Optimizing Sparse Linear Algebra Through Automatic Format Selection and Machine Learning

Christodoulos Stylianou
EPCC, The University of Edinburgh
Edinburgh, UK
c.stylianou@ed.ac.uk

Michèle Weiland
EPCC, The University of Edinburgh
Edinburgh, UK
m.weiland@epcc.ed.ac.uk

Abstract—**Sparse matrices are an integral part of scientific simulations. As hardware evolves new sparse matrix storage formats are proposed aiming to exploit optimizations specific to the new hardware. In the era of heterogeneous computing, users often are required to use multiple formats for their applications to remain optimal across the different available hardware, resulting in larger development times and maintenance overhead. A potential solution to this problem is the use of a lightweight auto-tuner driven by Machine Learning (ML) that would select for the user an optimal format from a pool of available formats that will match the characteristics of the sparsity pattern, target hardware and operation to execute.**

In this paper, we introduce *Morpheus-Oracle*, a library that provides a lightweight ML auto-tuner capable of accurately predicting the optimal format across multiple backends, targeting the major HPC architectures aiming to eliminate any format selection input by the end-user. From more than 2000 real-life matrices, we achieve an average classification accuracy and balanced accuracy of 92.63% and 80.22% respectively across the available systems. The adoption of the auto-tuner results in average speedup of $1.1\times$ on CPUs and $1.5\times$ to $8\times$ on NVIDIA and AMD GPUs, with maximum speedups reaching up to $7\times$ and $1000\times$ respectively.

Index Terms—**sparse matrix storage formats, machine learning, automatic format selection**

I. INTRODUCTION

Sparse matrices, since their inception, have become a crucial component of many scientific simulations in computational science and engineering [1]–[3]. Over the years, more than 70 representations of sparse matrices (sparse matrix storage formats) have been developed, each addressing the different types of matrices and/or the evolution of hardware towards multi- and many-core processors and accelerators [4]. Literature shows that different formats yield different performance profiles with no single format performing optimally across all different kinds of matrices and types of hardware [5]–[10].

Exploiting the properties of each format and adjusting the underlying data structure of the sparse matrix (instead of adopting a single storage format) to better fit the operation and target hardware provides optimization opportunities that will potentially improve the performance of the operation. Since the matrix is generally unknown during compile-time, this exploitation can only be done at runtime. Previous efforts [11]–[14] provide abstractions and mechanisms that effectively allow for runtime switching to the different formats that are supported.

Even though it is crucial to enable some dynamic switching mechanism to be able to change formats at runtime, it is also important to automate the process of selecting the optimal format. To facilitate the selection of the optimal format for a given matrix, target architecture and operation, auto-tuners have been developed (such as [9], [13], [15], [16]) with a focus on selecting the optimal format for the Sparse Matrix-Vector Multiplication (SpMV), the operation that often dominates the runtime of computing the solution to linear systems. These contributions demonstrated the impact of optimizing the performance of iterative solvers through automatic format selection on single node multi- and many-core systems.

In this work, we develop an auto-tuning system that is capable of efficiently tuning the performance of the SpMV kernel on a wide range of state-of-the-art systems across different vendors using ML whilst remaining architecture agnostic. The auto-tuning system offers multiple ML algorithms each with different costs and selection accuracy. We introduce *Morpheus-Oracle (Oracle)* [17], a library that supports automatic format selection by implementing the different auto-tuners as well as the feature extraction routines. A detailed description of *Oracle* is given in Section VI.

In summary, our contributions are:

- We show the distribution of optimal formats for the SpMV operation over 6 different matrix storage formats across a variety of systems and target hardware, including multi- and many-core processors and accelerators, for more than 2000 matrices.
- For the same set of matrices and systems, we quantify the improvement in performance achieved by choosing the optimal format compared to Compressed Sparse Row (CSR), a commonly used general-purpose format. The average runtime speedup is up to $\sim 2\times$ on CPU backends and up to $\sim 10\times$ on GPU backends.
- We provide a reusable model generation system with more than 2000 sparse matrices from real applications that users can exploit to train their own models. In addition we train and tune two different ML algorithms and compare their performance in terms of their ability to correctly select the optimal format and quantify the overheads introduced by introducing the auto-tuning mechanism relatively to the runtime cost of SpMV iterations.
- We introduce a simple, flexible and extensible auto-

tuning library complementing *Morpheus*, the library for dynamic format switching, that provides tuners which can confidently select the optimal format for a range of architectures.

II. BACKGROUND AND MOTIVATION

A. Motivation

New storage formats are proposed every time new architectures emerge aiming to exploit optimizations specific to the new hardware. In the era of heterogeneous computing hardware has become more diverse and as a result applications often require the use of multiple formats across the different types of hardware in order to remain optimal. Even-though new formats have been proposed ([5], [6], [10] as an effort to mitigate the performance portability issue there is still no single format that would perform optimally across the different sparsity pattern, hardware architectures and operations. As we show in Section VII-C, even when a particular format performs well in general, in some cases can severely under-perform resulting in poor runtime performance. This is predominantly noticeable on GPUs where the wrong format can leave the device under-utilized or result in excessive memory requests due to uncoalesced memory accesses. For applications to achieve optimal performance therefore a better solution is to select the optimal format from a pool of candidate formats at runtime.

Experienced users may have a feeling about the choice of the optimal format for the category or type of matrices they frequently used, however a decision such as this one is not trivial to make as it depends on a number of factors. Furthermore, choosing the optimal format by running the available options first can result in significant overheads. ML and Artificial Intelligence (AI) have been successful in various optimization tasks ranging from code optimization to model selection [18], including the task of selecting the optimal sparse matrix storage format [13], [16]. Adopting a ML model has the potential to offer an accurate and low-overhead solution to the problem of automatic format selection, eliminating any requirement for manual format selection input from the user enabling applications to remain optimal across the different types of hardware and sparsity patterns for any operation of interest.

B. Sparse Matrix Storage Formats

A plethora of sparse matrix storage formats have been developed over the years, with new formats introduced every time new architectures emerge [4]. Each format has different storage requirements, computational characteristics and comes with different interface for modifying and manipulating the entries of the matrix [19]. Such characteristics make the process of determining, in advance, which format will perform best for a particular matrix given an operation and target hardware non-trivial, with multiple factors contributing in such a decision. In this paper, we consider six different storage formats: 2 general purpose, 2 specific purpose and 2 hybrid formats.

The most basic and well-known formats are Coordinate (COO) and CSR. Both formats are considered *general purpose* formats, and are suitable for a wide range of arbitrary sparsity patterns and target architectures, with CSR usually adopted as the format of choice. COO is constructed from three arrays, where each non-zero element is stored with its pair of row and column indices and no guarantees in the ordering of the elements. Similarly, CSR also stores explicitly the non-zero values and column indices, but compresses the row indices and stores and array of pointers to mark the boundaries of each row instead, effectively imposing a natural ordering across rows.

Specific purpose formats on the other hand aim to exploit the characteristics of a specific class of matrices and are usually designed to perform optimally on a particular architecture, such as GPUs. For example, the Diagonal (DIA) format is designed to represent regular sparsity patterns and is a good fit for vector-like processors. DIA stores the non-zero elements in a two-dimensional array, where each column holds the coefficients of the diagonal of the matrix. In addition, it holds an integer offset array that keeps track of where each diagonal starts. Another example of interest is the ELLPACK (ELL) format, which assumes that there are at most K non-zero entries per row. As a result, it uses one two-dimensional array to represent the non-zero elements of the matrix and another one for the column indices of the values. Consequently, the DIA format is suitable for representing structures that dominate along the diagonals, such as banded matrices, and ELL for matrices that are structured or semi-structured (i.e have similar number of non-zeros per row). Note that both formats can suffer from excessive padding if the number of diagonals or the number of non-zeros per row is very large.

Hybrid formats usually aim to combine the strengths of two formats in order to eliminate some of their weaknesses, such as the excessive padding mentioned before. The first hybrid format of interest is Hybrid (HYB), which is a combination of ELL and COO. HYB uses a parameter K_H that indicates the number of non-zeros per row to be stored in the ELL portion. For rows with number of non-zeros larger than K_H, the surplus of non-zeros is stored in the COO portion instead. The second hybrid format of interest is Hybrid DIA/CSR (HDC). It uses a parameter N_D that represents the number of non-zeros in a diagonal above which the diagonal is considered to be a *"true" diagonal*. The true diagonals in the matrix are then stored in DIA format, whilst the remainder are stored as CSR.

C. Morpheus

Morpheus [11] is a C++ library that supports the runtime switching of sparse matrix storage formats through *DynamicMatrix*, a single dynamic "abstract" format. *Morpheus* provides a transparent mechanism that can efficiently switch to the different formats supported by the *DynamicMatrix* and it currently supports the six formats mentioned in Section II-B. In addition, *Morpheus* offers algorithms such as SpMV for four different backends in order to support most of the key HPC platforms: 1) Serial (Sequential), 2) OpenMP (Multi-

threaded), 3) CUDA (NVIDIA GPUs) and 4) HIP (AMD GPUs). Furthermore, *Morpheus* provides data management routines and enables data transfers across the different memory spaces of the supported backends and adopts the host-device model to support both homogeneous and heterogeneous platforms (i.e platforms with CPU only or CPU+GPU hardware) maintaining the same application source code.

By abstracting the different formats under a single *dynamic* format, encapsulating the internal implementation details and provide a single interface for algorithms across backends, users are left with a simple and intuitive interface that abstracts away the complexities. In this work, we build on top of *Morpheus*'s runtime switching mechanism to enable automatic format selection and completely eliminate the need for format selection input from the user.

III. Auto-tuning Pipeline High Level Overview

The focus of this work is to develop an auto-tuner for selecting the optimal format for the *DynamicMatrix* provided by *Morpheus* to switch to given a matrix, an operation and a target hardware. The most straightforward approach for achieving this task is to utilize a run-first tuner that runs the operation of interest for every format supported, measures the desired metric and selects the best performing format as the optimum. Such an approach will be at expense of the overall runtime performance (even though it will provide the most accurate prediction) as it requires multiple expensive conversions between the different formats, with the expense increasing as more formats are added. A better approach, which reduces the prediction cost, is to use ML models to find the optimal format. A high-level overview of our proposed auto-tuning pipeline is shown in Figure 1.
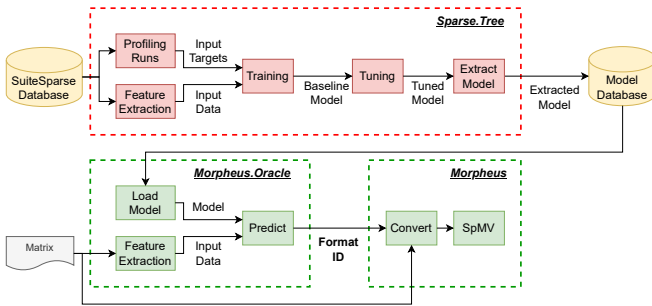

Fig. 1: High-level overview of the auto-tuning pipeline. Red and green boxes represent offline and online operations respectively.

The auto-tuning pipeline is divided in the offline (red) and online (green) stage. The offline stage has to be executed once for every new architecture and operation we want to do predictions for, and the results from that stage can then be reused during the online stage.

A. Offline Stage

The first stage of the pipeline is the actual model generation where we train, tune and extract the ML model in a file,

for a given architecture and operation, to be used later on by the auto-tuner. We use approximately 2200 real-valued, square matrices of varying sizes, sparsity patterns and different application domains, available from the *SuiteSparse Collection* [20].

For every matrix in the dataset, we first perform profiling runs on the operation and architecture of interest, and measure the runtime in order to determine the optimal format, i.e the format with the shortest runtime for the operation, and export its format ID to be used in the later stages. At the same time, we perform a feature extraction routine, described in detail in Section IV, on the matrices in order to generate inputs to be used during the training process. Both the input features (input data) and format ID (input targets) are used to train and tune the ML model, a description of which is given in Section V, and once the tuned model is obtained it is exported to a file and stored for later use.

To streamline the training process for users, we wrap this process in a *Python* framework called *Sparse.Tree*[1] which uses *scikit-learn* [21] under the hood. Users can use *Sparse.Tree* to generate models for new systems or use the pre-trained models from the *Model Database* for the x86 and ARM CPUs or NVIDIA and AMD GPUs used in this work.

B. Online Stage

In order to be able to automatically select the optimum format to be used by *Morpheus* in an application, we need to be able to make the decision efficiently and online, i.e. while the application is running. In the second stage of the pipeline, we implement *Oracle*, a C++ architecture-independent auto-tuner that uses the *DynamicMatrix* provided by *Morpheus* and loads an ML model from a file specified at runtime in order to make the decision. Note that by "architecture-independent" we refer to the fact that the tuner is agnostic to the target hardware it is tuning for, as this information is captured by the model loaded at runtime.

For the auto-tuner to be able to use the model, the features of the input matrix need to be extracted in the same way as during the first, training, stage. Then by traversing the model, *Oracle* returns the optimal format ID that *Morpheus* uses to switch to and perform the operation of interest. *Oracle* is described in more detail in Section VI.

IV. Feature Extraction

Feature extraction in the context of this work refers to the process of transforming the original sparse matrix into a set of numerical "features" that can be processed by the model while preserving the information about the sparsity pattern of the original matrix. Relevant features to the problem of interest result in a model that can make informed decisions, however there is a trade-off between the overheads required for computing these features and the accuracy of the decision that is made based on these features. In other words, by providing the model with a large amount of relevant information about

---

[1]Available at: https://github.com/morpheus-org/sparse.tree

Parameter	Description	Formula
M	# of rows	-
N	# of columns	-
NNZ	# of non-zeros	-
$\overline{NNZ}$	avg. NNZ per row	$\overline{NNZ} = \frac{NNZ}{M}$
ρ	density	$\rho = \frac{NNZ}{M*N}$
$max(NNZ)$	max NNZ per row	$max(NNZ) = max_{i=1}^{M} NNZ_i$
$min(NNZ)$	min NNZ per row	$min(NNZ) = min_{i=1}^{M} NNZ_i$
σ_{NNZ}	std of NNZ per row	$\sigma_{NNZ} = \frac{\sum_{i=1}^{M} |NNZ_i - \overline{NNZ}|^2}{M}$
N_D	# of diagonals	-
N_{TD}	# of true diagonals	-

TABLE I: Feature parameters used for training the model and, where relevant, the corresponding formula used for computing each one.

the problem will facilitate better learning, but at the cost of having to compute that information. For the purposes of this work, a set of 10 features has been selected (see Section IV-A) as shown in Table I, capturing information about the basic structure of the sparse matrix but also about the distribution of non-zeros across the rows and diagonals of the matrix.

A. Feature selection

The first three features – number of rows (M), number of columns (N) and number of non-zeros (NNZ) – aim to provide a general idea of the size of the matrix, and they are easy to capture as they are provided by the *DynamicMatrix*. According to Monakov et al. [8], COO is well suited for very sparse matrices with many empty rows and we therefore also add the average number of non-zerors per row $(\overline{NNZ})$ and the density (ρ) of the matrix to the set of features.

On the other hand, the performance of specific purpose formats, such as DIA and ELL, is heavily affected by the distribution of non-zeros as distributions that are not a good fit for each format will result in excessive padding of zero elements in the matrix hindering the overall performance. ELL allocates memory based on the maximum number of non-zero elements in a row, therefore matrices with extremely uneven distribution of non-zeros per row are not a good fit for such a format. This information is captured by measuring the maximum and minimum number of non-zeros per row $(max(NNZ), min(NNZ))$ and the standard deviation of the non-zeros per row (σ_{NNZ}). In a similar manner, the DIA format allocates memory based on the number of diagonals, therefore for matrices that have a large number of diagonals that each only have a few elements will again result in excessive padding. This time therefore the diagonals of the matrix are traversed, keeping count of the number of diagonals (N_D) with at least 1 non-zero and the number of true-diagonals (N_{TD}) that have number of non-zeros above a threshold. Since HYB and HDC are hybrid formats, the existing features remain representative. More details on how

the features are actually extracted for the different formats are given in Section VI-C.

V. MACHINE LEARNING MODEL

Our aim is to train a model that can predict the optimal storage format of a given sparse input matrix. This type of problem falls into the category of multi-class classification problems. During training, for each input matrix in the set, we extract a collection of 10 features and the target attribute that corresponds to the index of the optimal format, obtained from the profiling runs. The objective of the model is to try and determine a mapping between the input features and the optimal format ID, which can be described by Equation 1:

$$f(\vec{x_1}, \vec{x_2}, ..., \vec{x_n}) \rightarrow y_n(COO, CSR, ..., HDC) \quad (1)$$

where $\vec{x_i}$ represents the feature vector of the ith sparse matrix in the training set and y_n represents the target vector with each entry containing the index of one format from the six available.

To train a model to predict the value of the target of interest, we are using a decision tree ML algorithm that effectively learns simple decision rules inferred from the data features. The reasons for this choice are two-fold: firstly, it is simple to understand and interpret this method; and secondly, it requires little to no data preparation before training the model or using it for prediction. However, decision trees can overfit by creating over-complex trees that do not generalize the data well or become unstable in small variations in the data. To circumvent these issues and generate the optimal model architecture, an exhaustive Grid search is performed to search from the optimal hyperparameter values in a defined hyperparameter space. In addition, to improve the robustness of the model, an ensemble of decision trees is built, called a "random forest", that effectively fits a number of decision tree classifiers onto different sub-samples of the dataset. Whilst random forest classifiers can improve the predictive accuracy of the model and control overfitting, this comes at the expense of higher prediction times since multiple trees need to be traversed and the decision from each tree has to be combined into a single final result.

For this work we are specifically interested in training a model to predict the optimal format to be used during the SpMV operation for every backend supported in *Morpheus*, however the techniques and algorithms used here are transferable to other sparse operations. Models using both decision tree and random forest methods are trained and tuned in Python, for a number of x86 and ARM CPUs as well as NVIDIA and AMD GPUs, and extracted to a file to be used during the auto-tuning phase.

VI. MORPHEUS ORACLE - AN AUTO-TUNER FOR AUTOMATIC FORMAT SELECTION

To facilitate a systematic way of performing format *selection* we developed *Oracle*, a header-only C++ library for automatic format selection. It has been developed to complement the dynamic switching capabilities in *Morpheus* and tune

its performance by automating the process of selecting the optimal format to use for a given operation and target architecture. *Oracle* follows a similar design philosophy to *Morpheus*, where containers are separated from the algorithms. Containers here represent the different tuners that are supported by the package and are responsible for encapsulating the specifics of each tuner's implementation exposing the user only to an interface that configures and runs the tuner.

A. Tuners

Currently, *Oracle* supports three tuners: 1) *Run-first*, 2) *DecisionTreeTuner* and 3) *RandomForestTuner*. Each tuner is responsible for managing the complexities of selecting the optimal format. For example, the *Run-first* tuner records the iteration time each format takes to perform *N*-iterations for a given operation and applies statistics to determine which format was best. On the other hand, *DecisionTreeTuner* and *RandomForestTuner* require an ML model to be loaded from file that is represented using a tree structure which is traversed in order to determine the optimal format. *DecisionTreeTuner* only traverses a single tree whilst *RandomForestTuner* traverses multiple trees in the ensemble and then performs a voting scheme to decide the optimal format. In this case, the majority voting scheme is used that chooses the optimal format to be the one with the most votes.

The performance of each of the three tuners is a direct trade-off between runtime overhead and prediction accuracy. *Run-first* tuner offers the most accurate prediction at the expense of expensive conversions between each supported format and *DecisionTreeTuner* offers very fast but less accurate predictions. *RandomForestTuner* improves the prediction accuracy of *DecisionTreeTuner* by using multiple trees with the runtime of the prediction process proportional to the number of trees used. The support of multiple types of tuners allows users to choose one based on their requirements.

B. Operations

Each sparse algorithm offered by *Morpheus* can be tuned to use the optimal format, effectively optimizing the algorithm. An interface for the SpMV algorithm is defined through the *TuneMultiply* operation, and any additional operations will follow the same principle. By using compile-time introspection we can maintain a single high-level interface for the various supported operations and specialize the implementation for each tuner. Note that the *Oracle* package interacts with *Morpheus* by requesting: 1) the *DynamicMatrix* to tune for and 2) the execution space to run the operation in, and it returns the ID of the best format the *DynamicMatrix* should switch to.

The input of the tuning operation requires the *DynamicMatrix* and the tuner, along with the desired *execution space*, as template parameter. Upon completion of the tuning operation, the tuner can be queried for the optimal format. In the case where the *Run-first* tuner is passed to *TuneMultiply*, the tuning operation will perform *N*-iterations of SpMV for each format, with the tuner keeping track of the timings. However when ML tuners are used, they will evaluate the model and determine the optimal format. In order to be able to do so, ML tuners have to perform feature extraction on the fly.

C. Feature Extraction

The biggest challenge when dealing with sparse matrices is that each format has its own representation in memory that can be drastically different between formats. When it comes to feature extraction, the matrix has to be traversed in order to collect the necessary information. Extracting the features defined in Section IV on the fly normally would require traversing the rows and diagonals of the matrix multiple times, which for matrices with large number of rows can be expensive. For offline feature extraction during the training process traversing the matrix multiple times might be expensive, but not prohibitive, as the process is generally carried out only once. However, in an online scenario such as during the auto-tuning process this would potentially eliminate any benefits resulting from the optimization process.

To enable format extraction online, *Morpheus* has been extended to provide matrix statistics on a per-format basis and across the different supported backends, eliminating the need for any data transfers across spaces. Multiple statistics can be computed at the same time, reducing the number of times the matrix has to be traversed and reducing the runtime cost of the process. *Oracle* can then perform online feature extraction by inspecting the active format of the *DynamicMatrix* and in turn predict the optimal format by querying the model of the tuner.

VII. RESULTS AND EVALUATION

In this section, the performance of the ML auto-tuners available in *Oracle* is measured and evaluated. First, profiling runs are carried out to determine the distribution of the optimal format per matrix across the different systems available, and evaluate the performance benefit from switching to the optimal format instead of using a single default format (CSR). Next, the predictive performance of the ML models is quantified comparing the baseline and tuned models. The cost of the prediction process is quantified with respect to the time it takes to carry out a single SpMV operation and finally the performance of *Morpheus* in conjunction with the auto-tuner by *Oracle* is evaluated.

A. Setup

All experiments were carried out on the ARCHER2 [22], Cirrus [23] and Isambard [24] supercomputers; their compute node architectures are described in Table II. Experiments run across all four backends supported by *Morpheus* spanning a representative set of all major hardware architectures e.g x86 (Intel and AMD) and ARM CPUs as well as NVIDIA and AMD GPUs.

The dataset used in the experiments consists of approximately 2200 real, square matrices available from the SuiteSparse library, in an 80%-20% split between training and test set.

SYSTEM	SUBSYSTEM	QUEUE	CPU	GPU
ISAMBARD	A64FX	A64FX	1X FUJITSU A64FX (48 CORES)	-
	P3	INSTINCT	1X AMD EPYC 7543P (32 CORES)	4X AMD INSTINCT MI100
		AMPERE		4X NVIDIA AMPERE A100 40GB
	XCI	ARM	1X MARVELL THUNDERX2 ARM (32 CORES)	-
CIRRUS		STANDARD	2X INTEL XEON E5-2695 (18 CORES)	-
		GPU	2X INTEL XEON GOLD 6248 (18 CORES)	4X NVIDIA VOLTA V100 16GB
ARCHER2		STANDARD	2X AMD EPYC 7742 (64 CORES)	-

TABLE II: Node configurations for the systems used in the experiments.

B. Format Distribution

Since no single format performs best across different sparsity patterns, target hardware and operations, in order to get an understanding of which format performs optimally across the different matrices in the dataset, profiling runs of the SpMV operation are carried out on the available platforms. For every matrix in the dataset, supported format and available platform the runtime of 1000 SpMV repetitions is recorded and the format with the minimum runtime is set to be the optimal format for the particular matrix and platform. Figure 2 shows the distribution of the optimal formats per system and format for all matrices in the SuiteSparse dataset.
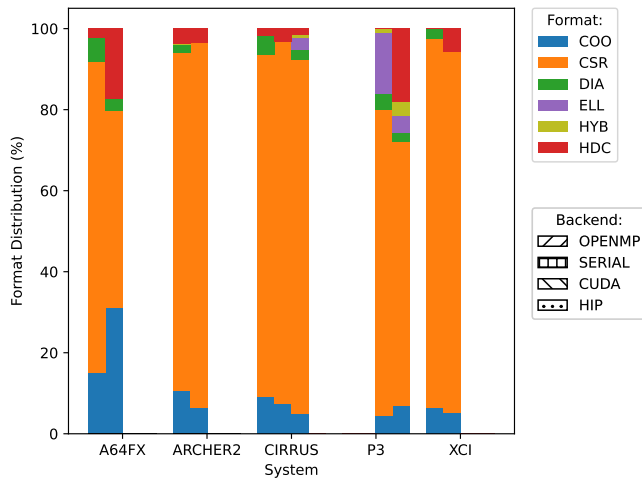

Fig. 2: Optimal Format distribution for 1000 repetitions of SpMV using the SuiteSparse dataset. The optimal format for each matrix is selected to be the one with the smallest runtime.

From Figure 2 it is clearly shown that for the biggest portion of matrices in the set the optimal format across systems and backends is CSR, validating its role as the most commonly used storage format. However, we do observe that even on the same hardware the distribution can change quite drastically. For the Serial backend on the A64FX system more matrices perform better by using HDC or COO formats, whereas for the OpenMP backend these become CSR. However, on Cirrus and Archer2 systems the opposite effect is observed where matrices that perform best with CSR on the Serial

backend are actually performing best with COO or DIA on the OpenMP backend. In addition, the distribution on GPU systems compared to the CPU systems is much more diverse with optimal formats chosen from almost every available format class.

The main takeaway here is that the format distribution for every system/backend in the experiment is unbalanced, with CSR being the clear majority. Therefore, the classification problem in Section V falls in the category of the imbalanced classification problems or rare event prediction. A large portion of real-life matrices perform well using CSR, however an auto-tuner that can predict rare events is useful if in the case where selecting a different format benefits performance noticeably.

C. Optimal Format Performance

As part of the experiment shown in Section VII-B, the timings of SpMV operation were recorded in an effort to quantify the real benefit in those cases where the optimal format is not CSR. In Figure 3 the runtime of SpMV using CSR is measured against the equivalent runtime of the optimal format for each matrix in the dataset on the OpenMP backend and across the available systems. Note that matrices with optimal format set to CSR are omitted for clarity. Whilst a lot of the matrices result in a speedup of less than $1.5\times$, there is a noticeable number of matrices that exhibit speedups between $1.5\times$ and $10.5\times$, with an average speedup of approximately $1.8\times$ for Cirrus, XCI and A64FX, and of $1.3\times$ on Archer2. Similar results are obtained for the Serial backend on the same systems; these have been omitted in the interest of the available space.
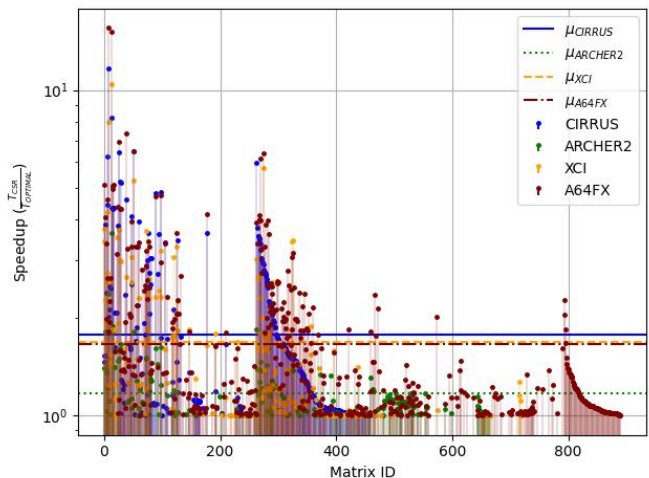

Fig. 3: Runtime speedup of SpMV using the optimal format against CSR on the OpenMP backend of the available systems for the SuiteSparse dataset. Matrices with optimal format set to CSR are omitted for clarity.

For the CUDA and HIP backends, the runtime speedups are more noticeable compared to the ones measured on the CPU backends. As shown in Figure 4, the average speedup for the CUDA and HIP backends is $8\times$ and $10\times$ respectively,
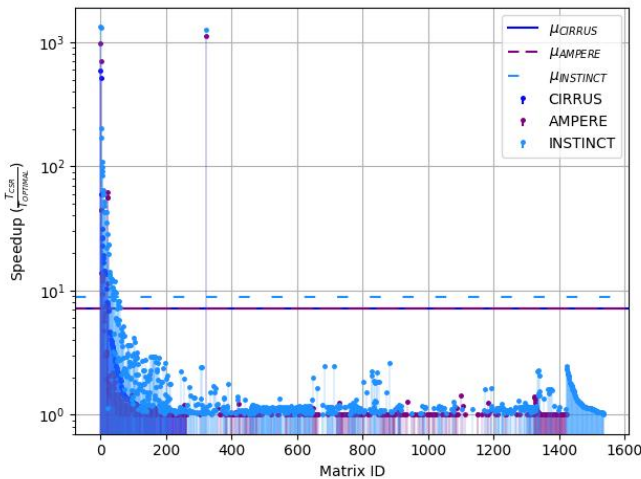
Fig. 4: Runtime speedup of SpMV using the optimal format against CSR on CUDA and HIP backends. An NVIDIA V100 and A100 GPU is used on Cirrus and Ampere (Isambard) systems and an AMD MI100 on Instinct (Isambard). Matrices with optimal format set to CSR are omitted for clarity. The average speedup is $8\times$ and $10\times$ on CUDA (both for V100 and A100) and HIP backends respectively.

with maximum speedups reaching up to $1000\times$. After closer inspection of the memory read/write requests, and the occupancy achieved during the launch of the SpMV kernel, for one of the matrices (*mawi_201512020030*), the CSR version issues $5\times$ more requests and the occupancy is $10\times$ smaller compared to the version using the optimal format. The sparsity pattern of the matrix results in uncoalesced accesses and leaves the GPU under-utilized when the CSR format is used.

These results justify the development and use of an auto-tuner such as the one proposed in this paper since the choice of format can have noticeable benefits in the performance of the operation. However, the auto-tuner must be lightweight to avoid performance degradation in the case where the optimal format is CSR.

D. Hyperparameter Tuning

Hyperparameter tuning for ML algorithms is essential for the overall performance of the ML model. This process relies more on experimental results rather than theory and the best method to determine the optimal settings is by trying different hyperparameter combinations and evaluate the performance of the model. To account for overfitting and ensure the model generalizes well on unseen data we perform a 5-fold CV on the training set and iteratively fit the model 5 times each time training on 4 folds and validating on the $5th$.

For hyperparameter tuning, a grid search is performed to search for the optimal hyperparameter values in a defined hyperparameter space, each time performing the entire 5-fold CV process. We compare all the generated models and the best one is selected to train on, using the full training set, obtaining the tuned model. In our case, we perform the tuning process for both the *decision tree* and *random forest* classifiers.

Table III shows the hyperparameters used to generate the baseline (left sub-columns) and tuned (right sub-columns) *random forest* for each available backend and system, along with the achieved accuracy and balanced accuracy on the test set. Similar qualitative results were achieved for the *decision tree* classifier, hence are omitted in the interest of the available space. We tune for the main parameters that affect the generalization ability of a *random forest* classifier (the number of estimators, max depth of the trees and the minimum number of samples on the leaf nodes) and we also test different secondary parameters such as bootstrap (sampling data points with or without replacement), minimum samples required before a split, maximum features considered before splitting a node and the criterion function used to measure the quality of the split.

Even though the tuning process results in a model that reports a similar average accuracy score (92.63%) to the baseline (92.36%), the tuned model is using significantly fewer and shallower trees (estimators) resulting in much faster prediction times. However, since the dataset is unbalanced, a more indicative metric to report is the balanced accuracy

System	Backend	Estimators		Bootstrap		Max Depth		Min Samples Leaf		Min Samples Split		Max Features		Criterion		Accuracy (%)		Balanced Accuracy (%)	
Archer2	Serial		40		F	21	21		3		2		4		entropy	95.13	**95.86**	83.65	**88.49**
	OpenMP		40		T	20	14		1		10		9		entropy	91.94	**92.18**	83.39	**85.12**
Cirrus	Serial		50		T	18	18		2		2		6		entropy	93.60	**94.08**	79.13	**81.71**
	OpenMP		30		T	19	15		1		10		8		gini	**92.65**	91.71	68.36	**78.39**
	Cuda		50		F	17	16		1		10		4		entropy	92.89	**93.60**	71.72	**74.32**
A64FX	Serial	100	90	T	T	20	18	1	1	2	2	10	5	gini	gini	87.93	**87.93**	84.56	**86.22**
	OpenMP		30		T	19	13		1		5		6		gini	91.26	**91.75**	89.29	**91.62**
P3	Cuda		40		F	22	14		2		10		4		entropy	86.46	**87.17**	**84.95**	83.82
	HIP		40		T	19	11		1		2		6		entropy	**93.38**	92.67	**90.82**	87.92
XCI	Serial		60		F	24	12		2		10		6		entropy	96.21	**96.92**	91.34	**95.72**
	OpenMP		20		T	16	10		1		5		10		entropy	94.54	**95.01**	55.25	**75.31**
															Mean	92.36	92.63	80.22	84.42
															Std ($\pm$)	2.93	3.02	11.04	6.64

TABLE III: Hyperparameters used during the tuning process of the Random Forest classifier. The accuracy and balanced accuracy achieved for the test set are also reported. We report both the baseline (left sub-columns) and tuned (right sub-columns) classifier parameters and score. The tuning process uses a 5-fold Cross Validation (CV) method.

calculated as the average of the proportion of correctly classified samples of each class individually. In this case, the tuned model (84.42% ± 6.64%) performs noticeably better compared to the baseline (80.22% ± 11.04%). It is worth pointing out that for some system and backend pairs the change in balanced accuracy is quite drastic (e.g 10% and 20% increase for the XCI and Cirrus systems using the OpenMP backend respectively).

For reference, the accuracy and balanced accuracy achieved on the tuned *decision tree* is 90.85% ± 7.87% and 78.12% ± 4.91% respectively. This result justifies the development of both *DecisionTreeTuner* and *RandomForestTuner* as this allows for a faster predictions when the *DecisionTreeTuner* is used without significant sacrifice in prediction accuracy.

E. Auto-tuner Performance

The tuned *random forest* classifier with parameters set as in Table III is deployed in C++ using the *RandomForestTuner* in *Oracle* on a synthetic benchmark. The benchmark performs 1000 SpMV operations using a sparse matrix switched to the optimum format selected by the tuner at runtime. The dataset used in the benchmark consists of all the matrices in the test set. For each system and backend pair, the auto-tuner runtime performance is measured in the form of equivalent SpMV operation using the CSR format, given by $T_{tuning} = \frac{T_{CSR}}{T_{FE}+T_{PRED}}$, where T_{CSR}, T_{FE} and T_{PRED} are the runtime of a single CSR SpMV, feature extraction and prediction operations respectively.

System	Backend	Mean	Std	Min	Q1	Q2	Q3	Max
Archer2	Serial	10	19	2	4	7	10	303
	OpenMP	25	20	2	14	21	31	179
Cirrus	Serial	10	30	2	3	4	7	359
	OpenMP	64	72	2	21	33	83	643
	CUDA	7	3	1	6	6	8	29
A64FX	Serial	6	9	1	3	4	5	120
	OpenMP	45	40	1	23	28	59	246
P3	CUDA	2	3	1	2	2	2	42
	HIP	15	9	1	7	18	23	30
XCI	Serial	12	28	2	6	7	9	335
	OpenMP	17	29	2	3	6	18	203

TABLE IV: The runtime cost, expressed in terms of SpMV operations using CSR, of using the auto-tuner. Measured as $T_{tuning} = \frac{T_{CSR}}{T_{FE}+T_{PRED}}$, where T_{CSR}, T_{FE} and T_{PRED} are the runtime of a single CSR SpMV, feature extraction and prediction operation respectively. Columns show the statistics of the runtime cost and Q1 to Q3 represent the quartiles.

Table IV shows statistics on the runtime cost of the tuner in the form of the number of SpMV operations using CSR for every system and backend pair. We do observe that the OpenMP backend requires on average more time to run the auto-tuner irrespective of the system, even-though it might use fewer estimators compared to the equivalent Serial backend. This indicates that the *random forest* using the OpenMP backend consists of fewer but more complex estimators compared to Serial. *RandomForestTuner* on GPUs on the other hand spends less time running the auto-tuner as the average runtime cost is only few repetitions compared to CPU backends. At least 75%

of the matrices in the test set require fewer than 100 repetitions for the tuning process. In real-life applications, for example solving a time-dependent Partial Differential Equation (PDE), would require many thousands of SpMV operations meaning that the auto-tuner proposed would not incur noticeable overheads. Even in the maximum cases observed, the cost of running the tuner remains within accepted limits.

F. Tuned SpMV Performance

Using the same setup as in Section VII-E we also quantify the runtime speedups in SpMV obtained by adopting the proposed auto-tuner compared to SpMV using CSR. The speedup is measured as shown in Equation 2:
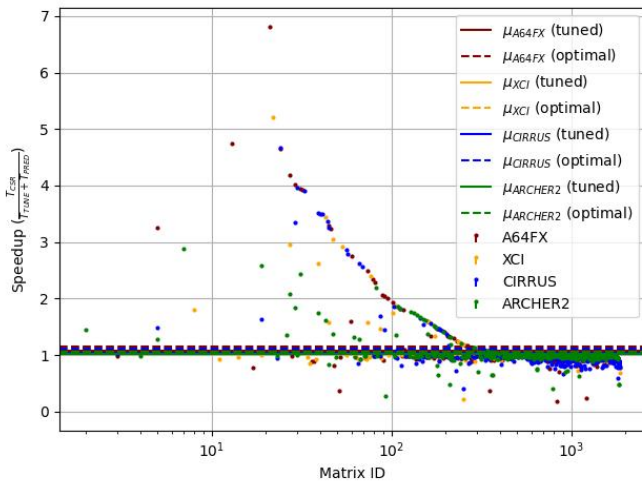
$$Speedup = \frac{T_{CSR}}{T_{TUNE}+T_{OPT}} = \frac{T_{CSR}}{T_{FE}+T_{PRED}+T_{OPT}} \quad (2)$$

where T_{CSR}, T_{OPT}, T_{FE} and T_{PRED} are the runtime of 1000 SpMV operations using CSR and the predicted format, feature extraction and prediction operations respectively.
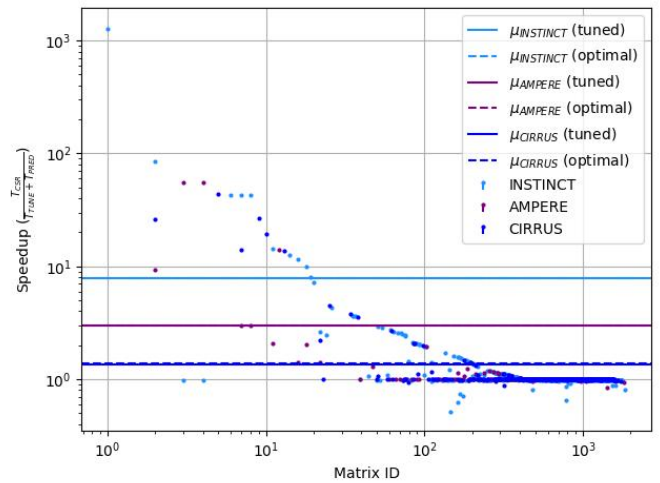
Figure 5 shows the achieved speedup on OpenMP, CUDA and HIP backends on the available systems for all the matrices in the test set. Note that the Serial backend is excluded due to space constraints as a similar trend to the OpenMP backend is observed. On CPUs, the speedup from introducing the auto-tuner and selecting the predicted format results in similar performance as if we were to use the CSR format. The result is consistent across all available systems with average speedup close to 1.1×. For the majority of matrices the overheads from the auto-tuner do not reduce overall performance as most of the samples are concentrated around 1. The few for which performance falls significantly below 1, we do observe the impact from wrongly classifying the optimal format. However, in many cases the auto-tuning process results in noticeable speedups, with maximum achieved speedup of 7× on the A64FX system.

On the GPU backends (Figure 5b), the auto-tuning approach is much more beneficial compared to the OpenMP backend as higher average speedups are achieved. On average, for the NVIDIA A100 (Ampere) and V100 (Cirrus) GPUs a 1.5× and 3× average speedup is achieved respectively, a result that follows the bandwidth ratio between the two architectures. The AMD M150 (Instinct) GPU however is the one architecture that clearly benefits from the introduction of the auto-tuner as it reports an average speedup of 8×. Note that compared to the CPU backends, on GPUs the performance of a mis-classification is less severe as fewer samples fall significantly below 1. In addition, for a number of matrices the achieved speedup improves performance by orders of magnitude highlighting the importance of adopting an auto-tuning approach for format selection. In all three backends shown in Figure 5 the average speedup achieved from using the auto-tuner matches the average optimal speedup for when the optimum format was selected (without performing any auto-tuning) suggesting that the overheads introduced by the auto-tuner become negligible as the number of SpMV repetitions increases.

(a) OpenMP

(b) CUDA/HIP

Fig. 5: Obtained runtime speedup from using the auto-tuner and predicted format against using CSR in performing 1000 SpMV operations on the available systems for every matrix in the test set. The average speedup (μ) from using the predicted format (includes also the time the tuner requires to make a prediction) matches the average speedup from using directly the optimal format, indicating that minimal overheads are introduced on average from using the tuner.

VIII. RELATED WORK

Over the years ML has been proven a valuable approach for performing various optimization tasks such as code optimization, task scheduling and model selection [18]. Applying auto-tuning techniques for optimizing sparse linear algebra remains an active area of research with developments spanning topics from format specific parameter tuning ([5], [10], [18]) to automatic format selection across current and emerging architectures.

Benatia et al. [25] proposed an Support Vector Machine (SVM) classifier for selecting the optimal format from four available formats for the SpMV on GPUs, reporting classification accuracy up to 88%. Similarly, Sedeghati et al. [26] used a Decision Tree based classifier to choose from five available formats resulting in 81% accuracy on GPUs. On the other hand, Li et al. [13] used an input adaptive SpMV auto-tuner based on ruleset classification that maintains a confidence value for each test sample. If the prediction of the classifier is below the defined threshold, the auto-tuner selects the optimal format and SpMV kernel otherwise switches to a run-first approach to make the decision, reporting accuracy of up to 85%.

To alleviate the problem of manually defining the features to be extracted, Zhao et al. [16] uses a Convolutional Neural Network (CNN) model for selecting the optimal format for SpMV both on CPU and GPU platforms. This approach requires a transformation/compression on the input matrix to a fixed size (128×128) image-like representation to be fed to the network, however reports the highest accuracy from all (93% and 90% on CPU and GPU platforms respectively) at the expense of higher prediction time compared to the ML alternatives.

Zhao et al. [27] takes the format selection approach a step further proposing an overhead-conscious selection mechanism for SpMV-based applications that also takes into account the overheads from format conversion. By building several regression models reports accuracy up to 88% and average runtime speedup of applications in the range of $1.14\times$ - $1.43\times$.

Our proposed auto-tuner manages to combine speed of ML methods such as the ones proposed by Li et al. and the accuracy reported by Zhao et al. on a range of storage formats and across most of the key HPC platforms. Note however that even-though the datasets reported in previous work are unbalanced, none of the authors reports the achieved balanced accuracy therefore is not possible to make a fair comparison between contributions based on the accuracy alone.

IX. CONCLUSIONS AND FURTHER WORK

Selecting the optimal sparse matrix storage format is important for allowing applications to remain optimal across the available hardware architectures. However, the selection process is not a trivial task. ML offers a systematic solution to this problem by approaching it as a classification task. In the case of the format selection, this problem can be categorized as a rare event prediction problem due to the imbalance observed in the data. By training, tuning and deploying an ensemble of *decision trees*, we are able to accurately predict the optimum format to be used for the SpMV operation across the main HPC architectures. We find out that although most of the time the best option is to use CSR, in some cases the runtime performance is improved by orders of magnitude from switching to the optimal format. Finally, our proposed light-weight auto-tuning approach introduces overheads in the overall runtime of SpMV which are amortised quickly and

within a few SpMV operations on average, with a more noticeable benefit to GPUs.

As a next step, we will explore ways of further improving the accuracy of our models either through balancing the dataset or other ML methods such as *gradient-boosted decision trees.* Furthermore, eliminating the requirement for manual feature extraction remains an avenue for further research.

REFERENCES

[1] T. Ichimura, K. Fujita, T. Yamaguchi, A. Naruse, J. C. Wells, T. C. Schulthess, T. P. Straatsma, C. J. Zimmer, M. Martinasso, K. Nakajima, M. Hori, and L. Maddegedara, "A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2018, pp. 627–637.

[2] A. M. B. Owenson, S. A. Wright, R. A. Bunt, Y. K. Ho, M. J. Street, and S. A. Jarvis, "An unstructured CFD mini-application for the performance prediction of a production CFD code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 10, p. e5443, 2020, e5443 cpe.5443. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5443

[3] P. Colella, "Defining Software Requirements for Scientific Computing," http://view.eecs.berkeley.edu/w/images/temp/6/6e/20061003235551!DARPAHPCS.ppt, 2006.

[4] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse Matrix-Vector Multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, Jan. 2017. [Online]. Available: https://doi.org/10.1145/3017994

[5] E. Coronado-Barrientos, M. Antonioletti, and A. Garcia-Loureiro, "A new AXT format for an efficient SpMV product using AVX-512 instructions and CUDA," *Advances in Engineering Software*, vol. 156, p. 102997, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997821000260

[6] W. Liu and B. Vinter, "CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 339–350. [Online]. Available: https://doi.org/10.1145/2751205.2751209

[7] W. Yang, K. Li, Y. Liu, L. Shi, and L. Wan, "Optimization of quasi-diagonal matrix–vector multiplication on GPU," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 183–195, 2014. [Online]. Available: https://doi.org/10.1177/1094342013501126

[8] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures," in *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, ser. HiPEAC'10. Berlin, Heidelberg: Springer-Verlag, 2010, p. 111–125. [Online]. Available: https://doi.org/10.1007/978-3-642-11515-8_10

[9] B.-Y. Su and K. Keutzer, "ClSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 353–364. [Online]. Available: https://doi.org/10.1145/2304576.2304624

[10] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, "A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units," *SIAM Journal on Scientific Computing*, vol. 36, 07 2013.

[11] C. Stylianou and M. Weiland, "Exploiting dynamic sparse matrices for performance portable linear algebra operations," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov 2022, pp. 47–57. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/P3HPC56579.2022.00010

[12] S. Filippone and A. Buttari, "Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003," *ACM Trans. Math. Softw.*, vol. 38, no. 4, aug 2012. [Online]. Available: https://doi.org/10.1145/2331130.2331131

[13] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An Input Adaptive Auto-Tuner for Sparse Matrix-Vector Multiplication," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 117–126. [Online]. Available: https://doi.org/10.1145/2491956.2462181

[14] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing," *ACM Transactions on Mathematical Software*, vol. 48, no. 1, pp. 2:1–2:33, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3480935

[15] G. Tan, J. Liu, and J. Li, "Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture," *ACM Trans. Math. Softw.*, vol. 44, no. 4, Aug. 2018. [Online]. Available: https://doi.org/10.1145/3218823

[16] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the Gap between Deep Learning and Sparse Matrix Format Selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 94–108. [Online]. Available: https://doi.org/10.1145/3178487.3178495

[17] C. Stylianou, "Morpheus-Oracle: a library for efficient automatic format selection of sparse matrix storage formats," 2 2023. [Online]. Available: https://github.com/morpheus-org/morpheus-oracle

[18] S. Chen, J. Fang, D. Chen, C. Xu, and Z. Wang, "Adaptive Optimization of Sparse Matrix-Vector Multiplication on Emerging Many-Core Architectures," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2018, pp. 649–658.

[19] N. Bell and M. Garl, "Efficient sparse matrix-vector multiplication on cuda," NVIDIA, Tech. Rep., 2008.

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

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] EPCC, "ARCHER2 UK National Supercomputing Service," 2022. [Online]. Available: http://www.archer2.ac.uk/

[23] EPCC, "Cirrus UK National Tier-2 HPC Service," 2020. [Online]. Available: http://www.cirrus.ac.uk/

[24] GW4 and UK Met Office, "Isambard 2 UK National Tier-2 HPC Service," 2022. [Online]. Available: http://gw4.ac.uk/isambard/

[25] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 496–505.

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

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