of features is large relative to the number of samples [20, 39].

To address these issues and improve generalization, regularization is commonly applied [29]. Ridge regression modifies the OLS objective by adding a penalty on the

ℓ_2 norm of the coefficient vector, which discourages large weights by minimizing their squared magnitudes [20]:

$$\mathcal{L}_{\text{ridge}}(\mathbf{w}) = \|X\mathbf{w} - y\|_2^2 + \lambda\|\mathbf{w}\|_2^2, \tag{2.11}$$

where $\lambda \geq 0$ is a hyperparameter controlling the strength of regularization. The resulting closed-form solution becomes [20]:

$$\mathbf{w}_{\text{ridge}} = (X^T X + \lambda I)^{-1} X^T y. \tag{2.12}$$

Ridge regression, introduced by Hoerl and Kennard [20], shrinks coefficients toward zero, improving model stability and reducing overfitting. It retains all features while limiting their influence, making it a robust baseline for high-dimensional or collinear datasets. Ridge regression remains a standard regularization technique in modern machine learning pipelines [14].

2.2.3 Classification for Reordering Selection

Classification is a supervised learning task in which the goal is to predict a discrete label from a set of input features [29]. Unlike regression, which predicts a continuous value, classification assigns inputs to one of several predefined categories. In this thesis, classification is used to select the reordering algorithm that minimizes the execution time of SpMV for a given matrix. This formulation sidesteps direct runtime prediction and instead treats algorithm selection as a decision task based on structural features. Model performance is evaluated using accuracy, top-k accuracy, and class-wise precision and recall, rather than continuous error metrics such as RMSE or MAE, all of which are explained in Section 3.4.3.

Our data are class-imbalanced, i.e. some reorderings are optimal for many more matrices than others. Moreover, misclassifications have very different consequences, selecting a near-tie costs almost nothing, whereas picking a far-slower method can multiply runtime. To capture these nuances we report, in addition to top-k accuracy and per-class precision/recall, two runtime-sensitive measures: regret (per-instance slowdown relative to the optimum) and relative total time (RTT), the ratio of cumulative runtime to an oracle chooser; see Section 2.3.7.

2.2.4 Gradient Boosting and XGBoost

Boosting is an ensemble technique that combines multiple weak learners, models that perform slightly better than random guessing, into a strong learner [29]. These weak learners may be classifiers or regressors, depending on the task [29]. The AdaBoost algorithm, introduced by Freund and Schapire [11], demonstrated that iteratively training weak classifiers on reweighted data, emphasizing previously misclassified examples, could significantly improve classification performance.

Gradient Boosting, formalized by Friedman [12], generalizes the boosting framework to arbitrary differentiable loss functions. Rather than reweighting data, it fits each new learner to the residual errors of the current ensemble, effectively performing functional gradient descent in function space. This formulation enables boosting to be applied to regression tasks, where the goal is to minimize continuous loss functions such as mean squared error.

XGBoost (Extreme Gradient Boosting) [7] is a widely used and highly optimized implementation of gradient boosting. Each weak learner in XGBoost is a shallow decision tree, and new trees are added sequentially to correct the residual errors of the ensemble.

It improves training efficiency and generalization through techniques such as second-order Taylor approximation of the loss function, shrinkage, column subsampling, and sparsity-aware algorithms. These enhancements make XGBoost particularly well-suited for structured (tabular) data, where it often outperforms deep learning models with significantly less tuning [40]. Its practical impact has been widely recognized, including being named InfoWorld's 2019 Technology of the Year [5].

In this thesis, XGBoost is applied to both regression tasks, predicting SpMV execution time, and classification tasks, identifying the optimal reordering algorithm.

XGBoost was selected for its strong empirical performance on tabular data and its ability to handle heterogeneous, non-normalized features without extensive preprocessing [40]. These properties align well with the structural features extracted from sparse matrices in this thesis.

XGBoost exposes a number of hyperparameters that influence model complexity and regularization. Key parameters include:

- `n_estimators`: The number of boosting rounds (trees). More trees can increase model capacity but risk overfitting.
- `learning_rate`: Step size shrinkage to control the contribution of each tree. Smaller values slow learning but often improve generalization.
- `max_depth`: Maximum depth of each decision tree. Controls model complexity; deeper trees capture more interactions.
- `min_child_weight`: Minimum sum of instance weights required to create a new leaf. Larger values increase regularization.
- `subsample`: Fraction of the training data sampled per boosting round. Prevents overfitting by introducing randomness.
- `colsample_bytree`: Fraction of features sampled per tree. Helps reduce correlation among trees and overfitting.
- `gamma`: Minimum loss reduction required to make a split. Acts as a pruning threshold.
- `reg_alpha`: L1 regularization term on weights (analogous to Lasso).
- `reg_lambda`: L2 regularization term on weights (analogous to Ridge).

These parameters are often tuned to balance underfitting and overfitting, especially when input features are diverse or noisy.

2.2.5 Feature Preprocessing

Feature Scaling and PCA

Feature preprocessing techniques, such as scaling and dimensionality reduction, are common steps in machine learning pipelines that can improve model performance or stability [14]. OLS and Ridge (see Section 2.2.2) are algorithms that can benefit from both.

Feature scaling refers to standardizing input features by subtracting the mean and dividing by the standard deviation of each column. This is especially beneficial for models that are sensitive to feature magnitude, such as linear regression and ridge regression. Feature scaling can reduce variance in OLS, and ensures the penalty proportionally affects the features in Ridge.

Principal Component Analysis (PCA) is a dimensionality reduction technique that projects data into a new orthogonal basis defined by directions of maximum variance [24]. By keeping only the leading components, PCA can help reduce noise and multicollinearity, and may improve generalization in models that struggle with high-dimensional input, such as OLS and Ridge. In the context of SpMV, structural features such as standard deviation of nonzeros per row (sd), diagonal occupancy (ndiag), or fill-in ratios may not align with directions of maximal variance, causing PCA to discard components that encode performance-relevant structure.

However, not all models benefit from preprocessing. Tree-based methods like XGBoost are invariant to monotonic transformations of the features and tend to perform well without scaling. PCA may even degrade performance by discarding useful structural information encoded in the original features [40].

As shown in Section 5.3, neither scaling nor PCA significantly improved prediction accuracy for SpMV execution time in our experiments. These findings reinforce that preprocessing must be considered in context, and cannot replace meaningful feature selection.

One-Hot Encoding

One-hot encoding refers to representing a categorical feature of n classes across $n-1$ binary features. Each class is represented by the binary feature, represented as corresponding category being 0 or 1, where as the category without a binary feature is implicitly represented by the remaining ones set to zero.

Categorical features are one-hot encoded so that the resulting binary indicators carry no implicit ordering, unlike integer (ordinal) encoding, which would falsely imply that, say, category 3 is 'closer to' category 2 than to category 0 [23].

To illustrate with an example let the reordring algorithms be RCM, Original, and AMD. Consider the representation of a RCM reordered matrix in the described feature example. The matrix features correspond to that of the original SuiteSparse, and the remaining two features correspond to a binary representation of RCM which has a value 1, and AMD has a value of 0. some representation of an original SuiteSparse matrix would simple have an RCM value of 0 and an AMD value of 0 too.

2.2.6 Avoiding Overfitting

An observed target value y_i can be expressed as the sum of an unknown underlying function $f(x_i)$ and an irreducible noise term ε_i:

$$y_i = f(x_i) + \varepsilon_i. \tag{2.13}$$

Overfitting occurs when a model learns to exactly reproduce the training targets, i.e. $\hat{y}_i = y_i$ for all i, thereby capturing not only the signal but also the noise in the data [19]. While this minimizes training error, it degrades performance on unseen data, undermining the model's ability to generalize [15].

To reliably detect overfitting, it is essential to evaluate model performance on data that was not seen during training. Using overlapping training and test sets typically results in overly optimistic metrics that do not reflect real-world performance [22]. A common approach is to partition the dataset into disjoint subsets: a *training set* for model fitting and a *test set* for final evaluation. This separation ensures that the model's generalization ability can be estimated under realistic conditions. Every instance of paired training and evaluation of models in this thesis partition the data such that 90% is allocated in a training set, with the remaining 10% allocated to the test set.

2.2.7 Cross Validation

Cross-validation is a resampling technique used to assess a model's ability to generalize to unseen data. It provides an estimate of out-of-sample performance and helps monitor model stability during training [36, 41].

In k-fold cross-validation, the dataset is partitioned into k approximately equal-sized subsets, or *folds*. Each fold is used once as a validation set while the remaining $k-1$ folds form the training set. This process repeats until every fold has served as the validation set exactly once, and the performance scores are averaged to provide an overall estimate [26].

Cross-validation is also widely used for model selection, such as choosing hyperparameters that maximize average validation performance while minimizing overfitting risk [15].

We adopt 5-fold cross-validation throughout the study. For the regression models we apply a standard 5-fold split, whereas for the classification models we employ *stratified* 5-fold cross-validation so that each fold preserves the class distribution. Across the five rotations every fold serves once as the validation set; we then average the per-fold metrics, RMSE, MAE, and R^2 for regression, and top-k accuracy, and RTT for classification, to estimate generalization performance.

2.3 Evaluation Metrics

Defining and understanding evaluation metrics precedes interpretation of model performance. This section defines Mean Absolute Error (MAE), Root Mean Square Error (RMSE) and the Coefficient of Determination (R^2).

2.3.1 Mean Absolute Error (MAE)

MAE is defined for a set of target values $\mathbf{y}$ and their predicted counterparts $\hat{\mathbf{y}}$ as:

$$\mathrm{MAE} = \frac{1}{N} \sum_{i=0}^{N-1} |y_i - \hat{y}_i|, \tag{2.14}$$

where N is the number of samples. MAE scores a model linearly, penalizing errors absolutely. This is useful for a straightforward assessment of the accuracy of the model. A high MAE indicates that predictions, on average, deviate substantially from the true values. A downside to this metric for our evaluation purposes is that it is neither agnostic to hardware or threading vs. non-threading. Our analysis frequently compares model performance in these configurations, therefore we frequently normalize this metric by the mean target value of the set used, referring to this metric as Normalized MAE (NMAE).

2.3.2 Root Mean Square Error (RMSE)

RMSE is defined similarly to MAE, quantifying model error significantly differently:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}. \tag{2.15}$$

RMSE scores a model quadratically, penalizing larger errors more heavily than linear alternatives such as MAE. This makes it especially useful when large deviations are particularly undesirable. The result is expressed in the same units as the target variable, making it directly interpretable in terms of the original output. A high RMSE indicates that the model's predictions deviate substantially from the ground truth, either due to large individual errors or systematic bias. RMSE score interpreted in tandem with MAE can also indicate the error spread, i.e. RMSE $\gg$ MAE indicate significant outliers in model predictions, while RMSE $\approx$ MAE indicate evenly distributed inaccuracy. As with NMAE, we also compute and report normalized RMSE (NRMSE) the same way, and for the same reasons.

2.3.3 Coefficient of Determination

R^2, the coefficient of determination, is defined as:

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{\sum_{i=0}^{N-1}(y_i - \bar{y})^2}, \tag{2.16}$$

where $\bar{y}$ is the mean of the target values. The R^2 score measures the proportion of variance in the target that is explained by the model. A score of $R^2 = 1$ indicates perfect prediction, while $R^2 = 0$ implies that the model performs no better than always predicting the mean. Negative values indicate that the model performs worse than this baseline.

R^2 is useful for comparing the explanatory power of different models, particularly across datasets with varying scales. However, it does not convey the magnitude of prediction error and can be misleading in contexts where absolute accuracy matters, or when the target values have low variance. For this reason, R^2 is interpreted alongside MAE and RMSE in this thesis to provide a more complete view of model performance.

2.3.4 Accuracy and Top-k Accuracy

Accuracy is defined as the ratio between the number of correct predictions and the total number of predictions. In the context of our multi-class classification task, a prediction is correct if the model identifies the reordering algorithm that yields the lowest SpMV execution time for a given matrix.

While accuracy provides a concise summary of overall model performance, it does not indicate how far off incorrect predictions are from the optimal choice. In practice, the model may predict a reordering that performs nearly as well as the optimal, but this is still counted as incorrect. To account for such near-misses, we also report Top-k accuracy, which considers a prediction correct if the correct class is among the k most probable predictions [45].

2.3.5 Precision, Recall, and Class Support

We report additional class-wise metrics to diagnose model behavior across different reordering algorithms. Precision and recall are defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2.17}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{2.18}$$

where a true positive denotes a correctly predicted class label, a false positive denotes an incorrect prediction of that class, and a false negative indicates a missed prediction for that class.

These metrics provide insight into whether certain classes are under- or over-represented in the model's predictions. Class support refers to the number of true instances of each class and helps contextualize the reliability of precision and recall scores per class [45].

2.3.6 Confusion Matrix

A confusion matrix is a commonly used tool for evaluating the performance of a classifier across n classes. It is an $n \times n$ matrix where each row represents the true class labels and each column represents the predicted class labels. The entry at row i, column j indicates the number of instances of class i that were predicted as class j. This representation allows for a detailed analysis of class-wise prediction patterns and potential class imbalances.

Confusion matrices are particularly useful for identifying systematic misclassifications and assessing classifier performance beyond overall accuracy [35].

2.3.7 Runtime-Based Metrics for Classification

In addition to standard classification metrics, this thesis reports runtime-based metrics to better capture the performance implications of reordering decisions.

They complement classification accuracy by quantifying the real-world impact of misclassifications in terms of SpMV execution cost.

Relative Total Time (RTT)

RTT is defined as the ratio between the total execution time incurred by the model's reordering predictions and the total execution time under an oracle that always selects the optimal reordering. Consider the following example measurements:

Matrix	Model time T_{model}	Oracle time T_{oracle}
A	2.10	2.00
B	1.40	1.20
C	3.00	2.40

RTT is then computed such that:

$$\text{RTT} = \frac{2.10 + 1.40 + 3.00}{2.00 + 1.20 + 2.40} = \frac{6.50}{5.60} \approx 1.16.$$

A value close to 1.0 indicates that the model performs nearly as well as the best possible selection strategy.

Regret

Regret refers to the per-instance runtime penalty incurred by choosing a suboptimal reordering. For a given matrix, it is defined as the ratio between the execution time of the chosen reordering and that of the optimal one, i.e. $T_{\text{model}}/T_{\text{oracle}}$. A regret of 1.0 means the model selected the best option; higher values indicate how much slower the selected reordering was relative to the best.

Chapter 3

Methods

In this chapter we present the machine learning framework used for regression and classification tasks. The regression model estimates SpMV execution time for a given matrix and reordering, while the classification model identifies the reordering algorithm that yields the lowest execution time for a given matrix. Both models leverage two feature sets derived from the corresponding matrices.

3.1 Regression

We consider two regression problem formulations for predicting SpMV execution time for a matrix, each constructed with with different modeling assumptions and use cases.

3.1.1 SpMV Execution Time Prediction

The primary problem formulation predicts the SpMV execution time for a matrix with its corresponding matrix features. We refer to this problem formulation as SpMV execution time prediction.

This variant maximizes use of available structural information to directly model the SpMV runtime. It reflects a best-case scenario where matrix content is always accessible and runtime prediction accuracy is prioritized over interpretability or deployment convenience.

Results for this problem formulation are reported a cross a range of configurations in Sections 5.1.1, 5.1.2, 5.1.2, and 5.1.3.

3.1.2 Reordered SpMV Execution Time Prediction

A secondary problem formulation predicts the SpMV execution time for a matrix with the matrix features of its SuiteSparse form, extended by a supplementary set of one-hot encoded categorical features (see Section 2.2.5), describing the corresponding reordering algorithm applied to the SuiteSparse matrix.

To illustrate what this means, consider a feature matrix $X \in \mathbb{R}^{3\times 2}$ representing three matrices by arbitrary metrics each. Further assume that the first row corresponds to a SuiteSparse matrix, i.e. no reordering applied (that we know of), the second correspond to a SuiteSparse matrix reordered with RCM, and the last a SuiteSparse matrix reordered with AMD. We define a new matrix $Y \in \mathbb{R}^{\nVDash \times \nvDash}$ with their one-hot encoded labels:

$$Y = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{3.1}$$

where the first column is a binary category, one if the matrix is RCM-reordered, 0 otherwise. Equivalently with AMD, the second column is 1 if the matrix it represents is AMD-reordered, and otherwise 0. The case of when the matrix is neither, i.e. unordered, is represented by both columns represented as zero. Finally, we simply combine X and Y row-by-row into $\tilde{X} \in \mathbb{R}^{3\times 4}$, such that the features encode both the arbitrary metrics, aswell as the reordering algorithm applied.

We refer to this problem formulation (i.e. $\hat{X}$ is used as the feature matrix), as reordered SpMV execution time prediction.

This approach is conceptually related to the classification task. It assumes only the original (unordered) matrix is available at inference time. The model thus learns to estimate the runtime of a given reordering applied to a known matrix. This allows estimation of runtime for specific reorderings before execution, rather than predicting which reordering is optimal.

We implement `LabelEncoder` from Sci-kit learn for one-hot encoding [34] the reordering-algorithm labels as features.

Results for this problem formulation are shown in Section 5.1.2.

3.1.3 Implementation Details

We trained a range of regression models, all with DIESEL features, which are described in Section 3.3, with the exception of two that also used DIESEL features, but augmented as described in Section 3.1.2.

Targets used are specified by SpMV execution time measurements are single-threaded (ST) or multi-threaded, e.g. 64-threaded (MT64) (see Section 4.3), aswell as the CPU architecture (see Section 4.2) the measurements were made on. We disclose each setup:

1. **OLS and Ridge (ST).** Fitted as interpretable baselines on ST measurements on the TX2 CPU, using both raw features and the scaled/PCA variants.

2. **XGBoost (ST).** Trained and evaluated on the same ST measurements and feature variants.

3. **XGBoost (MT64).** Trained on MT64 TX2 measurements; results are reported with and without hyper-parameter optimisation.

4. **XGBoost (ST & MT64, reordered).** Trained for reordered SpMV prediction on the TX2. Six HP/GP variants ($n \in \{16, 32, 48, 64, 72, 128\}$) appear as extra categorical features in the ST data, whereas only one variant appears in the MT64 data, so the two datasets differ in size.

5. **XGBoost (thread sweep).** Trained on every CPU listed in Section 4.2 for $n \in \{16, 32, 48, 64, 72, 128\}$ threads, with hyper-parameter optimisation performed separately for each *(CPU, n)* pair.

The ST measurements include multiple internal configurations of HP and GP (the six n-part variants), whereas each n-thread measurements contains only the corresponding single configuration. Consequently, in the plain SpMV prediction task the ST dataset has more matrices (and therefore more rows) than its MT64 counterpart; in the reordered task the ST dataset instead has more categorical feature classes. The ST and MT64 datasets are therefore not fully analogous.

Results for all regression models are presented in Section 5.1.

A runnable reference implementation is also provided in Appendix A.

3.2 Classification

We define classification tasks aimed at predicting the best reordering algorithm (see Table 2.1) for a given sparse matrix, minimizing its execution time. Each instance corresponds to a base matrix from SuiteSparse, and the target label is the reordering algorithm that yields the lowest measured SpMV runtime in one-hot encoded format.

To illustrate the encoding, consider the example measures of SpMV execution times in matrices A, B and C and their reordered variants in Table 3.1 We see that for A, the

Matrix	**No Reordering**	**RCM**	**AMD**
A	0.5	1.1	1.2
B	1.2	0.5	1.7
C	1.3	1.5	0.2

Table 3.1: Example of SpMV execution time measurements for classification encoding explanation.

fastest reordering algorithm is none, for B it is RCM, and for C its AMD. We define $Y \in \mathbb{R}^{3\times 2}$ as the one-hot encoded representation of this, with column 1 is binary, 1 is RCM is the fastest and 0 otherwise, Equivalently so for column 2 with AMD:

$$Y = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{3.2}$$

We implement `LabelEncoder` from Sci-kit learn for one-hot encoding [34] the classification targets.

To mitigate severe class imbalance, we exclude the reordering algorithms of Hypergraph Partitioning (HP) and Graph Partitioning (GP) families entirely. These algorithms appear under multiple internal variants in the dataset, each with too few samples to support stable model training. All other reordering classes, i.e. RCM, AMD, ND, Gray, and Original, are retained in both ST and n-threaded contexts.

We train separate classification models for the ST and n-threaded measurements, using SpMV runtime labels measured on TX2 architecture. To contextualize model performance, we evaluate two baseline strategies: (i) always predict RCM, and (ii) never apply reordering (i.e., always select the Original matrix). These serve as simple heuristics requiring no features or training.

Evaluation of classification accuracy, top-k performance, and per-class metrics is presented in Section 5.2.

3.3 Feature Selection

Mohammed et al. [30] introduced a set of 14 features for predicting the optimal sparse storage format and SpMV kernel. They are presented along with their descriptions and computational complexities in Table 3.2. Rather than designing a new feature set, we apply this existing set to evaluate its applicability in our classification and regression contexts, as it has been successfully used in previous work [30] to characterize sparse matrices for SpMV performance modeling. Features marked with an asterisk (*) are sensitive to reordering, i.e., they may change depending on the permutation applied to the matrix rows and/or columns.

Table 3.2: Selected features from sparse matrices. Features marked with (*) are sensitive to reordering algorithms. Table reproduced from [30].

Feature	**Description**	**Complexity**
m	The number of rows	$O(1)$
n	The number of columns	$O(1)$
nnz	The total number of nonzero values in the matrix	$O(1)$
density	The density of the matrix, $nnz/(m \times n)$	$O(1)$
mean	The mean number of nonzero values per row (npr)	$O(1)$
sd*	The standard deviation of npr in a matrix	$O(2m)$
var*	The variance of npr in a matrix	$O(1)$
maxnnz*	The maximum of npr in a matrix	$O(m)$
maxavg*	The difference between $maxnnz$ and $mean$	$O(1)$
distavg*	The mean distance between first and last nonzero values in each row	$O(nnz)$
clusteravg*	The mean of the number of distinct consecutive npr	$O(nnz)$
ndiag*	The number of matrix diagonals with one or more nonzero values	$O(nnz)$
diagfill*	The diagonal fill-in ratio for matrices, $(ndiag \times m)/nnz$	$O(1)$
fill*	The fill-in ratio for matrices, $(m \times maxnnz)/nnz$	$O(1)$

3.4 Training and Evaluation

We describe the shared setup across all machine learning tasks. All stochastic components (data splits, hyperparameter sampling) are controlled using a fixed random seed for reproducibility.

3.4.1 Data Splitting

To evaluate generalization performance, 10% of the dataset is held out as a test set. The remaining 90% is used for cross-validation and model training. Splitting is performed using scikit-learn [34] with a fixed random seed to ensure reproducibility.

For classification tasks, we apply stratified sampling during the split to preserve class label distributions in both the training and test sets. This is necessary due to class imbalance, where some reordering algorithms occur far more frequently than others.

3.4.2 Cross-Validation

To assess model stability and robustness, we perform 5-fold cross-validation on the training set. The training data is partitioned into five equally sized folds using a fixed random seed for reproducibility. In each iteration, one fold is held out for validation while the remaining four folds are used for training. Performance metrics are computed on the validation fold and averaged across all folds. Cross-validation is implemented using scikit-learn [34].

3.4.3 Evaluation Metrics

We use distinct evaluation metrics for regression and classification tasks. We measure regression performance using NRMSE, NMAE, and R^2 score (see Section 2.3). These metrics are reported on the test set, and R^2 is averaged across cross-validation folds (see Section 5.1.1).

Classification Model performance is evaluated using accuracy, top-k accuracy ($k = 1, 3$), and class-wise precision and recall. These metrics reflect both overall prediction accuracy and per-class performance in imbalanced settings (see Section 5.2). We also make use of RTT (See Section 2.3.7), which we define to be the ratio of total time in which the model choices would've incurred, normalized by the sum of the fastest choices available. Lastly we use regret, which is the single-instance ratio of the time the model choice would've incurred divided by the fastest choice available. We also implement some confusion matrices for comparing

3.5 XGBoost

We choose XGBoost [7] for all ML tasks explored in this thesis due to its strong reputation and performance on tabular data, particularly in comparison to deep learning models.

For all regression tasks, XGBoost is configured with the `reg:squarederror` objective, which minimizes squared error loss. Final performance is reported as RMSE, the square root of this loss (see Section 2.3.2).

For classification, XGBoost uses the `multi:softprob` objective, which outputs class probabilities across all classes. Predictions are evaluated using top-k accuracy and precision/recall metrics (see Section 2.3).

Additional implementation details are determined by the hyperparameters (see Section 3.5.1).

3.5.1 Hyperparameter Selection

For models using XGBoost, we apply hyperparameter optimization (HPO) to improve generalization, particularly in settings with high execution time variability such as parallel SpMV.

We perform randomized hyperparameter search using `RandomizedSearchCV` from `scikit-learn`, with 5-fold cross-validation and the negative mean squared error as the scoring metric. The parameter search space includes:

- `n_estimators`: {100, 300, 500}

- `learning_rate`: $\{0.01, 0.05, 0.1\}$
- `max_depth`: $\{3, 6, 9\}$
- `min_child_weight`: $\{1, 3, 5\}$
- `subsample`: $\{0.7, 0.8, 0.9\}$
- `colsample_bytree`: $\{0.7, 0.8, 1.0\}$
- `gamma`: $\{0, 0.1, 0.2\}$
- `reg_alpha`: $\{0, 0.1, 0.5\}$
- `reg_lambda`: $\{0.5, 1, 1.5\}$

The search evaluates 50 randomly sampled hyperparameter configurations. Models are trained on 90% of the data and evaluated on the remaining 10%, with a fixed random seed to ensure reproducibility. The final configuration is selected based on the best average cross-validation score.

Hyperparameter tuning was primarily applied to models predicting parallel execution time, where default configurations showed greater instability and underperformance. For single-thread models, tuning yielded negligible improvements and is omitted in those results.

Chapter 4

Experimental Setup

We explain the experimental setup.

4.1 Matrix Selection

Our experiments are based on a set of 490 base matrices originally selected by Trotter *et al.* [43] from the SuiteSparse Matrix Collection [8]. Each matrix is accompanied by reordered variants (e.g., RCM, AMD, HP, GP), with parallel SpMV timing measurements available for multiple hardware targets via Zenodo at record 7821491. That archive does not include the matrices themselves, but lists their SuiteSparse identifiers along with SpMV measurements under each reordering.

All matrices are square, non-complex, and range from 10^6 to 10^9 nonzeros. The dataset spans diverse domains, including circuit simulation, optimization, structural engineering, and biological computation.

Matrices used for feature computing and single thread SpMV execution time measurements were re-used from Trotter *et al.* [43], located in storage on the eX3.

4.1.1 Data Loss

There is a discrepancy in the selection of SuiteSparse matrices, aswell as their reordered variants used for DIESEL feature computation, and corresponding their multi-thread SpMV execution time measurements. A total of four SuiteSparse matrices and their reordered variants are therefore excluded, for a new total of 486. their group and name that identify them on SuiteSparse are found in Table 4.1. It should be noted that the single-thread contexts, one of these four are available, also shown in aforementioned table.

	Group	**Matrix**
1	GenBank	kmer_V1r
2	MAWI	mawi_201512020130
3	MAWI	mawi_201512020330
4^*	AG-Monien	debr

Table 4.1: Matrices excluded after merging features with timing data. *Dropped only from the multi-thread dataset due to missing timing data.

4.2 Hardware

Access to diverse hardware enables analysis of SpMV execution time that extends beyond intrinsic matrix properties. In this work, we evaluate performance across a wide range of multi-core CPU architectures, including both ARM and x86-64 systems, as summarized in Table 4.2. This table is adapted from Trotter *et al.* [43], from which the n-thread SpMV execution time measurements are also sourced.

All new single thread measurements are conducted in a strictly single-thread configuration using one core per run. These supplement previously published parallel SpMV measurements from Trotter *et al.* [43], which were collected using multi-thread execution with OpenMP on the same set of matrices and hardware platforms. For regression tasks evaluating cross-platform generalization, models are trained and evaluated independently per CPU architecture.

Table 4.2: Hardware used in our experiments. Adapted from Trotter *et al.* [43].

	Skylake	Ice Lake	Naples	Rome	Milan A	Milan B	TX2	Hi1620
CPUs	Intel Xeon Gold 6130	Intel Xeon Platinum 8360Y	AMD Epyc 7601	AMD Epyc 7302P	AMD Epyc 7413	AMD Epyc 7763	Cavium TX2 CN9980	HiSilicon Kunpeng 920-6426
Instr. set	x86-64	x86-64	x86-64	x86-64	x86-64	x86-64	ARMv8.1	ARMv8.2
Microarch.	Skylake	Ice Lake	Zen	Zen 2	Zen 3	Zen 3	Vulcan	TaiShan v110
Sockets	2	2	2	1	2	2	2	2
Cores	2×16	2×36	2×32	1×16	2×24	2×32	2×32	2×64
Freq. [GHz]	1.9–3.6	2.4–3.5	2.7–3.2	1.5–3.3	2.5–3.5	2.5–3.5	2.0–2.5	2.6
L1I/core [KiB]	32	32	64	32	32	32	32	64
L1D/core [KiB]	32	48	32	32	32	32	32	64
L2/core [KiB]	1024	1280	512	512	512	512	256	512
L3/socket [MiB]	22	54	64	64	32	64	32	64
Bandwidth [GB/s]	256	409.6	342	204.8	409.6	409.6	342	342

4.3 Measurement

To measure SpMV execution time for a given matrix, we perform 100 consecutive measurements and retain the fastest, ensuring a warm cache and minimizing variability. This emulates optimal conditions.

All measurements are based on the CSR SpMV kernel provided in the `ellspmv` repository by Trotter *et al.* [43], available at `https://github.com/jamtrott/ellspmv`. The code is compiled with GCC 11.4.0 using `-O3` and `-march=native` optimization flags.

Throughout this thesis we distinguish between single-thread and n-thread SpMV, the former refers to using exclusively one thread on the CPU uses the *1D* SpMV threading strategy, which partitions the matrix by rows and assigns one block per thread. This is the simpler of two common parallelization strategies; the other is the *2D* strategy, which partitions the nonzeros more evenly across threads to improve load balancing. The experiments and models in this thesis focus exclusively on the former. While the 2D approach has been shown to offer benefits in some cases [43], it is outside the current scope but remains a candidate for future work.

We make single-thread SpMV execution time measurements on the TX2 platform. n-thread measurements are reused from prior work by Trotter *et al.* [43], conducted under similar system conditions on the same TX2 platform, but using all available threads.

Chapter 5

Results and Discussion

In this chapter we share and discuss performance of Regression models predicting on a range of SpMV time target sets, varying by SpMV threading, (see Section 4.3), and CPU architecture (see Section 4.2). We implement the DIESEL feature set (see Section 3.3) for all ML tasks in the thesis. We make an exception by using an augmented DIESEL feature set for modeling the reordered SpMV time prediction, as described in Section 3.1.2. We evaluate the effects of feature preprocessing techniques (Feature Scaling and PCA, see Section 2.2.5) and Hyperparameter optimization (HPO) (see Section 3.5.1) on regression models. We also share and discuss classification model performances on a range of classifications target sets, derived from the aforementioned SpMV time measurements. For each set of SpMV time measurements, we one-hot encode the reordering-algorithm which minimizes the time. the time measurements used for classification targets again vary by threading and architecture.

5.1 Regression

This section presents and evaluates results of regression models predicting on a range of SpMV time execution time targets, all with DIESEL features. Target sets include single-thread (ST) SpMV times on the TX2 architecture, 64-thread (MT64) SpMV times on the TX2 architecture, and n-thread SpMV times, one set of targets for each architecture explored in this thesis. OLS and Ridge regression algorithms are evaluated as baselines for the also evaluated XGBoost algorithm applied for regression. We also compare the effects of feature preprocessing and HPO on model performance.

We first show ST SpMV time prediction results for OLS and Ridge, with and without feature preprocessing (Feature scaling and PCA, see Section 2.2.5). Next, we show ST SpMV time prediction results for XGBoost, also with and without feature preprocessing. We then show MT64 SpMV time prediction results for XGBoost, with and without HPO. Then we show reordered SpMV execution time prediction results for XGBoost, comparing models trained on ST and MT64 based measurement targets. Finally, we assess XGBoost prediction performance on a range of n-threaded SpMV time targets, varying by architecture measurement origin.

5.1.1 Linear Regression

We evaluate OLS and Ridge regression for predicting TX2 architecture ST SpMV times using DIESEL features. These serve as interpretable baselines for later comparisons with more complex models.

Least Squares

OLS regression is applied to the TX2 architecture ST SpMV times and DIESEL features. Performance and limitations are discussed below.

Table 5.1 summarizes the evaluation metrics obtained from 5-fold cross-validation and the test set.

Results indicated that feature preprocessing had no measurable effect, and the negligible discrepancy between cross-validation and test set scores suggests the model is stable. RMSE (see Section 2.3.2) is approximately 3.6 times larger than MAE (see Section 2.3.1), suggesting that while the average prediction error is low, a number of notable outliers may be present. Both errors are substantial, with NRMSE and NMAE at 112% and 32%, respectively. The R^2 scores (see Section 2.3.3) are high across both cross-validation and test sets—0.89 and 0.90, respectively—indicating that the model explains a substantial proportion of the variance in SpMV execution times. This reinforces the conclusion that, despite its simplicity, the linear model captures strong structural correlations in the data.

Table 5.1: Cross-validation and test set results for linear regression predicting SpMV execution times on the TX2 architecture using DIESEL features. Preprocessing method (Default, Scaled, PCA) had no impact on the results. Mean target time is 0.1113 seconds.

Set	RMSE	MAE	R^2
Cross-Validation (fold means)	0.1474	0.0366	0.8961
Test Set	0.1250	0.0342	0.9077

Ridge

Ridge regression is applied to the TX2 architecture ST SpMV times and DIESEL features. Performance and limitations are discussed below.

Table 5.2 summarizes the evaluation metrics obtained from 5-fold cross-validation and the test set.

Across a range of regularization strengths (λ), performance remains effectively indistinguishable from OLS. This suggests that regularization does not meaningfully impact predictive accuracy for this dataset and feature set.

Overall, Ridge does not provide meaningful improvements over the standard linear model, reinforcing the view that more flexible models are needed to improve accuracy further.

5.1.2 XGBoost

We show and discuss results for XGBoost regression for predicting on ST- and MT64 SpMV time targets. In this subsection all of the time targets originate from measurements on the TX2 architecture. All models use DIESEL features, though one setup implements augmented DIESEL featured, as described in Section 3.1.2.

Single-Thread Execution: Effect of Feature Preprocessing

XGBoost regression is applied to the ST SpMV time target and DIESEL features, with and without feature processing. Performance and limitations are discussed below.

Table 5.3 shows the evaluation metrics for model predictions on the test set.

Table 5.2: Cross-validation and test set results for Ridge Regression on TX2 architecture using DIESEL features. Multiple λ values were tested, but none significantly improved over Linear Regression or XGBoost.

lambda	**Preprocessing**	**Mean RMSE**	**Mean MAE**	**Mean R^2**
0.1	Default	0.1475	0.0366	0.8960
0.1	Scaled	0.1474	0.0366	0.8962
0.1	PCA	0.1474	0.0366	0.8962
1.0	Default	0.1479	0.0372	0.8954
1.0	Scaled	0.1481	0.0372	0.8957
1.0	PCA	0.1481	0.0372	0.8957
10.0	Default	0.1481	0.0377	0.8950
10.0	Scaled	0.1541	0.0400	0.8878
10.0	PCA	0.1541	0.0400	0.8878
		Test Set Metrics		
	RMSE	**MAE**	**RMSE/avg time**	**R^2**
Ridge (λ=0.1)	0.1251	0.0341	1.124	0.9076
Ridge (λ=1.0)	0.1226	0.0343	1.101	0.9113
Ridge (λ=10.0)	0.1257	0.0346	1.129	0.9068

We find that neither of the preprocessing technique improves model precision significantly. As such, Further evaluation metric discussion implicitly is for the non feature-preprocessing model. Remaining XGBoost models presented also abandon feature preprocessing.

During cross-validation, the R^2 score span across the folds is (0.86, 0.98). This indicates model stability, consistently explaining the variance well in all folds.

NRMSE and NMAE on the test set are 86.2% and 9.8%, respectively. This reflects a considerable improvement in both average and worst-case prediction error over linear models (112% and 32%). The R^2 score also improve from 0.91 to 0.95.

The RMSE is about 8.44x larger than the MAE. This ratio is roughly double that of the linear models, still suggesting that while the average prediction error is low, a number of notable outliers may be present.

Table 5.3: Test set evaluation metrics for XGBoost models predicting TX2 architecture single-thread SpMV execution times, using DIESEL features. Three preprocessing methods are implemented: no preprocessing (Default), feature scaling (Scaled), and PCA. RMSEs and MAEs are shown normalized to the mean target time which is 0.1113s.

Method	**NRMSE**	**NMAE**	**R^2**
Default	0.8616	0.0979	0.9457
Scaled	0.8616	0.0997	0.9457
PCA	1.104	0.1437	0.9109

64-Thread Execution: Effect of Hyperparameter Optimization

XGBoost regression is applied to the MT64 SpMV time target and DIESEL features, with and without hyperparameter optimization. Performance and limitations are discussed below.

Table 5.4 shows evaluation metric results on the test set predictions.

For the model implemented without HPO, the R^2 score span across the cross-validation folds is $(-0.03,\ 0.84)$. This indicates substantial model instability during training, where the model performed slightly worse than always predicting the mean at its worst, and also explaining a great amount of the variance at its best, depending on what data was seen during training.

The test set NRMSE and NMAE are 145.3% and 34.9%, Respectively. This shows a significant degradation relative to predictions on serial execution times on otherwise identical setup, (86.2% and 9.8%). The R2-score similarly drops relative to the serial execution time, from 0.95 to 0.85.

This degradation is not unexpected. Threading SpMV introduces additional system-level variability such as thread scheduling effects, load imbalance, and non-deterministic memory contention that are not reflected in the matrix-level features used.

Despite the increased error, the model retains a meaningful degree of generalization.

The RMSE is $\tilde{3}$.8x larger than the MAE, again suggesting that while the average prediction error is low, a number of notable outliers may be present.

HPO significantly improved predictive performance for predicting MT64 SpMV times. Whereas tuning had little effect on ST time predictor models, it proved beneficial for the MT64 time predictor, reducing NRMSE and NMAE to 115.1% and 30.2%, corresponding to a reduction of 20.8% and 13.3%, respectively. R^2 also from 0.85 to 0.91. This suggests that the added variability and complexity in MT64 SpMV execution makes the model more sensitive to its internal configuration, and thus more responsive to tuning. In contrast, the more predictable single-thread prediction task was well-modeled even with default settings.

Table 5.4: Test set evaluation metrics for two XGBoost models predicting TX2 architecture MT64 SpMV execution times, using DIESEL features, one of which has undergone hyperparameter optimization. RMSE and MAE are shown normalized to the mean target time which is 0.0086s.

Model	**NRMSE**	**NMAE**	R^2
XGBoost (Default)	1.4534	0.3488	0.8473
XGBoost (Tuned)	1.1511	0.3023	0.9053

Reordered SpMV Execution

XGBoost regression is applied to the SpMV times targets and augmented DIESEL features. The targets vary by threading, i.e. ST and MT64. Performance and limitations are discussed below.

Table 5.5 shows evaluation metric results on the test set predictions produced by the models.

For the ST predicting model, the R^2 score span across the cross-validation folds is $(0.87,\ 0.91)$. The small span again indicates model stability.

For the MT64 predicting model, the R^2 score span across the cross-validation folds is $(-0.41,\ 0.89)$. This indicates great model instability during training, where the model performed significantly worse than always predicting the mean at its worst, and again explaining a great amount of the variance at its best, depending on what data was seen during training.

For the ST predicting model, NRMSE is 51.42%, and NMAE is 7.4%, reflecting significant improvements in accuracy from Section 5.1.2. The RMSE is approximately 6.97 times large than the MAE, this again suggests that the model produces a number of large outliers. The R^2 score is remarkably high, indicating that the model captures a large proportion of execution time variance.

For MT64 model predictions, the NRMSE and MAE are 228.5% and 44.64%, respectively. This is the worst result so far, and again shows significant degradation of model accuracy in MT64 SpMV time predictions, compared to ST SpMV time predictions.

Table 5.5: Test set evaluation metrics for two XGBoost models predicting TX2 architecture SpMV execution times, using augmented DIESEL features. One of the target set measurements implements ST SpMV, the other implements MT64 SpMV.

Model	RMSE	MAE	R^2	Mean Predicted Time
XGBoost (ST)	0.0342	0.0049	0.9493	0.0665
XGBoost (MT64)	0.0128	0.0025	0.7636	0.0056

5.1.3 Cross-Architecture Regression-Performance Stability Assessment

We show, compare, and discuss results for XGBoost regression for predicting n-thread SpMV time targets, measurements originating from the varying CPU architectures described in Tab 4.2. All models used DIESEL features, and each one implements HPO.

Cross-Validation

Figure 5.1 shows a box plot of R^2 scores across cross-validation folds for the n-thread SpMV execution time predictors, grouped by CPU architecture. We observe that the median R^2 scores are above 0.6 for all architectures except Hi1620, which exhibits the lowest performance: all folds fall below 0.4, with one outlier reaching approximately 0.6.

The variability across folds differs by architecture. TX2, Naples, and particularly Rome exhibit tight distributions, indicating stable model performance during training. In contrast, Milan A, Milan B, and Skylake show broader spreads, suggesting higher fold-to-fold variability. Ice Lake demonstrates the widest distribution, spanning nearly 0.4 in absolute R^2 score between quartiles.

Overall, the figure suggests that the model achieves reasonably consistent performance across architectures, with stability varying by architecture. Most architectures demonstrate acceptable fold-level stability, though some, especially Hi1620, show signs of weaker generalization.

R2

Figure 5.2 shows the R^2 scores on the test sets for the n-thread SpMV execution time predictors, evaluated separately for each CPU architecture. Most architectures achieve scores above 0.8, indicating strong generalization. Notable exceptions include Naples and Milan B, both near 0.6, and Milan A, which falls further to approximately 0.45.

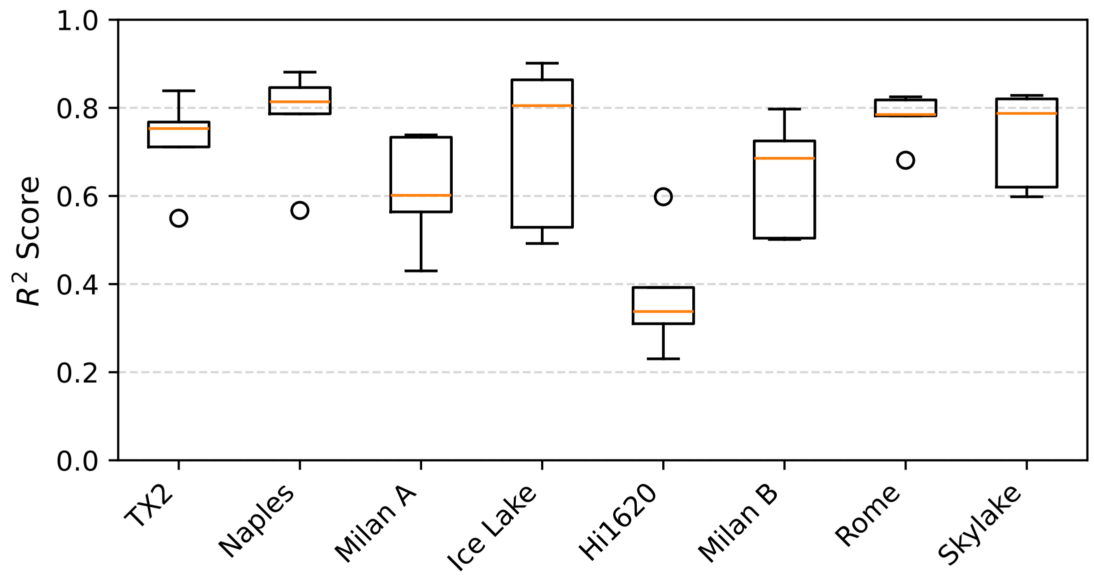

Figure 5.1: Box plot of R^2 scores across cross-validation folds for n-thread SpMV execution time prediction using DIESEL features, grouped by CPU architectures. Most architectures exhibit median R^2 scores above 0.6 with relatively tight fold-to-fold variability, indicating stable model performance. Exceptions include Ice Lake and Hi1620, which show wider distributions and lower predictive accuracy, respectively.

These results suggest that while the predictors generalize well on most architectures, their explanatory power is not uniformly reliable. The performance drop on Milan A in particular highlights a case of limited generalization capacity. Overall, the test scores confirm that regression-based modeling of parallel SpMV remains viable, but can vary considerably across architectures.

NRMSE and NMAE

Figures 5.3, 5.4, and 5.5 present the NRMSEs, NMAEs, and NRMSE-to-NMAE ratios on each CPU architecture. We observe that NRMSE remains relatively stable across most architectures, ranging from just below 0.9 on Rome to approximately 1.5 on TX2 and Hi1620. Outliers include Naples (just below 2.25), Milan B (around 2.75), and Milan A (peaking at roughly 3.25). W find that the NMAE figure follows a similar shape, with Milan A again showing the highest normalized error (nearly 0.6), followed by Naples (0.5) and Milan B (0.45).

The NRMSE-to-NMAE ratios remain remarkably consistent across most architectures, clustered between 3.0 and 4.0. Exceptions include Milan A and Milan B, which exhibit higher ratios of approximately 5.5 and 6.0, respectively, suggesting larger outlier errors relative to average deviations. These results indicate that predictive performance is generally stable across architectures, with a few notable exceptions.

We also observe a clear correspondence between higher R^2 scores and lower NRMSEs and NMAEs, as expected. This reinforces the consistency of the different evaluation metrics in capturing prediction quality.

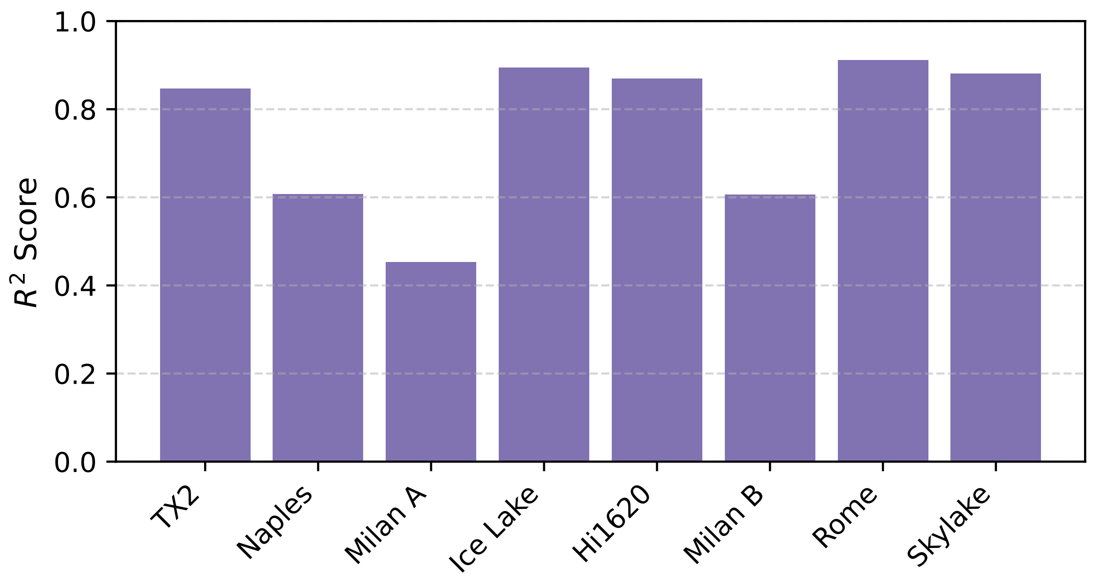

Figure 5.2: R^2 scores on the test sets for parallel SpMV execution time prediction using DIESEL features, evaluated per CPU architecture. Most architectures achieve strong generalization with scores above 0.8. Lower performance is observed on Naples, Milan B (both near 0.6), and Milan A (approximately 0.45), indicating variation in model effectiveness across architectures.

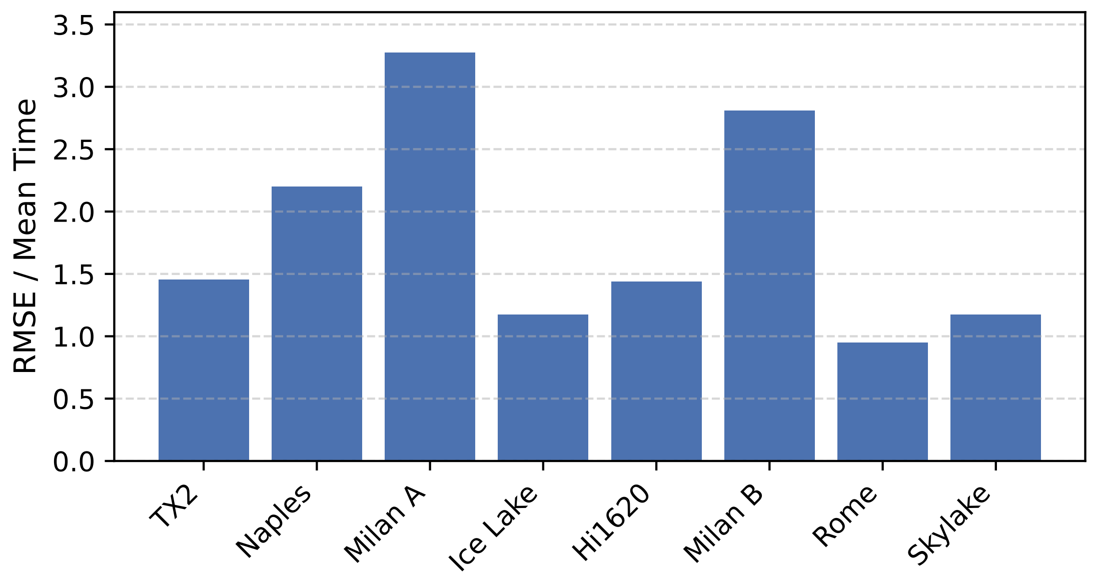

Figure 5.3: Normalized RMSE of parallel SpMV execution time prediction across CPU architectures. Values are normalized by the mean execution time per architecture. Most architectures fall below 1.5, with outliers including Naples, Milan B, and Milan A, the latter peaking above 3.0.

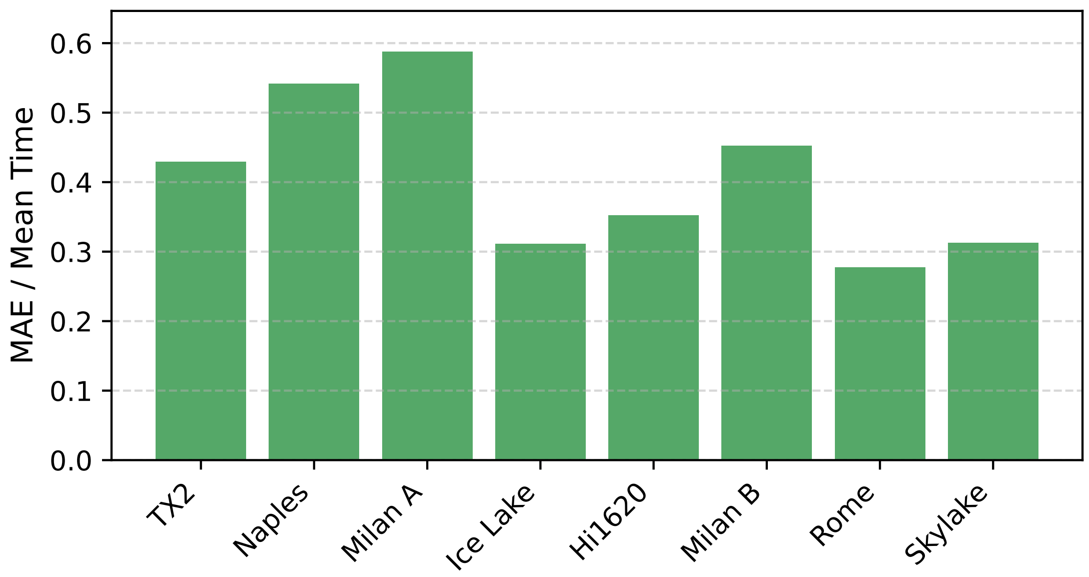

Figure 5.4: Normalized MAE of parallel SpMV execution time prediction across CPU architectures. Milan A exhibits the highest average error relative to mean time, followed by Naples and Milan B. Most architectures remain below 0.4.

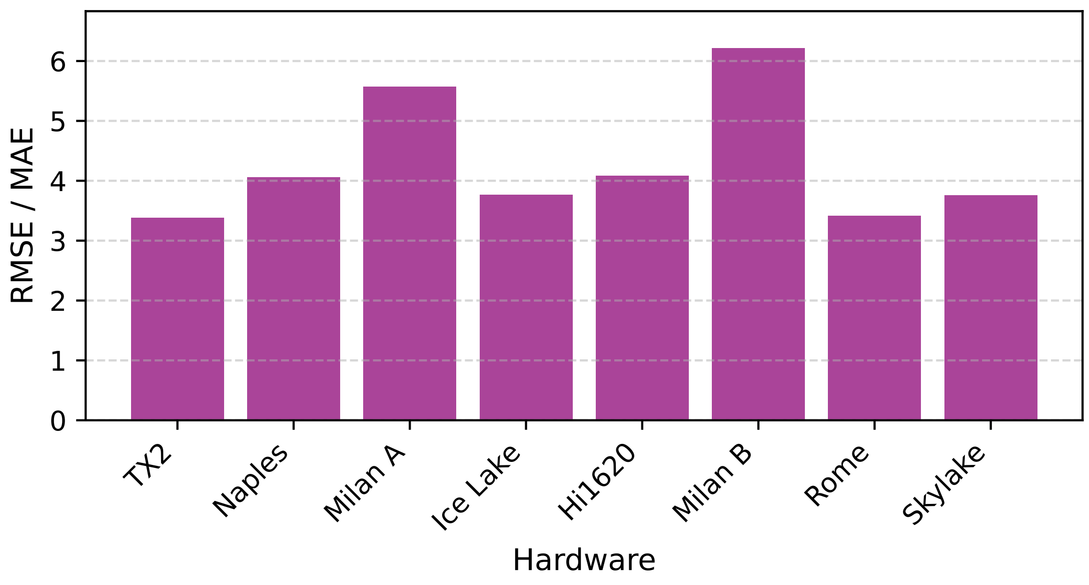

Figure 5.5: Ratio of RMSE to MAE for parallel SpMV execution time prediction across CPU architectures. Most values lie between 3.0 and 4.0, indicating consistent error spread. Higher ratios for Milan A and Milan B suggest heavier-tailed error distributions on those architectures.

5.1.4 Summary

Regression-based modeling of SpMV execution time using DIESEL features proves broadly effective across single-thread, n-thread, and reordered settings. Linear models (OLS and Ridge) offer strong baselines with interpretable structure and stable performance, but lack the flexibility to capture nonlinear patterns. XGBoost consistently outperforms linear models in single-thread context, where it achieves high R^2 and low error. However, model performance degrades under multi-threaded execution due to increased system-level variability not captured by matrix features. Hyperparameter optimization improves robustness in this setting, particularly for MT64 targets. Cross-architecture experiments show that generalization is achievable, with most models maintaining R^2 above 0.6 and moderate error spread. Still, architectures like Milan A and Hi1620 highlight the limits of transferability. Overall, regression methods demonstrate strong potential for SpMV performance prediction, especially when matched to predictable execution environments and supported by tuning.

5.2 XGBoost Classification

As an alternative to regression modeling, we also evaluate a classification-based approach: predicting which reordering scheme will yield the best performance for a given matrix. This sidesteps direct modeling of runtime and instead treats the reordering selection as a categorical decision task. We evaluate the classifier's ability to generalize from DIESEL features computed on the original matrix to reorderings that minimize execution time, comparing its predictions to baseline strategies. Both single-thread (ST) and n-thread contexts are examined, with the latter expected to exhibit higher variance and greater misclassification cost.

5.2.1 Single-Thread Execution

We evaluate an XGBoost classifier for predicting reorderings minimizing the TX2 architecture single-thread SpMV execution times, using DIESEL features. Two baseline models are included: always applying RCM, and never reordering.

The cross-validation accuracy is $51.59 \pm 6.62\%$. we find this shows some instability during training.

Table 5.6 shows model evaluation metrics for the classifier, as well as the baseline models of always implementing the RCM reordering algorithm, and never implementing any reordering algorithm. The classifier consistently outperforms the RCM baseline model. Both significantly beat the non-reordering baseline, operating within 2.5% of perfect optimization-wise. Notably, the baseline model that always selects RCM achieves a relative total runtime (RTT) of 102.48%, despite being optimal in only 20 out of 49 cases. This indicates that in the remaining 29 cases, RCM often yields near-optimal performance even when not the best, resulting in low regret. The model's robustness makes it a strong default, but the classifier still offers modest accuracy gains and consistently higher top-k performance, improving likelihood of hitting or nearing the optimal choice.

Table 5.7 show the per-class diagnostic metrics. We find that they show the model learns to detect RCM as optimal reasonably well. It also tries and fails to predict the remaining classes, but fails to identify them reliably, suggesting weak class separation rather than total neglect.

Regret analysis in the serial setting reveals that the classifier operates very close to optimal on average, with a mean regret of 1.0120 and median of 1.0000. This suggests that even when the model misses, the performance degradation is minimal. The highest observed regret is 1.14, and the worst five cases show misclassifications such as predicting RCM instead of ND, or Gray instead of Original. These represent edge cases rather than systemic failure. Given the low overall variance, regret analysis offers limited diagnostic insight in this setting but confirms the robustness of the model's output.

Metric	Classifier	Baseline (RCM)	Baseline (Original)
Accuracy	0.5102	0.4082	1.0120
Top-2 Accuracy	0.6939	-	-
Top-3 Accuracy	0.8980	-	-
Rel. Total Time	1.0208	1.0248	1.2262

Table 5.6: Evaluation metrics for XGBoost classifier predicting the reordering algorithm with the fastest serial SpMV execution time on the TX2 architecture. the model uses DIESEL features for each base-matrix, i.e. from SuiteSparse. Also includes two baseline models: always implementing the RCM algorithm, and never reordering.

Class	Precision	Recall	Support
AMD	0.38	0.43	7
Gray	0.50	0.50	6
ND	0.17	0.17	6
Original	0.44	0.40	10
RCM	0.70	0.70	20

Table 5.7: Per class diagnostics for XGBoost classifier predicting the reordering algorithm with the fastest serial SpMV execution time on the TX2 architecture. the model uses DIESEL features for each base-matrix, i.e. from SuiteSparse.

These results compare favorably with prior work on ML-based optimization of SpMV, such as the DIESEL framework [30], which achieved 88.2% classification accuracy in selecting optimal format/kernel combinations. While DIESEL addresses a different task—predicting storage formats and kernels—the setting is conceptually similar: both aim to approximate oracle performance using structural features. Despite achieving lower raw accuracy (51.0%) in the reordering selection task, our classifier yields nearly optimal execution times in most cases, with a mean runtime regret of just 1.012. This mirrors DIESEL's reported workload accuracy of 91.96% and demonstrates that structural features can support effective algorithm selection, even when classification precision is limited.

5.2.2 64-thread Execution

We evaluate an XGBoost classifier for predicting reorderings minimizing the TX2 architecture 64-thread SpMV execution times. The same DIESEL features and baselines are used as in the single-thread target classification setup.

The cross-validation accuracy is $61.57 \pm 5.42\%$. As in the serial case, we find again this shows some instability during training.

Table 5.8 shows model evaluation metrics for the classifier, as well as the baseline models of always implementing the RCM reordering algorithm, and never implementing any reordering algorithm. We find that the classifier more convincingly outperform the baseline models than in the serial case. The RTT is high at about 1.4, showing significant room for improvement. Furthermore, the baseline model performances on parallel prediction reinforce the challenge and significance of reordering algorithms in parallel SpMV execution time prediction.

Table 5.9 presents per-class metrics. RCM remains dominant in class frequency and is learned most reliably. Precision and recall are balanced, indicating consistent recognition. Original is predicted with moderately high recall, showing the model identifies it frequently when optimal, but also over-predicts it. ND shows weak learning: precision and recall remain balanced but low. Gray achieves perfect precision but low recall, reflecting strong hesitation and under-detection. AMD is ignored entirely—zero predictions, highlighting class suppression, likely due to very low support.

Regret analysis in the parallel setting shows higher variance and greater performance penalties from misclassification compared to the serial case. While the median regret remains 1.000, indicating that half of the predictions are effectively optimal, the mean regret increases to 1.0651—over five times higher than in the serial setting. The 90th percentile regret reaches 1.1149, and the worst case exceeds 2.0, meaning the predicted reordering took more than twice as long as the optimal one. These findings underscore that although the model captures general patterns, its errors in parallel SpMV carry significantly greater runtime costs, making misclassifications more consequential.

These findings underscore that although the model captures general patterns, its errors in parallel SpMV carry significantly greater runtime costs, making misclassifications more consequential. Compared to format/kernel selection models such as DIESEL [30], which achieves 88.2% accuracy and minimal performance loss, our classification model operates at a lower raw accuracy (65.3%) but maintains competitive top-2 (85.7%) and top-3 (93.9%) accuracy. While DIESEL and this work address different prediction targets, both aim to approximate oracle performance with limited structural information. The increased runtime variance and regret observed here may reflect the added difficulty of reordering selection under parallel execution, though limitations in the current feature representation cannot be ruled out. Overall, the results demonstrate that data-driven reordering prediction remains viable but challenging in the multicore regime, with accuracy gains translating into meaningful but non-trivial performance improvements.

Metric	Classifier	Baseline (RCM)	Baseline (Original)
Accuracy	0.6531	0.5102	0.2857
Top-2 Accuracy	0.8571	-	-
Top-3 Accuracy	0.9388	-	-
Rel. Total Time	1.4053	1.4278	1.5063

Table 5.8: Evaluation metrics for XGBoost classifier predicting the reordering algorithm with the fastest TX2 architecture MT64 SpMV execution time. Includes two baseline models: always implementing the RCM algorithm, and never applying reordering (i.e., always using the original matrix).

Class	Precision	Recall	Support
AMD	0.00	0.00	2
Gray	1.00	0.33	3
ND	0.33	0.40	5
Original	0.64	0.64	14
RCM	0.71	0.80	25

Table 5.9: Per-class diagnostic metrics for XGBoost classifier predicting the reordering algorithm with the fastest TX2 architecture MT64 SpMV execution time. The model uses DIESEL features computed on each base matrix.

5.2.3 Cross-Architecture Classifer-Performance Stability Assessment

We evaluate eight XGBoost classifiers, one per CPU architecture (see Section 4.2), trained to predict the reordering algorithm that minimizes n-thread SpMV execution time.

All models use DIESEL features derived from the original SuiteSparse matrices. As baselines, we evaluate a model that always selects RCM and another that never applies reordering (i.e., always selects the original matrix structure).

Cross-Validation Performance

Figure 5.6 shows cross-validation performance across CPU architectures using box plots for accuracy and relative total time.

Median accuracies range from approximately 50% to a peak of $\tilde{6}$2% on Skylake, with one notable outlier—Hi1620—showing a median near 41%. Most architectures exhibit considerable spread, with wide inter-quartile ranges. Exceptions include Ice Lake and Rome, which show tighter distributions but also contain one and two outlier folds, respectively. Milan B shows a similar pattern, with one fold achieving an unusually high accuracy.

Median RTTs are consistently low, with most models achieving close to optimal performance (less than 105% of oracle runtime). Naples stands out with a significant outlier above 130%. TX2, Hi1620, Milan B, and Skylake show tight inter-quartile ranges, suggesting stable generalization. However, each also has one fold with noticeably degraded performance—between 115% and 125% of oracle time.

Notably, accuracy and relative runtime do not strongly correlate. Several models with modest accuracy achieve near-optimal runtime, while others with higher accuracy still produce costly misclassifications. This highlights that accuracy alone is not a reliable proxy for runtime performance in this task.

Relative Total Times

Figure 5.7 shows the RTT incurred by the classifier's predictions on each CPU architecture. The XGBoost classifier consistently outperforms the baseline of applying no reordering, with the exception of Skylake, where it performs slightly worse. The improvement margin varies, ranging from substantial (e.g., Naples) to marginal (e.g., Ice Lake).

On several architectures, namely Milan B, Rome, and Skylake—the baseline strategy of always choosing RCM yields lower total runtime than the learned classifier. However,

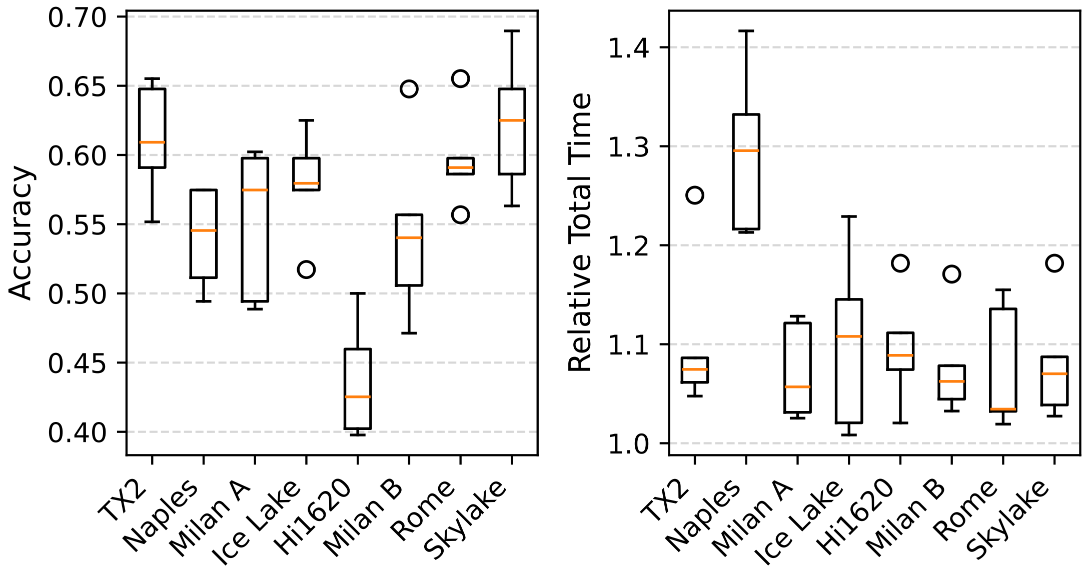

Figure 5.6: Box plots of cross-validation accuracy and relative total time for each CPU architecture. Each box shows the median (orange line), inter-quartile range (box), and potential outliers (points). Accuracy exhibits varying stability across architectures, while relative total time remains consistently low with a few exceptions.

we also observe the limitations of the RCM baseline, particularly on Naples where it performs poorly.

Overall, the XGBoost classifiers outperform the no-reordering baseline on most architectures, and outperform the RCM baseline on a smaller subset. These results suggest that data-driven reordering decisions offer a runtime advantage on average, but may under perform strong hand-crafted heuristics in specific contexts.

Across all eight CPU architectures this corresponds to a mean speed-up of 1.10× over leaving the matrix unordered, ranging from 0.98× on Skylake (a modest 2% slowdown) up to 1.26× on Rome, while still remaining roughly 1.16× the oracle time on average.

Previous auto-tuning studies report higher prediction accuracy but operate in broader design spaces than row/column permutations. SMAT [27] and DIESEL [30] achieve 88–92% top-1 accuracy and typically run within 5–10% of oracle performance, delivering speedups of 3–4× over vendor libraries such as MKL. WISE [44] attains an average speedup of 2.4× over MKL, closely approaching the oracle's 2.5× performance. Cerberus [21] achieves 83.3% prediction accuracy and reaches 90% of oracle performance. In contrast, our selector, focusing solely on row/column permutations, achieves a mean speedup of 1.10× over the original ordering, with a maximum of 1.26× on Rome, and remains within 1.16× of the oracle on average.

Accuracies

Figure 5.8 shows the Top-1, Top-2, and Top-3 accuracies for the XGBoost classifiers across CPU architectures, alongside two baselines: always selecting RCM and applying no reordering.

Top-1 accuracy remains consistently between 55–60% for most architectures, with Rome slightly exceeding 60% and Skylake peaking above 70%. A loose correlation is observed between Top-1 accuracy and the RCM baseline, with Rome standing out

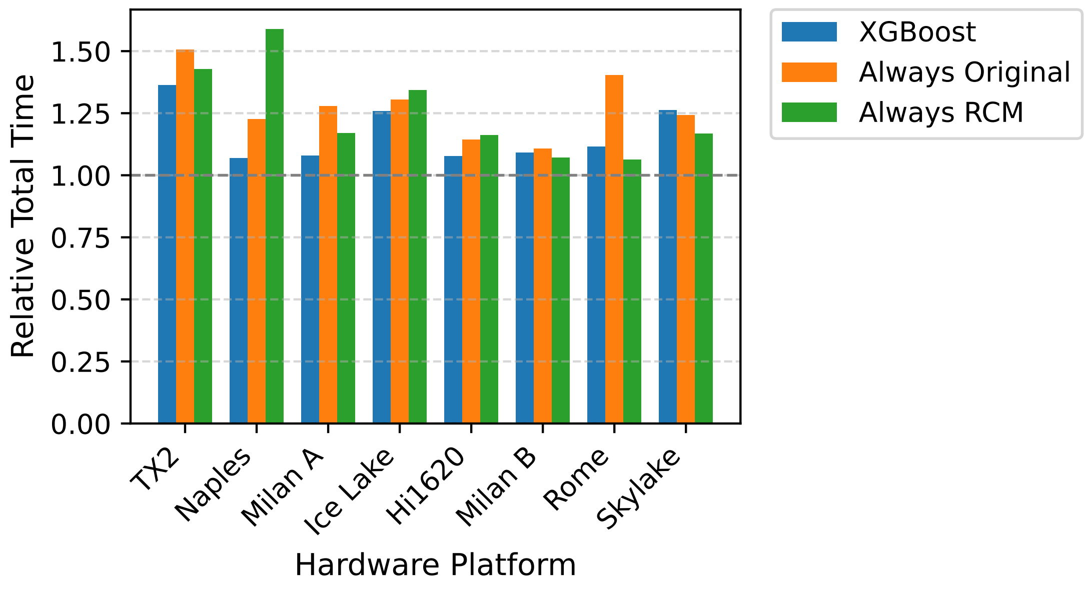

Figure 5.7: RTT for XGBoost classifiers predicting the reordering algorithm with the fastest n-thread SpMV runtime, for each CPU architecture. We compare XGBoost classifiers with two baselines: the first is always choosing RCM, the second is never applying reordering.

as an exception—showing relatively high classifier accuracy despite moderate RCM performance.

Interestingly, Skylake exhibits the highest classifier accuracy while also being the only architecture where the non-reordering baseline achieves lower total runtime than the classifier. This suggests the classifier may over-predict suboptimal reorderings despite being accurate in classification terms.

Top-2 and Top-3 accuracies are consistently high across architectures, indicating that the true optimal reordering is often among the top predicted candidates. This highlights the classifier's utility in narrowing down viable reordering options.

Hi1620 shows the lowest performance across all metrics. This suggests that RCM and the original ordering are rarely optimal on this architecture, and that the classifier may over-predict these classes, leading to degraded accuracy and runtime performance.

5.2.4 Confusion Matrices

To supplement the classification analysis, we compare confusion matrices from two classifiers trained on different architectures: TX2 and Rome. Figures 5.9 and 5.10 visualize the test set predictions for each classifier.

For TX2, the confusion matrix reveals a heavy class imbalance in the ground truth labels: RCM and no reordering are optimal in the majority of cases (25 and 20 times, respectively), while AMD is optimal only twice. The classifier reflects this imbalance by making 47 out of 49 predictions as either RCM or Original. Only two predictions are made on ND (one correct), and none on AMD or Gray. This shows a strong model bias toward the majority classes, with minimal learning of the minority classes.

In contrast, the Rome dataset exhibits a more balanced class distribution among ND, Original, and RCM, although AMD and Gray remain rare. The Rome classifier predicts a broader spread of classes, including occasional predictions on AMD and Gray. This suggests that given more representative training data, the model can learn a wider range

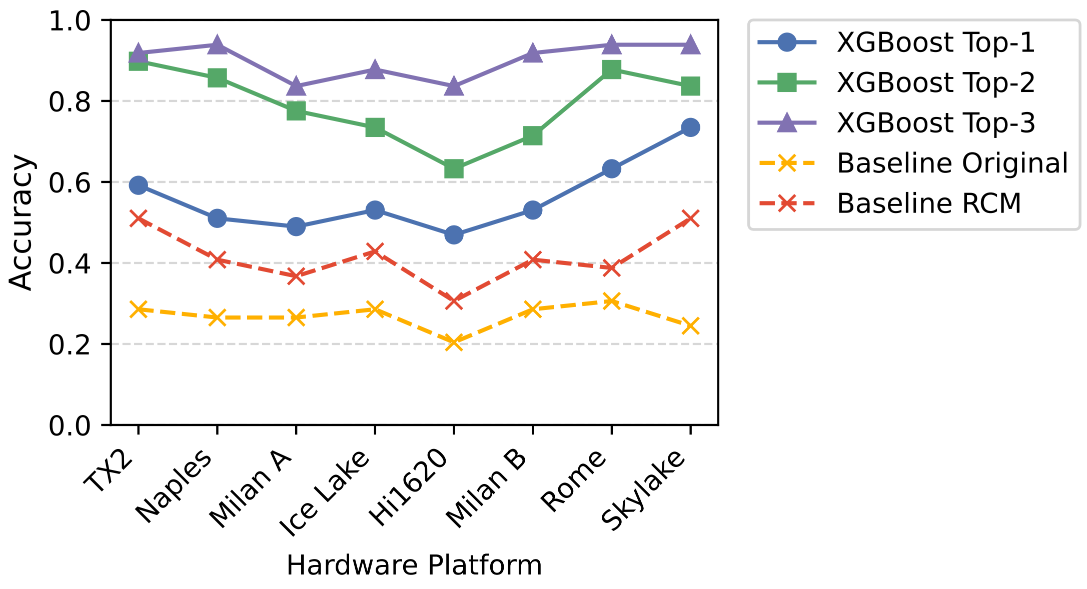

Figure 5.8: Top-1, Top-2, and Top-3 classification accuracy across CPU architectures. Also shown are baseline accuracies from always selecting the original ordering or RCM. While Top-1 accuracy varies, Top-2 and Top-3 remain consistently high. This demonstrates that the classifier often ranks the optimal scheme among its top predictions.

of class distinctions. Despite this, the Rome classifier achieves a lower accuracy (by about 0.2) than the TX2 classifier. This implies that while class diversity improves prediction breadth, it does not necessarily improve top-1 accuracy, though it may contribute to improved performance in top-k metrics or runtime regret.

Overall, these results suggest that confusion matrices are a valuable diagnostic tool for understanding classifier behavior and limitations. In particular, they highlight the importance of class support in the training set and show that even moderate accuracy can mask substantial differences in class prediction behavior.

5.2.5 Summary

The classification approach offers a viable alternative to direct runtime prediction by learning to identify high-performing reorderings from structural features. While overall classification accuracy remains moderate (50–65%), runtime outcomes are often close to optimal—particularly in the single-thread case. Top-3 accuracy is consistently high, indicating that the true optimal choice is frequently among the model's top predictions. In parallel settings, increased variability and class imbalance amplify the cost of misclassifications, underscoring the importance of regret-based evaluation. Across architectures, confusion matrices reveal systematic biases tied to training distribution, but also show that broader class support improves prediction diversity. Taken together, these results demonstrate that XGBoost-based reordering selection is effective in principle, with performance constrained primarily by limited class representation and structural ambiguity. Further gains likely require richer features and balanced data.

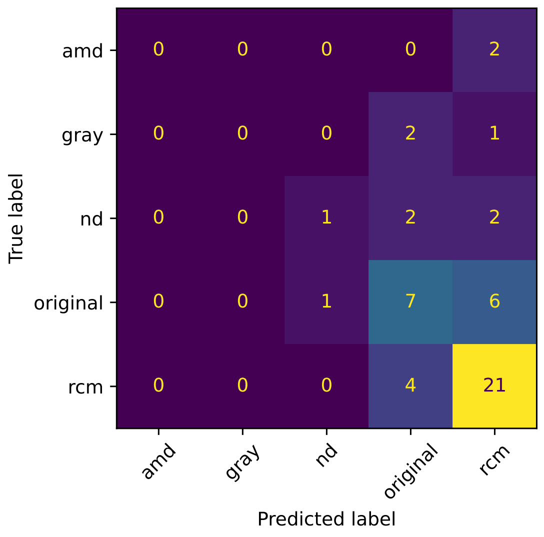

Figure 5.9: Confusion matrix for the test set of the classifier predicting the best reordering algorithm on TX2 architecture 64-thread SpMV execution times.

5.3 Feature Scaling and PCA

Tables 5.1, 5.2, and 5.3 indicate that neither of the preprocessing techniques (see Section 2.2.5) provide meaningful benefits across the evaluated machine learning models. Least squares does not exhibit any improvement from feature scaling or PCA. Ridge, while consistently benefiting from feature scaling across all tested λ values, shows no additional gains from PCA beyond what is already achieved by scaling. XGBoost, on the other hand, remains unaffected by feature scaling and is actively worsened by PCA, with performance degrading by approximately 28%. These findings are based on direct matrix-wise execution time prediction. Although the reordered SpMV execution time formulation with reordering as a categorical feature was not tested with PCA, the consistent lack of benefit observed across all models suggests similar outcomes would apply. We map the same reasoning to classification tasks aswell.

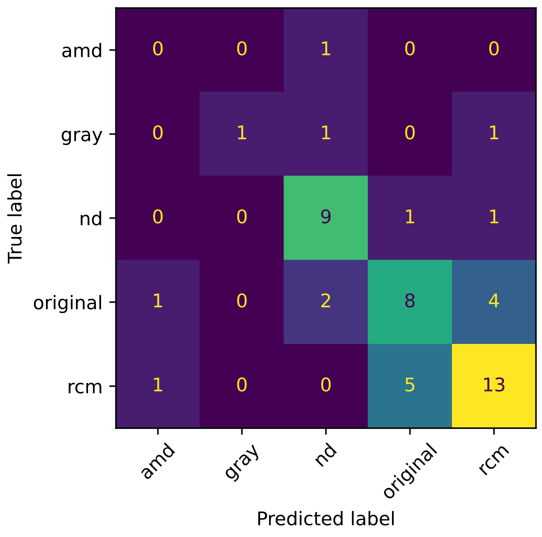

Figure 5.10: Confusion matrix for the test set of the classifier predicting the best reordering algorithm on Rome architecture n-thread SpMV execution times.

Chapter 6

Conclusion

In this chapter we summarize the thesis work and our findings. We review the methodology, present the core results and their interpretation, reflect on limitations, and suggest directions for future work.

We evaluate a dataset of 486 matrices from the SuiteSparse collection [8], including several reordering variants per matrix. Single-thread (ST) SpMV execution times are measured across multiple hardware platforms, and n-thread SpMV execution times are drawn from prior work by Trotter et al. [43]. DIESEL features are extracted for each matrix to enable learning.

6.1 Regression Tasks and Results

6.1.1 ST SpMV Time Prediction

The first regression task involves predicting SpMV execution time for each matrix instance—original or reordered—using its DIESEL features.

We implement OLS, Ridge, and XGBoost models. OLS and Ridge serve as baselines in the ST setting, both achieving $R^2 \approx 0.9$, with normalized RMSE and MAE around 112% and 32%, respectively. XGBoost outperforms both, reaching $R^2 = 0.95$, with NRMSE and NMAE of 86.2% and 9.8%.

Feature preprocessing (e.g., normalization and PCA) yields negligible benefit or slightly worsens performance, particularly when applied to XGBoost. Hyperparameter optimization offers limited gains in this formulation.

6.1.2 Multi-Thread (CPU) SpMV Time Prediction

For the MT64 regime, only XGBoost is evaluated. Performance degrades compared to the ST case, with $R^2 = 0.85$, and NRMSE and NMAE increasing to 145.3% and 34.9%, respectively. Feature preprocessing again provides no benefit. However, hyperparameter optimization significantly improves performance, reducing NRMSE and NMAE to 115.1% and 30.2%, respectively, and increasing R^2 to 0.91.

6.1.3 Base-Matrix Regression Formulation

We also evaluate a formulation where each matrix is represented only by DIESEL features of its original form, with the reordering algorithm encoded as a categorical input. This enables inference on unseen reorderings without requiring feature recomputation.

For ST execution time, this model achieves $R^2 = 0.95$, with NRMSE and NMAE of 51.4% and 7.4%, representing strong predictive accuracy. For 64-thread execution time, performance drops to $R^2 = 0.76$, with NRMSE and NMAE rising to 228.5% and 44.6%. These results highlight the increased difficulty of predicting multi-threaded performance from static structure alone.

6.1.4 Cross-Architecture Generalization

We also evaluated XGBoost regression models trained and tested per CPU architecture using DIESEL features. These models achieved robust performance on most architectures, with median R^2 scores above 0.6 and several exceeding 0.8. Error metrics (NRMSE and NMAE) remained moderate across platforms, though generalization degraded on certain hardware (e.g., Milan A, Hi1620). These findings indicate that structural matrix features can support meaningful performance modeling across hardware targets, particularly when paired with hyperparameter tuning.

6.2 Classification Task and Results

We train XGBoost classifiers on DIESEL features of the original matrices to predict the reordering algorithm that yields the fastest SpMV execution time.

In the ST regime, classification accuracy is moderate (51.0%), but relative total runtime is close to optimal. The model consistently outperforms both baseline strategies and achieves high top-3 accuracy (89.8%), often identifying near-optimal reorderings. Runtime regret is low, with most misclassifications resulting in minimal slowdown.

In the n-thread regime, classification accuracy increases (up to 65.3% depending on architecture), and relative total runtime improves consistently over the no-reordering baseline and, in many cases, the RCM baseline. Top-3 accuracy remains high across all hardware (often above 90%), indicating robustness. These results demonstrate the feasibility of learning-based reordering selection, particularly when narrowing down to a few strong candidates is sufficient.

6.3 Limitations

The classification models are limited by class imbalance. Rare classes (e.g., AMD and Gray) occur infrequently and are difficult to learn. Some classes had to be excluded due to insufficient support. Expanding the dataset or applying balancing techniques may improve support for underrepresented reorderings.

In the n-thread regime, classification and regression models both show degraded performance compared to ST, reflecting the difficulty of modeling n-thread runtime variability. Static matrix-level features may be insufficient for capturing system-level effects such as memory bandwidth, cache contention, and thread scheduling.

All models were evaluated on fixed hardware. Generalization across architectures would require incorporating hardware-level descriptors into the feature set and training on a more diverse dataset.

6.4 Future Work

This thesis demonstrates the feasibility of predicting SpMV execution time and selecting high-performing reordering algorithms using machine learning. Future work could extend this by incorporating more matrices, additional reordering schemes, and a broader range of hardware.

Feature engineering is likely to yield further gains, especially in the multi-threaded regime. Incorporating runtime profiling, cache simulations, or handcrafted metrics may enhance model expressiveness.

Another promising direction is hardware-aware or hardware-agnostic modeling. By encoding hardware characteristics and training across diverse platforms, future models may generalize to unseen or emerging architectures, enabling predictive modeling in early hardware-software co-design.

6.5 Closing Remarks

This work contributes to data-driven performance modeling for sparse numerical kernels, focusing on SpMV and matrix reordering. By evaluating both regression and classification strategies across ST and multi-thread regimes, we demonstrate that structural features can inform performance-critical decisions—even under imperfect modeling conditions.

While regression yields strong predictive accuracy in the ST setting, the classification-based reordering selection offers a lightweight, structure-aware alternative. Despite imperfect classification accuracy, it delivers competitive execution times and provides a viable path toward integration in automated solver pipelines. The consistent performance of classification models in top-k accuracy and runtime regret highlights their practical value in approximate decision-making settings.

Across eight CPU architectures, our classification-based selector achieves a mean speedup of 1.10× over the original ordering (with a peak of 1.26× on Rome), while staying within approximately 1.16× of oracle performance on average. This compares favorably to the difficulty of the reordering task, though it remains modest relative to prior ML-based SpMV optimization systems operating in broader design spaces. SMAT [27] and DIESEL [30] report 88–92% accuracy and 3–4× speedups over vendor libraries. WISE [44] approaches 2.5× oracle speedups using reinforcement learning, and Cerberus [21] achieves 83.3% accuracy and 90% of oracle performance using architecture-aware prediction.

To our knowledge, this thesis presents the first attempt to use supervised machine learning to both predict SpMV execution time and automatically select performance-enhancing reordering schemes, using only matrix-level features. Unlike prior work focusing on storage formats or kernel dispatch, our models target row and column permutations within a fixed CSR representation, showing that this narrower domain still admits meaningful, generalizable performance gains.

Appendix A

Reproducibility Script

Listing A contains the minimal, runnable Python script used to reproduce the single-thread XGBoost regression pipeline described in Section 3.1.3.

```
"""
Minimal example: predict TX2 SpMV times with Diesel features.
Reproduces the single-thread (ST) regression pipeline used in
    Section
"""
import pandas as pd
from xgboost import XGBRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import root_mean_squared_error,
    mean_absolute_error, r2_score
from sklearn.model_selection import KFold


times = pd.read_csv(
     "/home/mads/master_thesis/data/csr-parallel-times/
        csr_all_armq_064_threads_ss490.csv"
)
times.dropna(axis=0, inplace=True)
times.rename(columns={"name": "matrix"}, inplace=True)
time_cols = times.filter(regex=("time*")).columns
times = times.melt(
     id_vars=["group", "matrix"],
     value_vars=time_cols,
     var_name="reordering_scheme",
     value_name="time",
)
times["reordering_scheme"] = times["reordering_scheme"].str.
    removeprefix("time_")
features = pd.read_csv("/home/mads/master_thesis/data/
    diesel_features_all.csv")
features = features.rename(columns={"scheme": "
    reordering_scheme", "name": "matrix"})
devset = features.merge(times, on=["group", "matrix", "
    reordering_scheme"], how="inner")
```

```
y = devset.pop("time")
devset.drop(columns=["group", "matrix", "reordering_scheme"],
    inplace=True)
X = devset.copy()
# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
     X, y, test_size=0.1, random_state=42
)
# Define the model
model_cv = XGBRegressor(
     learning_rate=0.01, n_estimators=500, max_depth=6,
        eval_metric="rmse"
)
rmses_cv = []
maes_cv = []
r2_scores_cv = []
kf = KFold(n_splits=5, shuffle=True, random_state=42)
for train_idx, val_idx in kf.split(X):
     X_train_cv, X_val_cv = X.iloc[train_idx], X.iloc[val_idx]
     y_train_cv, y_val_cv = y.iloc[train_idx], y.iloc[val_idx]
     model_cv.fit(X_train_cv, y_train_cv, eval_set=[(X_val_cv,
        y_val_cv)], verbose=False)
     y_pred_cv = model_cv.predict(X_val_cv)
     rmses_cv.append(root_mean_squared_error(y_val_cv, y_pred_cv
        ))
     maes_cv.append(mean_absolute_error(y_val_cv, y_pred_cv))
     r2_scores_cv.append(r2_score(y_val_cv, y_pred_cv))

model = XGBRegressor(
     learning_rate=0.01, n_estimators=500, max_depth=6,
        eval_metric="rmse"
)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Compute test set metrics
avg_time = y_test.mean()
rmse = root_mean_squared_error(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean␣time␣in␣test␣set:␣{avg_time:.4f}")
print(f"RMSE␣on␣test␣set:␣{rmse:.4f}")
print(f"MAE␣on␣test␣set:␣{mae:.4f}")
print(f"RMSE/avg_time␣on␣test␣set:␣{rmse/avg_time:.4g}")
print(f"MAE/avg_time␣on␣test␣set:␣{mae/avg_time:.4g}")
print(f"R2␣on␣test␣set:␣{r2:.4f}")
```

Bibliography

[1] '1. Introduction'. In: *Direct Methods for Sparse Linear Systems*, pp. 1–6. DOI: 10.1137/1.9780898718881.ch1. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898718881.ch1. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898718881.ch1.

[2] Patrick R. Amestoy, Timothy A. Davis and Iain S. Duff. 'Algorithm 837: AMD, an approximate minimum degree ordering algorithm'. In: *ACM Trans. Math. Softw.* 30.3 (Sept. 2004), pp. 381–388. ISSN: 0098-3500. DOI: 10.1145/1024074.1024081. URL: https://doi.org/10.1145/1024074.1024081.

[3] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[4] Michele Benzi. 'Preconditioning Techniques for Large Linear Systems: A Survey'. In: *Journal of Computational Physics* 182.2 (2002), pp. 418–477. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.2002.7176. URL: https://www.sciencedirect.com/science/article/pii/S0021999102971767.

[5] NVIDIA Blog. *XGBoost – What Is It and Why Does It Matter?* https://www.nvidia.com/en-us/glossary/xgboost/. InfoWorld 2019 Technology of the Year mention (accessed 2025-05-01).

[6] U.V. Catalyurek and C. Aykanat. 'Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication'. In: *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pp. 673–693. DOI: 10.1109/71.780863.

[7] Tianqi Chen and Carlos Guestrin. 'XGBoost: A Scalable Tree Boosting System'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM, Aug. 2016, pp. 785–794. DOI: 10.1145/2939672.2939785. URL: http://dx.doi.org/10.1145/2939672.2939785.

[8] Timothy A Davis and Yifan Hu. 'The University of Florida sparse matrix collection'. In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[9] Pedro Domingos. 'A few useful things to know about machine learning'. In: *Communications of the ACM* 55.10 (2012), pp. 78–87.

[10] Robert D. Falgout and Ulrike Meier Yang. 'hypre: A Library of High Performance Preconditioners'. In: *International Conference on Computational Science*. Springer, 2002, pp. 632–641.

[11] Yoav Freund and Robert E. Schapire. 'A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting'. In: *J. Comput. Syst. Sci.* 55.1 (1997), pp. 119–139. DOI: 10.1006/jcss.1997.1504.

[12] Jerome H. Friedman. ‘Greedy Function Approximation: A Gradient Boosting Machine’. In: *Ann. Statist.* 29.5 (2001), pp. 1189–1232. DOI: `10.1214/aos/1013203451`.

[13] Jianhua Gao et al. ‘A Systematic Literature Survey of Sparse Matrix-Vector Multiplication’. In: *arXiv preprint arXiv:2404.06047* (2024). Provides an overview of sparse formats including CSR (open-access).

[14] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow.* 2nd. Chapter 2-4 discuss preprocessing, scaling, and PCA. O’Reilly Media, 2019. ISBN: 9781492032649.

[15] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning.* Chapter 5: Machine Learning Basics. MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[16] William D. Gropp et al. ‘Towards realistic performance bounds for implicit CFD codes’. In: *Proceedings of the 2000 Parallel CFD Conference* (2000).

[17] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, 2010.

[18] Trevor Hastie et al. *The elements of statistical learning: data mining, inference, and prediction.* Vol. 2. Springer, 2009.

[19] Douglas M. Hawkins. ‘The Problem of Overfitting’. In: *Journal of Chemical Information and Computer Sciences* 44.1 (2004), pp. 1–12. DOI: `10.1021/ci0342472`.

[20] Arthur E. Hoerl and Robert W. Kennard. ‘Ridge Regression: Biased Estimation for Nonorthogonal Problems’. In: *Technometrics* 12.1 (1970), pp. 55–67. ISSN: 00401706. URL: `http://www.jstor.org/stable/1267351` (visited on 05/05/2025).

[21] Soojin Hwang et al. ‘Cerberus: Triple mode acceleration of sparse matrix and vector multiplication’. In: *ACM Transactions on Architecture and Code Optimization* 21.2 (2024), pp. 1–24.

[22] IBM. *What is overfitting?* Accessed: 2025-05-08. 2021. URL: `https://www.ibm.com/think/topics/overfitting`.

[23] Gareth James et al. *An Introduction to Statistical Learning.* 2nd ed. see “Qualitative Predictors” on dummy variables. Springer, 2021. Chap. 8.2.

[24] Ian T. Jolliffe. ‘Principal Component Analysis’. In: *Springer Series in Statistics* (2002). DOI: `10.1007/b98835`.

[25] George Karypis and Vipin Kumar. ‘A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs’. In: *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: `10.1137/S1064827595287997`. eprint: `https://doi.org/10.1137/S1064827595287997`. URL: `https://doi.org/10.1137/S1064827595287997`.

[26] Ron Kohavi. ‘A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection’. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI).* 1995, pp. 1137–1143.

[27] Jiajia Li et al. ‘SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication’. In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation.* 2013, pp. 117–126.

[28] Wai-Hung Liu and Andrew H. Sherman. 'Comparative Analysis of the Cuthill-McKee and the Reverse Cuthill-McKee Ordering Algorithms for Sparse Matrices'. In: *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 198–213. ISSN: 00361429. URL: `http://www.jstor.org/stable/2156087` (visited on 01/05/2025).

[29] Tom M Mitchell. *Machine learning.* Vol. 1. 9. McGraw-hill New York, 1997.

[30] Thaha Mohammed et al. 'DIESEL: A novel deep learning-based tool for SpMV computations and solving sparse linear equation systems'. In: *The Journal of Supercomputing* 77.6 (2021), pp. 6313–6355.

[31] Andrey Monakov. *Sparse Matrix-Vector Multiplication, Part 1.* AMD GPUOpen Labs. 2020. URL: `https://gpuopen.com/learn/sparse-matrix-vector-multiplication-part-1/`.

[32] Olav Møyner and co-authors. 'Accelerating the OPM Flow Solver on GPUs'. In: *arXiv preprint arXiv:2303.12345* (2023).

[33] Leonid Oliker, Rupak Biswas and William P. Nitzberg. *Effects of ordering strategies on sparse matrix vector multiply performance.* Tech. rep. NASA/CR-2002-211606. Accessed: 2025-04-07. NASA Ames Research Center, 2002. URL: `https://ntrs.nasa.gov/api/citations/20020058146/downloads/20020058146.pdf`.

[34] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[35] David MW Powers. 'Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation'. In: *arXiv preprint arXiv:2010.16061* (2020).

[36] Payam Refaeilzadeh, Lei Tang and Huan Liu. 'Cross-Validation'. In: *Encyclopedia of Database Systems.* Springer, 2009, pp. 532–538. DOI: `10.1007/978-0-387-39940-9_565`.

[37] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* 2nd. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2003. ISBN: 978-0898715347.

[38] C.E.M Schoutrop. 'Computational Physics of Low-Temperature Plasma Simulation: Stoichiometric transformations, Krylov methods and improved quadrature rules'. English. Proefschrift. Phd Thesis 1 (Research TU/e / Graduation TU/e). Applied Physics and Science Education, Feb. 2023. ISBN: 978-90-833062-1-6.

[39] G. A. F. Seber and Alan J. Lee. *Linear Regression Analysis.* 2nd. Wiley-Interscience, 2003. ISBN: 978-0471415404.

[40] Ravid Shwartz-Ziv and Amitai Armon. 'Tabular Data: Deep Learning is Not All You Need'. In: *Information Fusion* 81 (2022), pp. 84–90. ISSN: 1566-2535. DOI: `10.1016/j.inffus.2021.11.011`. URL: `https://www.sciencedirect.com/science/article/pii/S1566253521002360`.

[41] M. Stone. 'Cross-Validatory Choice and Assessment of Statistical Predictions'. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 36.2 (1974), pp. 111–147.

[42] James D Trotter. 'High-performance finite element computations'. PhD thesis. Simula Research Laboratory, 2020.

[43] James D. Trotter et al. ‘Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs’. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC ’23)*. SC ’23. Denver, CO, USA: Association for Computing Machinery, 2023. ISBN: 9798400701092. DOI: `10.1145/3581784.3607046`. URL: `https://doi.org/10.1145/3581784.3607046`.

[44] Serif Yesil et al. ‘Wise: Predicting the performance of sparse matrix vector multiplication with machine learning’. In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 2023, pp. 329–341.

[45] Min-Ling Zhang and Zhi-Hua Zhou. ‘A review on multi-label learning algorithms’. In: *IEEE Transactions on Knowledge and Data Engineering* 26.8 (2014), pp. 1819–1837.

[46] Haoran Zhao et al. ‘Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon’. In: *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 2020, pp. 601–609. DOI: `10.1109/ICCD50377.2020.00105`.