Received 17 April 2024, accepted 14 May 2024, date of publication 17 May 2024, date of current version 24 May 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3402326

RESEARCH ARTICLE

STuning-DL: Model-Driven Autotuning of Sparse GPU Kernels for Deep Learning

ROBERTO L. CASTRO, DIEGO ANDRADE, AND BASILIO B. FRAGUELA
CITIC, Computer Architecture Group, University of A Coruña, 15071 A Coruña, Spain

Corresponding author: Roberto L. Castro (roberto.lopez.castro@udc.es)

This work was supported by grant PID2022-136435NB-I00, funded by MCIN/AEI/ 10.13039/501100011033 and by ''ERDF A way of making Europe'', EU; also by Xunta de Galicia under the ConsolidaTon Programme of CompeTTve Reference Groups, ref. ED431C 2021/30. The work of Roberto L. Castro was supported by a predoctoral grant from the Ministry of Science, InnovaTon and UniversiTes, ref. FPU19/03974.

ABSTRACT The relentless growth of modern Machine Learning models has spurred the adoption of sparsification techniques to simplify their architectures and reduce the computational demands. Network pruning has demonstrated success in maintaining original network accuracy while shedding significant portions of the original weights. However, leveraging this sparsity efficiently remains challenging due to computational irregularities, particularly in GPU kernels. A new trend of template-based GPU kernels for semi-structured sparsity shows promise in efficiency but lacks autotuning capabilities to adapt to input dynamics, often underperforming in scenarios where they have not been meticulously hand-tuned. We present STuning-DL, the first pruning-aware autotuner for third-party template-based implementations enabling efficient optimization of sparse kernels for Deep Learning, spanning from high-level aspects (CUDA C++ level) down to GPU-native instructions specifics (assembly-level). STuning-DL tunes and optimizes at run-time sparse kernels' performance for each input problem, yielding speedups of up to 5.42× on NVIDIA T4-16GB and up to 3.6× on NVIDIA A100-40GB GPU in sparse matrices from real world models compared to existing heuristics from sparse libraries like cuSparse and cuSparseLt.

INDEX TERMS CUDA, GPU, learning-based predictive model, network pruning, sparse computation, SpMM, Tensor Core.

I. INTRODUCTION

Modern Machine Learning (ML) models, such as transformer-based architectures, have obtained unrivaled performance in multiple domains like Natural Language Processing (NLP) with Large Language Models (LLMs) [60], or Computer Vision with Vision Transformers (ViT) [14], [59]. However, generally, this remarkable progress comes at the cost of burgeoning architectural complexity, escalating computational demands, and heightened energy consumption [1], [3], [20], [25].

GEneral Matrix-Matrix Multiplication (GEMM) is a fundamental operation in many algorithms including most ML models. As a consequence, improving its performance has been a topic of continuous research and development over time [31], [64]. However, optimizing this routine to cater to diverse input problems and platforms presents a non-trivial challenge. This complexity is especially pronounced within the realm of ML due to the widely varying properties exhibited by each layer within a model [26], [28]. Moreover, certain ML applications grapple with additional complexities stemming from this challenge. One such instance arises in the context of Automated Machine Learning (AutoML), where the network architecture undergoes continuous evolution in pursuit of the optimal configuration tailored to a specific input dataset [13], [63]. In that sense, the autotuning of computational kernels has garnered significant interest, offering various strategies to optimize GEMM kernels based

The associate editor coordinating the review of this manuscript and approving it for publication was Yiming Tang.

on the specific characteristics of the input problem and hardware architecture [10], [30], [58], [65], [69].

In this pursuit of accelerating GEMM computations and enhancing the overall performance of ML models, recent research has ventured into the domain of weight pruning [22]. The core aim of weight pruning is to generate faster lightweight representations of dense original models with reduced memory consumption, while simultaneously endeavoring to avoid or minimize any increase in loss of accuracy. This is achieved through the removal of the weights considered less relevant in the overall network. Depending on the granularity of the pruning technique employed (structural/non-structural), the outcome of the pruning process can be sparse matrices serving as representations for network components. Consequently, GEMMs are transformed into Sparse Matrix-Matrix Multiplications (SpMMs), resulting in a fundamental alteration in the computations.

Historically, sparse computation has been a recurring theme in the domain of High Performance Computing (HPC) [11]. However, sparsity in HPC has been predominantly associated with scientific workloads, characterized by sparse matrices that exhibit markedly different properties from those encountered in Deep Learning (DL) [16]. These distinguishing characteristics include: (1) exceedingly high sparsity levels ($\geq$ 99%), (2) a concentration of non-zero elements constituting quite structural matrices (e.g., banded matrices), (3) shorter row lengths, and (4) greater variability in row lengths within a matrix. Autotuning techniques for sparse computation in scientific problems have been studied before; however, (1) they are mostly limited to Sparse-Matrix Vector Multiplications (SpMV), which are largely less common in DL than SpMMs, and (2) the design and the procedures are strongly tailored to the aforementioned input characteristics, which renders the tools and findings gleaned from previous studies in this field non-transferable to the domain of sparsity in DL [8], [15], [36], [37], [50].

As a consequence of these strong differences, the advent of sparse computation in DL has prompted the development of novel implementations tailored to accommodate these distinctive workloads. However, a critical aspect remains unaddressed – the development of autotuning techniques for *sparse GPU kernels tailored specifically to DL problems*. A recent trend in DL compilers has attempted to address this open question by automating the generation of code for sparse computations [8], [57], [61], [62], [68], [71]. However, as is frequently the case with such tools, their aim of adaptability to various platforms and formats often comes at the cost of performance, preventing them from reaching the level of hardware-native performance that ad-hoc implementations can achieve.

In the pursuit of creating a novel autotuner that addresses this open question, first, we made an in-depth study of the techniques that have historically been applied in the context of autotuning to linear algebra kernels. While hand-written autotuning heuristics have long been employed in this context, recent research has revealed their susceptibility to issues related to input specifications and limited generalizability across diverse inputs [28], [64]. This concern is particularly critical in the context of DL, where the potential range of configurations is vast, especially when incorporating sparsity into the equation. In response to these limitations, a burgeoning trend involves the utilization of *Predictive Models* for the development of adaptive and input-aware libraries with enhanced efficiency [28]. Nevertheless, applying this approach in the context of sparsity for DL introduces different problems that constitute an important challenge, namely:

Problem 1: Implementations are highly-coupled to one compressed storage format. Until now, the focus in sparsity for HPC has predominantly revolved around optimizing performance aspects. Consequently, previous studies mainly concentrated on autotuning the sparse compression formats employed to represent sparse inputs, with the primary aim of achieving performance enhancements [37], [62].

However, in the context of sparsity for DL, a pivotal and novel dimension emerges as a critical concern: *the accuracy of the network*. Sparse input matrices are inherently shaped by the pruning algorithms, which can generate highly irregular sparse matrices [16]. This irregularity, however, can significantly undermine performance due to inefficient hardware utilization [4]. Therefore, a new trend of semi-structured pruning techniques, which aims to find trade-offs between performance and accuracy, can yield quite structured patterns that offer better performance, but little to no room for tuning their representation [29]. This trend is caused because of the tremendous complexity of achieving speedups with sparse matrices in DL, since they exhibit a relatively high density, and their irregular nature does not align with the highly regular execution style of GPUs, particularly Tensor Cores Units (TCU).

The aforementioned situation imposes a substantial limitation when attempting to create an autotuning system, which severely restricts their tunability in terms of formats and makes the custom existing implementations only useful for a particular distribution of non-zeros. This renders the core principles of previous research, which primarily concentrated on refining sparse representations, inapplicable to this novel problem.

Problem 2: Poor generalization to new input problems and other platforms. The daunting task of crafting efficient kernels for sparse computation in DL has spurred the proliferation of specialized kernels tailored to address specific input problem shapes and hardware architectures [4], [7], [16]. This limitation recognizes the inherent difficulty of preserving the performance across all conceivable scenarios. Among the most critical issues with this trend, is their inability to perform effectively with input shapes other than those for which they have been optimized, which typically are small/medium problem sizes. Consequently, when dealing with a larger input problem, they fail to scale efficiently. Regarding platform portability, existing implementations are also highly coupled and optimized for a particular architecture because of this complexity. Hence, this renders them non-portable in terms of performance across various hardware architectures.

This phenomenon has already been empirically demonstrated in prior research [5], and it represents an important challenge in terms of autotuning, which aims to address adaptability to varying input dynamics and hardware platforms.

Problem 3: Lack of autotuning engines for sparsity on DL. The limitations described above have hindered the progression of sparse implementations for DL from aligning with the inherent characteristics of dense linear algebra libraries: tunability, scalability, and portability across diverse architectures. The lack of previous studies has represented an important challenge, and this paper aims to answer the questions present in this field in that sense, as well as to establish a way to design new implementations that would favor the aforementioned desirable features in the context of sparse linear algebra libraries for DL.

Template-based GPU kernel implementations are emerging as a promising solution to the previously mentioned limitations. Similar developments have already proven successful in the realm of dense computations [64], offering a middle ground between the hardware-native performance of hand-tuned implementations, which provide limited room for autotuning, and the lower performance but highly tunable and portable approach of DL compilers.

In this paper we introduce *STuning-DL*, a new autotuning tool for accelerating sparse template-based GPU kernels for DL. STuning-DL confronts this scenario by targeting an ultimate goal: *instantiating template-based kernel implementations* with the utilization of *Predictive Models* for the development of adaptive and input-aware libraries with enhanced efficiency. As we mentioned above, a similar idea has been applied to dense computation before, but to the best of our knowledge, this *concept has not been explored yet within the realm of sparse computation for DL.* This target is achieved by creating a vast hardware-dependent tuning space (assembly-level), instead of just facing the autotuning problem for sparse computation from a purely software-level perspective (e.g., sparse format), as the vast majority of existing works do.

This papers addresses these 3 problems with the following contributions:

- We first provide an in-depth analysis of the potential flaws and opportunities for autotuning in the field of sparse computation for DL workloads, particularly focused on SpMM problems. We also study the effectiveness of existent solutions in the context of DL.
- We present a detailed and transparent methodology for generating high-quality datasets in this context that is easy to extend to other potential new architectures, implementations, and formats that may arise in the field.
- We present an in-depth evaluation of several classification methods in the performance tuning of sparse computational kernels for DL workloads. This includes the assessment of each model from different perspectives: accuracy, latency, and final practical performance (kernel evaluation).
- Encompassing all of this, we propose STuning-DL, an autotuning tool highly coupled to the underlying hardware details *reaching GPU native instruction optimization* by means of template-based sparse linear algebra implementations, and outputting high-performance near-optimal SpMM kernel instances. The tool is able to operate at run-time requiring negligible overhead. To our knowledge, this is the first autotuning project focused on SpMM template-based kernels for DL, covering from high-level format details to assembly-level aspects.
- We evaluate STuning-DL on real-world models from the DLMC dataset and the Llama 2 architecture. STuning-DL achieves speedups of up to 5.42× on the T4-16GB GPU and 3.6× on the A100-40GB GPU compared to the configurations extracted from the NVIDIA sparse libraries cuSparse and cuSparseLt on the same input problems, but requiring a negligible autotuning overhead.

II. BACKGROUND

This section presents a foundational overview of GPU architecture and terminology, while also introducing some background information on autotuning and supervised classification methods. Since all the GPU implementations referenced in this paper are written in CUDA, we adopt the NVIDIA terminology.

A. GRAPHIC PROCESSING UNITS AND SPMM

GPUs are composed by an array of Streaming Multiprocessor (SM) elements that share a L2 cache and a DRAM GPU memory, or Global Memory (GMEM). Each SM is divided in partitions of different processing blocks. All the processing blocks in the same SM share a L1 cache that can be partially used as Shared Memory (SMEM). Each of those processing blocks within an SM is equipped, in a simplified form, with a Register File (RF), a L0 instruction cache, and four types of processing units: Floating-Point Units (FPU), Tensor-Core Units (TCU), Int Units (ALU) and Special Function Units (SFU).

The design of an efficient CUDA kernel highly depends on three aspects: (1) the efficiency of the data movements to the top levels of the memory hierarchy (i.e., GMEM → SMEM → RF), and back (data storage), (2) the usage of the hardware resources during the computation, and (3) the degree of overlap between (1) and (2) to hide memory latency with computation. Block tiling and software pipelining techniques are extensively used not only for this purpose but also for improving data locality.

The execution model hierarchy of a GEMM GPU CUDA kernel can be categorized into three levels of granularity:

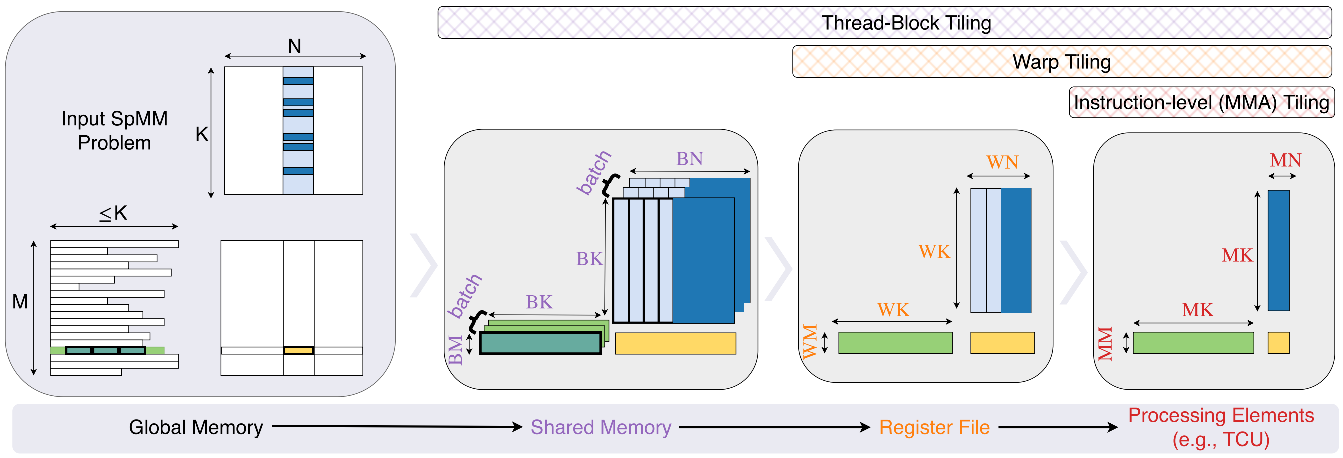

FIGURE 1. **Hierarchical decomposition of an SpMM kernel.**

thread-blocks, warps, and threads. A thread block comprises a collection of warps, with warps serving as the fundamental scheduling unit in CUDA, encompassing 32 threads each. Within a single warp, all threads are dispatched to the same SM and execute instructions synchronously, adhering to a SIMD (Single Instruction, Multiple Data) architecture. Any deviation from this uniformity results in thread divergence, leading to serialized execution. The number of warps accommodated by an SM depends on several factors, including the available resources and the allocation designated for each warp. This parameter is referred to as *occupancy*, and optimizing it is crucial for effectively harnessing GPU resources.

Figure 1 illustrates the *interrelation* between GPU *hardware components* and the software elements in the *CUDA programming model* for a SpMM kernel of dimensions $M \times K \times N$. This hierarchical decomposition begins with the GMEM, where the entire matrices serve as inputs to the SpMM function. Three distinct levels of decomposition are subsequently considered: (1) Thread-Block Tile, assigned to a block of threads, this level of decomposition is stored in SMEM. Then, (2) the Warp Tile, associated with a warp of threads, is stored in RF. Finally, (3) the Instruction Tile (shape), positioned at the bottom of the hierarchy, pertains to the processing unit in use (e.g., TCU). The specific instruction shape adopted depends significantly on the architecture at hand. For instance, on the Ampere architecture, a single Tensor Core instruction can receive a tile of 16 × 16x8 elements, enabling the execution of a Matrix-Matrix Multiplication (MMA) in a single instruction, and thus removing the overhead of fetching and decoding multiple instructions to perform the same computation.

This scenario raises a set of configurable parameters that strongly depend on the specific GPU architecture. The optimal tile sizes for a given input problem and GPU architecture have been studied and optimized for years in dense computation. Some of these tunable parameters have been referenced in Figure 1 for each dimension of an SpMM operation, such as the block tile size ($BM \times BK \times BN$), the warp tile size ($WM \times WK \times WN$), the instruction (MMA) size ($MM \times MK \times MN$), and the batch size (batch). However, autotuning for sparse computations in DL problems, and particularly the relation between such hardware-dependent parameters (some of them at *ptx* code-level [39], an intermediate virtual assembly) with the sparse matrices generated by different weight pruning processes, still remains an open question.

B. NETWORK WEIGHT PRUNING AND SPARSE FORMATS

Weight pruning lays its foundations in the idea that modern large ML models are overparameterized [22]. As a consequence of this observation, this research is focused on the study of the best methods to select which neuron connections to remove, simplifying models' complexity, all while trying to preserve their accuracy. These pruning techniques may operate at different granularity levels.

Structured pruning techniques drop whole components (e.g., filters, layer, heads) of a network, which reduces the architecture complexity while keeping dense the computation [29]. However, this approach is too aggressive in the weight selection for removal and the sparsity level achievable is very limited.

Recent advancements [27], [47] in weight pruning techniques with lower granularities have proven successfully to maintain the original dense accuracy while enabling higher levels of sparsity, opening the path to a boost in the performance. This trend not only contributes to the sparsity level, but also to the emergence of more constrained sparse compression formats, since in this second group of techniques real sparse matrices are generated. This implies transitioning from a dense to a sparse computational paradigm. Depending on the granularity level, we can differentiate between:

Unstructured pruning: is the most flexible approach, which allows dropping weights with no constraints in the distribution of non-zeros. Modern techniques have successfully pruned models to sparsity levels $>$ 90% with minimum to no negative impact on accuracy [47]. Despite this, GPU implementations for unstructured sparsity in DL are usually slower than their dense counterparts in most scenarios because of the irregularity of the sparse matrices, which hinders the efficient usage of the GPU resources [4]. Such sparse matrices are typically represented by well-known sparse compression formats extensively used in other domains such as CSR, COO, ELLPACK, HYB, CSR5, or merged-based CSR formats.

Semi-structural pruning: facilitates the distribution of non-zeros across the sparse matrix in a more uniform manner, consequently benefiting critical aspects of GPU kernels, including inter- and intra-warp load balancing, mitigating thread divergence, enhancing data locality, and optimizing memory transactions with wider memory instructions. Several non-zero patterns have been proposed to achieve the best accuracy-to-performance trade-offs: block-wise pruning [18], [29] selects 2-D blocks of non-zero elements with size $BM \times BK$; vector-wise pruning [4], [7], [32] selects vectors or 1-D blocks of elements of length l; N:M pruning [35] divides the matrix into vectors of consecutive M elements and selects the best N out of M for each one of them; V:N:M pruning [5] divides the matrix into blocks of size $V \times M$, drops a subset of $M - 4$ columns for each group, and selects N elements per row in the preserved columns.

C. AUTOTUNERS AND PREDICTIVE MODELS

Autotuning engines serve as the core of numerous Deep Neural Network (DNN) compilers. These autotuners operate by taking tensor problems as inputs and meticulously navigating vast search spaces to determine the optimal kernel configuration [6], [70]. However, the ideal configuration depends on various low-level device properties, such as L2 and L1 cache sizes, SMEM size, instruction shapes available, and many others. Therefore, these compilers often fall short of achieving the hardware-native performance because they prioritize aspects like portability across different hardware architectures, often overlooking crucial low-level hardware details in their optimization processes [64].

Most BLAS libraries, such as cuBLAS, are written in *sass* code [66] (GPU assembly code) and have been heavily hand-optimized by experts, so they are able to achieve hardware-native performance. However, it is very hard to achieve portable implementations since *sass* code is native to the target GPU architecture. Furthermore, autotuners in this kind of kernels seek the best configuration for a limited set of inputs. For instance, GEMM implementations are often per-default tuned for squared matrices [38], [48]. For that reason, traditional autotuning techniques underperform in ML workloads, where the amount of different input shapes is large.

The expanding diversity of input data in ML has spurred the proliferation of specialized databases (SAMPL) [52], which are designed to store and reuse tuning logs. However, modern ML models exhibit growing dynamism, not only dynamic data structures but also variable shapes, rendering traditional caching methods less effective. Additionally, maintaining these databases is very expensive.

With the aim of dealing with such dynamism, a new trend of autotuners based on Predictive Models is emerging [2], [28], [53], [55] which tries to balance flexibility and hardware-native performance in core routines such as GEMMs and convolutions. The idea is to use ML techniques to *model performance*, and replace hand-written rules by supervised classification techniques. Some of these supervised methods previously used for dense computation include [28], [46]:

Decision trees (DT) [51] are a non-parametric supervised learning method employed for regression and classification tasks. DTs try to predict the value of a target variable by learning simple decision rules, which can be easily interpreted as a chained set of `if-then-else` statements, inferred from the input data features. However, DTs can be unstable and may produce completely different trees with small data variations. This problem can be mitigated using DTs with an ensemble.

Random Forest (RF) [21] is an ensemble method of DTs that can provide better generalization and reduced overfitting. It is a meta estimator that fits many independent DTs on various sub-samples of the dataset, and then uses averaging predictions (for regression) or takes majority vote (for classification) to provide the result.

Gradient Tree Boosting (GTB) [67] is another ensemble method that builds DTs sequentially. It makes use of gradient descent to improve the model by adding DTs that focus on the errors made by the previous ones.

Naive Bayesian Classifier (NBC) [49] is a supervised learning method based on Bayes' theorem. It represents the simplest variant of a Bayesian network and assumes that, given a class variable, all the features are strongly (naively) independent. Despite being usually false, this straightforward approach can generate simple but effective models.

Logistic Regression (LoR) [34], in this project, serves as a linear classification model, which predicts the probability of a given input belonging to a particular class. This prediction is made by assessing the input's features through a logistic function.

K-Nearest Neighbours (KNN) [44] finds the K training samples that are closest in distance to the new point, and predict its label from them. Neighbors-based methods are also referred to as non-generalizing learning methods, since they just remember the training points.

Support Vector Machine (SVM) [9] is a supervised learning method that divides the data by drawing hyperplanes in the feature space. SVMs represent an efficient solution for high dimensional spaces. However, if the number of features is much greater than the number of samples, it can lead to overfitting.

Multi-Layer Perceptron (MLP) [45] is a fully connected feed-forward neural network formed by: (1) an input layer, (2) at least one hidden layer, and (3) one output layer.

These methods have already demonstrated their effectiveness in tackling dense computational BLAS tasks [28]. Remarkably, Predictive Models have demonstrated near-optimal performance in convolution. Furthermore, when applied to GEMM tasks, they have exhibited outstanding performance gains, outpacing traditional hand-tuned approaches (i.e., cuBLAS) by up to 3×. Recent studies [15], [37] have also applied these learning methods in sparse computation for performance modeling and for selecting the most suitable sparse format representation. However, sparse compression format represents only a high-level software aspect. Many other variables which are highly coupled to the underlying hardware remain unaccounted for, making it challenging to achieve hardware-native performance. Furthermore, such studies are generally limited to SpMV, and they focus on traditional HPC scientific computation, which is very different from the DL one. As such, Predictive Models hold significant promise as an unexplored avenue for addressing sparse BLAS problems in DL but not limited to software level aspects, tuning also hardware-dependent (*ptx*-level) details instead, in the pursue of such hardware-native performance.

III. CHALLENGES AND OPPORTUNITIES IN SPARSE COMPUTATION FOR DL

In this section, we identify existing autotuning techniques applied to general sparse computation, and their effectiveness on ML workloads. Table 1 provides a comparative analysis of SOTA sparse libraries and third-party implementations with support for fp16 precision. Alternative implementations for other data representations have not been considered in this analysis [61], [71].

In the following section, we will analyze the impact of autotuning on libraries with this capability, but focusing on DL-like problems. We will center this study on semi-structured methods, since they achieve the best accuracy-to-performance trade-off. First, we focus on 2-D block sparsity. Then, we will delve into 2:4 sparsity. The experiments have been conducted on an RTX 3090 NVIDIA GPU.

A. AUTOTUNING ALGORITHMS ON 2-D BLOCK-STRUCTURED SPARSITY

Block-wise (2-D) sparsity represents the most regular semi-structured sparse format. With larger block sizes (e.g. 32, 64, 128), key properties of an efficient kernel, such as data coalescence, intra-warp load balance, and no thread divergence, can be guaranteed. Consequently, aside from the overhead incurred by loading sparse metadata and calculating offsets, this sparse format transforms the sparse product into a pretty close version of a GEMM, but with a smaller input. For that reason, several attempts has been made in the community to extend the autotuning rules to this format. Consequently, various dense libraries and tensor compilers have incorporated support for 2-D block-wise sparsity format, such as AutoTVM.

TABLE 1. Comparison of sparse BLAS libraries with support for fp16 precision. "DL-sparse-friendly:" Has the library been designed for sparse ML problems?; "acc.:" Is the sparse granularity flexible or too restrictive to preserve the accuracy (energy) of the dense model? "⚡:" very flexible, "🔋:" flexible, "🪫:" restrictive, "✖:" very restrictive; "perf.:" Is the sparse granularity favorable for GPU execution or is it too irregular to achieve good performance? "⚡:" high performance, "🔋:" good performance, "🪫:" slightly better than dense, "✖:" mostly worse than dense; "templated:" Is the sparse implementation open-source C++ template-based?; "autotuner:" Does the sparse library offer an autotuner that allows the configuration of the implementation to different platforms? "hand-tuned:" traditional autotuner based on hand-tuned vendor heuristics, "code gen.:" Target-Specific Code Generation, ML/tensor-compilers; "GS:" Grid-Search, `cusparseLtMatmulSearch` function evaluates all available algorithms and selects the fastest one [40]; *: implemented algorithms: XGBTuner, GATuner, RandomTuner, GridSearchTuner.

Library	DL-sparse-friendly	Sparsity Format			templated	autotuner
		granularity	acc.	perf.		
cuSparse [41]	👎	unstructured	⚡	✖	✘	✔ (hand-tuned)
	👍	2-D block	✖	🔋	✘	✔ (hand-tuned)
Sputnik [16]	👍	unstructured	⚡	✖	✘	✘
SparseTIR [68]	👍	unstructured	⚡	✖	✘	✔ (code gen.)
	👍	2-D block	✖	🔋	✘	✔ (code gen.)
Triton [57]	👍	2-D block	✖	🪫	✘	✔ (code gen.)
AutoTVM [6]	👎	2-D block	✖	🔋	✘	✔*
	👎	1-D block	🪫	🪫	✘	✔*
cuSparseLt [40]	👍	2:4	🔋	🪫	✘	✔ (hand-tuned + GS)
Shfl-BW [24]	👍	1-D block	🪫	🔋	✔	✘
VENOM [5]	👍	V:N:M	🔋	⚡	✔	✘

Figure 2 shows the speedup of cuSparse and AutoTVM using the BlockedEll format w.r.t. cuBLAS on 2-D block-structured sparsity. The matrices have been grouped in 6 sparsity intervals. The red crosses represent the average values for a given sparsity level window. The "ideal" bars reflect the theoretical speedup expected considering the arithmetic count reduction for each sparsity level. The sparse matrices were extracted from the PruneBERT model with block-wise sparsity [1] using 32 × 32 blocks and an average weight sparsity of 92.5% on linear-layers. Overall, PruneBERT contains 28.2% of the original weights. The autotuning process of AutoTVM to select the best configuration for each linear layer in PruneBERT took around 24 hours to finish.

The results show that, on average, AutoTVM performs slightly better than cuSparse for sparsity levels below 70%, and mostly equal for levels between 70% and 75%. However, cuSparse, which has been hand-tuned for sparse computation, is faster on average terms in moderate to high sparsity ratios ($\geq$ 75%). Overall, cuSparse can be up to 8× faster than AutoTVM. Autotuning techniques in AutoTVM seem to behave better in denser cases where the sparse product is closer to a dense GEMM. However, they important limitations to scale the performance with the sparsity, with almost a constant speedup of around 1.8× across the whole sparsity range. These results could be due to autotuning techniques not being explicitly designed for sparse computing, but inherited from its dense analogous version instead. This is why we classified AutoTVM as a non

[1] https://huggingface.co/madlag/bert-base-uncased-squad1.1-block-sparse-0.07-v1

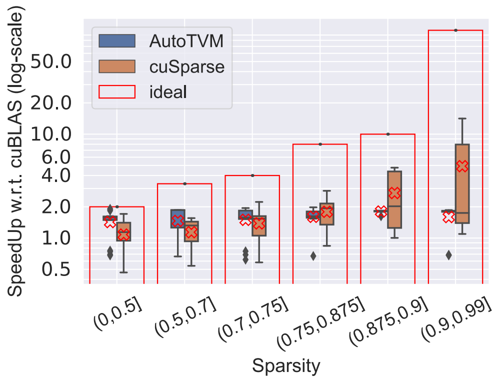

FIGURE 2. Speedup on Prune-BERT. The x-axis groups the sparse matrices extracted this model according to different levels of sparsity.

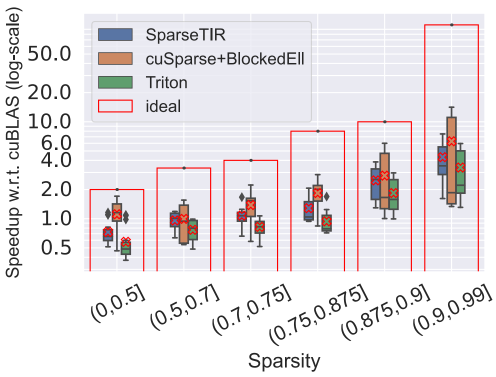

FIGURE 3. Speedup on Prune-BERT. The x-axis groups the sparse matrices extracted this model according to different levels of sparsity.

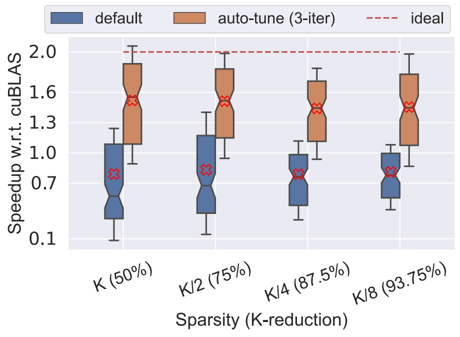

FIGURE 4. Speedup of cuSparseLt w.r.t. cuBLAS using default kernel configurations and after running the `cusparseLtMatmulSearch` autotuning function for 3 iterations.

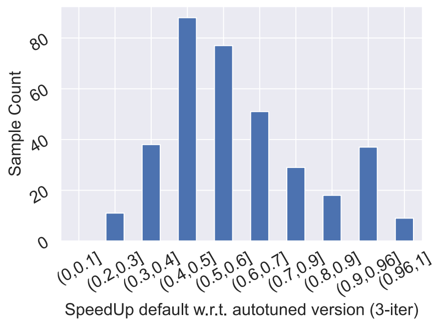

FIGURE 5. cuSparseLt speedup of the default configuration over the autotuned one after 3 iterations.

DL-friendly sparse library in Table 1. On the contrary, while cuSparse starts with a lower performance, it scales with the sparsity level, which is the behavior expected.

Overall, block-wise pruning can be too restrictive, and it has been demonstrated to influence very negatively the accuracy when block-size > 8. As an example, in the previous PruneBERT model with block-size 32, 106 attention heads out of 144 have been completely removed, and the F1 score of the model is reduced from 88.5 to 81.36 on SQuAD v1. Despite the performance-oriented nature of this sparsity format, the speedups achieved for both AutoTVM and cuSparse are still far from the ideal.

1) PARTICULAR CASE-STUDY: ML COMPILERS

Figure 3, shows a performance comparison of sparseTIR and cuSparse using the BlockedEll format w.r.t. cuBLAS. The matrices and model configuration in this evaluation belong also to the PruneBERT model.

Tensor compilers (i.e., SparseTIR, Triton), on average, underperform when compared to the vendor-tuned libraries for sparse computation (i.e., cuSparse) on 2D-block sparse matrices. In particular, these tensor compilers achieve, on average, a $45-70\%$ of the performance of cuSparse using this format for sparsity levels below 90%. This performance disparity comes at the cost of the flexibility provided by these tools. However, in sparse computation, this trade-off is crucial, as it can determine whether the computation is slower or faster than dense computations, so that the decision of pruning a model may not yield performance benefits. These results are especially concerning when considering that 70% of the PruneBERT sparse matrices exhibit sparsity levels below 90%. Furthermore, sparseTIR and Triton start to outperform the dense counterpart version for sparsity levels $\geq 85\%$, and $\geq 87.5\%$, respectively, which is already considered high sparsity in DL.

Autotuning for more flexible semi-structured sparsity formats than 2-D blocks, such as N:M, is still an open-question on this kind of tools. This represents a promising avenue aiming to achieve better accuracy-to-performance trade-offs than 2-D block-wise pruning.

B. AUTOTUNING ON CUSPARSELT: 2:4 SPARSE KERNELS

Figure 4 shows the performance of NVIDIA cuSparseLt on matrices extracted from models in the DLMC dataset [17], and BERT [12]. This library provides not only default kernel configurations pre-defined by experts, but also an autotuning function (`cusparseLtMatmulSearch`) that tries to find more suitable kernel configurations for an specific input problem to improve the performance.

Figure 4 shows the performance of cuSparseLt w.r.t cuBLAS using (1) the baseline configuration according to NVIDIA heuristics, and (2) after 3 autotuning iterations

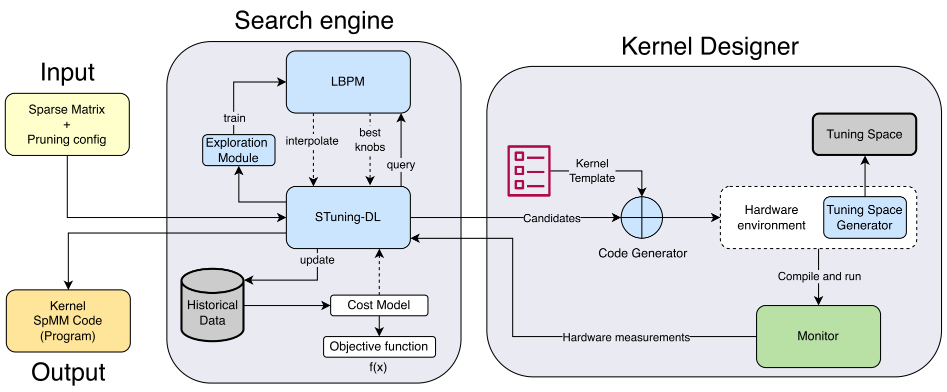

FIGURE 6. Overview of STuning-DL.

(kernel launches per algorithm) of the `cusparseLtMatmulSearch` function. Since cuSparseLt only supports 50% sparsity, we have shrank the K dimension of the original matrices to simulate smaller problem sizes (higher sparsity levels) and see how this aspect affects the results obtained (x-axis). Remind that cuSparseLt peak performance is $2\times$. A few matrices slightly overcome this limit, probably because the counterpart dense implementation has not been perfectly optimized for some problem sizes. Overall, the default kernel configuration achieves an average speedup of $0.75\times$ (slower than dense), while the same kernels after an autotuning process achieve on average around $1.50\times$.

Figure 5, shows the frequency of the relative performances obtained by (1) w.r.t. (2) on these matrices. As we can see, NVIDIA default heuristics only select the best algorithm (relative speedup 0.97-1.0) in 5% of the instances in the dataset, while in most cases the default configuration is $2 - 2.5\times$ slower.

We can conclude that the performance of a sparse kernel is heavily influenced by the configuration selected. However, as we mentioned in the introduction, the diversity of matrix sizes in ML problems is vast, so this process needs to be executed many times for each network architecture. These conclusions raise a major concern in `cusparseLtMatmulSearch`, since it evaluates all available algorithms and selects the fastest one [40] for each input problem. In that sense, we have observed that the average time spent on this function increases with the problem size ($M \times N \times K$). Overall, increasing the FLOPS, increases the search time, and with the datasets used in this evaluation, the search-process took up to 175 minutes, around 3 hours, for a single matrix. Furthermore, this search process does not guarantee the best performance. As Figure 4 depicts, some matrices have speedups $\leq 1\times$, which means that they are slower than the dense counterpart version.

In this context, Table 2 shows the set of configurations that the autotuner has selected in all the matrices of the dataset. In total, 4 combinations are considered for the whole range of problem sizes. This clearly represents a low tunability and a limitation in ML workloads, being possibly one of the reasons why some matrices are still far from the hardware-native performance.

TABLE 2. CuSparseLt's considered configurations (cfg) within their tuning space for the evaluated matrices.

Tunable params.	cfg1	cfg2	cfg3	cfg4
BM	128	128	256	128
BN	64	64	128	128
BK	64	64	64	64
WM	64	64	64	64
WN	32	32	64	64
WK	64	64	64	64
MM	16	16	16	16
MN	8	8	8	8
MK	32	32	32	32
Batch	2	5	2	2

C. EMERGING TREND: TEMPLATED LIBRARIES

Existing solutions lack enough flexibility to correctly adapt to the variability of key aspects such as the sparsity level, the sparsity format, or the input problem shape. However, in the same way that happened in dense computation [56], there is an emerging trend of template-based implementations for DL routines. A new wave of third-party kernels has surfaced, providing implementations for different sparse formats, such as Shfl-BW [24] and VENOM [5]. Such solutions offer as configurable parameters the whole set of variables described in Figure 1 such as Thread-Block, Warp, and MMA tile shapes, as well as the Batch size. This allows to greatly adapt the kernel implementations to different input problems and hardware platforms, reaching *ptx*-code level. However, such implementations lack an autotuner.

Furthermore, several open questions still remain if we analyze and compare this problem with the new template-based approaches proposed for dense computation [64]. For instance, new features must be added to the tuning process, such as the sparsity level, the sparsity pattern, and other optional variables concerning the specific sparse format used.

TABLE 3. Definitions and notations.

	Symbol	Definition
		General notations
	$\mathbb{IS}$	Task Parameter Input Space
	$\mathbb{TS}$	Tuning Parameter Space
	α	size of $\mathbb{IS}$
	β	size of $\mathbb{TS}$
	$\mathbb{O}$	The Oracle, an ideal estimator
Use-case		$C = \alpha \cdot A \cdot B + \beta \cdot C + bias$ s.t. $A \in \mathbb{C}^{m \times k}, B \in \mathbb{C}^{k \times n}$, $C \in \mathbb{C}^{m \times n}$, where A is a sparse matrix and α, β are scalars
		Task Parameter $\mathbb{I}$nput $\mathbb{S}$pace
Features	M	Number of rows of the SpMM LHS operand (A)
Features	K	SpMM Inner dimension
Features	N	Number of columns of the SpMM RHS operand (B)
Features	d	Density of A
Features	l	Vector length (only if A has vector-wise format)
Features	V:N:M	V length (only if A has VENOM format)
		$\mathbb{T}$uning Parameter $\mathbb{S}$pace
Classes	BM, BK, BN	Thread-Block tile size for M, K, N dimensions
Classes	WM, WK, WN	Warp tile size for M, K, N dimensions
Classes	MM, MK, MN	MMA (instruction) shape for M, K, N dimensions
Classes	Batch	Degree of memory software pipelining

In subsequent sections, we will provide a comprehensive exploration of these differences, delving into their intricacies and challenges.

IV. STUNING-DL: A SPARSE KERNEL AUTOTUNING TOOL FOR DEEP LEARNING

Figure 6 contains an overview of the STuning-DL architecture, which comprises two main components: the **Search Engine** and the **Kernel Designer**.

The Learning-Based Predictive Model (LBPM) module is the core component of the Search Engine, which works as a predictor, traversing the Tuning Space (Section IV-B) and providing the best configuration (knobs) as response to a query for a given input problem. At this point, a hardware-aware component known as the Tuning Space Generator (TSG) is responsible for generating kernel configurations that maximize hardware utilization, co-designed for a specific GPU architecture and sparse implementation (Section IV-B1). These configurations together will constitute our Tuning Space. The LBPM module relies on supervised learning techniques for classification tasks to find the best solution based on a predefined objective function (Section IV-A). The LBPM is trained using historical data from past executions, which represents the Input Space (Section IV-C) for our classifier. This historical data is prepared for training by an intermediate Exploration Module (Section IV-D), which includes a pre-processing stage to enhance data regularity for classification. Finally, the Kernel Designer component takes the configurations predicted and provided by the classifier as input, and uses them to instantiate template-based implementations with the corresponding settings.

A. PERFORMANCE TUNING THROUGH CLASSIFICATION METHODS

Performance modeling SpMM kernels for DL problems effectively remains an open question due to the various limitations that have hindered its progress, as we have seen previously. Consequently, conventional autotuning approaches or manual optimization methods are not effective for DL workloads because of the multidimensionality and variability of the input data. In this paper we tackle this problem by studying various supervised classification techniques previously proven effective in modeling the performance of other linear algebra implementations, including GEMMs and convolutions [28].

Table 3 shows a summary of the definitions and notations used in this paper. In order to describe the Performance as a Classification Task approach, let $\mathbb{IS}$ be the Task Parameter Input Space containing all the possible input problems or tasks to the application, while $\mathbb{TS}$ is the Tuning Parameter Space containing all the parameter configurations to be optimized. We call α the size of $\mathbb{IS}$, and β the size of $\mathbb{TS}$.

Let $t \in \mathbb{TS}$ be a multidimensional tuning parameter configuration that is a solution to a particular input task $i \in \mathbb{IS}$, the objective function is defined as $f_t : \mathbb{IS} \rightarrow \mathbb{R}$. Hence, the target is the maximization of the objective function, so we have to find, for each $i \in \mathbb{IS}$, the t that guarantees:

$$arg\ max_{t \in \mathbb{TS}} f_t(i) \qquad (1)$$

which, in this case-study, means the best solution in terms of performance. Hence, a classifier will encompass the mapping from multidimensional input description i to a multidimensional solution t. From this point on, we will refer to the "Oracle" as an ideal estimator that is able to select the fastest configurations for every $i \in \mathbb{IS}$. Furthermore, as Table 3 shows, we will allude to the set of variables that define each $i \in \mathbb{IS}$ as (task) *features*, and to the set of tunable parameters for each $t \in \mathbb{PS}$ as *classes*.

B. TUNING PARAMETER SPACE ($\mathbb{TS}$)

The definition of $\mathbb{TS}$ is critical in order to ensure the adaptability of our kernels to different architectures and to the high variability of $\mathbb{IS}$ in DL problems. Hence, the quality of $\mathbb{TS}$ will be crucial in the final performance. Note that we can obtain a very accurate classifier in the selection of the best configuration within the scope of $\mathbb{TS}$, but such configuration can be far from the Oracle if $\mathbb{TS}$ was not carefully defined.

As we saw in previous sections, one important limitation of existing solutions is that their $\mathbb{TS}$ is restricted to a small set of classes, and each class is also limited to a small range of possible values (see Table 2). Taking this into consideration, we designed $\mathbb{TS}$ considering the casuistry of input tasks $i \in \mathbb{IS}$ in DL problems. The set of possible values for some parameters, such as the instruction shape, is architecture-dependent, and is constrained to a small set of values. However, the selection of the best configuration for the remaining parameters is both highly tunable and sensitive. For this reason, it tends to be hand-tuned by experts. For

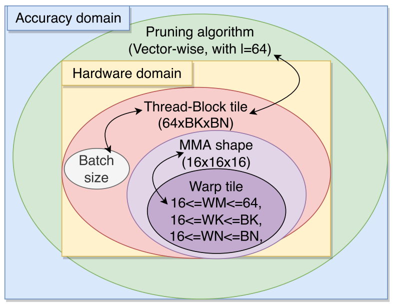

FIGURE 7. Overview of $\mathbb{TS}$ classes and their interrelationships for a vector-wise input sparse matrix with vector-length $l = 64$ format.

instance, returning to Figure 1, using a too big Thread-Block tile ($BM \times BK \times BN$) on a small input problem ($M \times K \times N$) can cause blocks of threads to compute, in overall terms, a too big sub-problem, damaging the Thread-Level-Parallelism. And conversely, a too small block size can hinder thread blocks from taking advantage of data locality, being thus unable to hide memory latency with computation.

Figure 7, illustrates the interrelationships among $\mathbb{TS}$ classes for a vector-wise pruned sparse input matrix with a vector length $l = 64$. The diagram delineates two domains, each represented by a different box, which exert mutual influence upon one another. In this context, a domain refers to the area each tunable parameter is related to. The first domain (blue), referred to as the *accuracy domain*, pertains to the model pruning phase, while the second, the *hardware domain* (yellow), concerns itself with the execution of the computational kernels for a given sparse model on a specific platform. The classes stemming from each of these environments are visually represented as ellipses within the graphical depiction.

In this example, the outer tuning class of the hardware domain, associated to the Thread-Block tile size, will be influenced by the pruning technique and the configuration used to generate the sparse input matrices during the model sparsification process. Regarding the Warp tile size, it has to be defined in terms of the Thread-Block tile size and the Instruction (MMA) shape selected. Finally, the number of memory pre-fetching stages (batch) is also dependent on the Thread-Block tile size. Overall, all the range of possible values will be limited by the hardware constraints of the GPU. Some of these constraints are the SMEM size, the number of concurrent warps per SM, the number of threads per SM, and the number of registers per thread.

1) TUNING SPACE GENERATOR

The previous findings inspired us to create an integrated tool that, given a GPU architecture and template-based implementation, generates a subset of $\mathbb{TS}$, being the objective function the maximization of the GPU occupancy. Occupancy represents the ratio of active warps per multiprocessor to the maximum possible number of active warps. It is important to note that a higher kernel occupancy does not always imply a higher performance. However, low levels of occupancy invariably lead to a detrimental impact on the mitigation of memory latency, and overall, a degradation in the final performance. This effect is particularly critical in sparse computational problems because of their memory-bound nature.

TABLE 4. Values considered for each class.

Tunable params.	values
BM	{16,32,64,128}
BN	{16,32,64,128,256}
BK	{16,32,64}
Warp tile size	{16,32,64}
MMA shape	{16-by-16-by-16, 16-by-8-by-16, 16-by-8-by-32}
Batch size	{2,3,4}

This component draws inspiration from [42], enabling the computation of multiprocessor occupancy for a GPU, based on a specified CUDA kernel configuration. This approximation serves as an initial filter to *prune the* $\mathbb{TS}$ *search-space*, not only by selecting just the best configurations in terms of occupancy but also remove combinations that exceed the GPU resources. Thus, laying the groundwork for subsequent autotuning steps by significantly shrinking the number of configurations to be considered, profiled, and evaluated. Moreover, it plays a pivotal role in the development of a high-quality tuning space, a critical aspect, as previously emphasized.

Following these principles, the list of configurations will be generated trying to achieve high GPU occupancy. Table 4 shows the values supported for each parameter in $\mathbb{TS}$.

As an example, for a vector-wise-based kernel [24] and one GPU architecture (i.e., A100-40GB), this is translated into $\beta \approx 400$, which represents a huge number of classes to tune. A detailed study on the impact of the $\mathbb{TS}$ design will be made in Section V-B.

C. TASK PARAMETER INPUT SPACE ($\mathbb{IS}$)

Another crucial aspect that must be well defined in order to meaningfully evaluate our approach is the data that will be used as a subset of $\mathbb{IS}$ to train our models. As we saw, $\mathbb{TS}$ can be large due to all the variables that must be considered in sparse problems for DL. Furthermore, supervised techniques require datasets that map the training samples in $\mathbb{IS}$ to the optimal configurations in $\mathbb{TS}$. As a result, in order to build the baseline dataset, for each input $i \in \mathbb{IS}$ used in the training, $f_t(i)$ must be benchmarked for every $t \in \mathbb{TS}$ on each target architecture. In that sense, the size of the dataset generated will be crucial in the final performance of our classifier. Furthermore, note that for our classifier to be aware of every possible class t defined in $\mathbb{TS}$, there must be at least one input i in the training set where t shows the best performance. In these terms, it is very important how we divide the training and the validation set, since those classes that are only present in the validation, will be unknown by the classifier. An in-depth analysis of the impact that growing the dataset has in

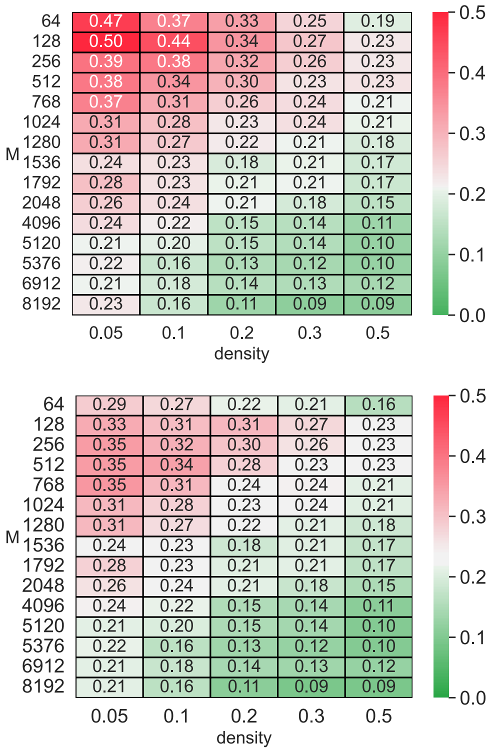

FIGURE 8. Heatmap number of configurations for sparse matrices grouped by density and *M*. On top, the search-space before normalization, and on the bottom, after normalization.

the LBPMs performance, and how it is partitioned, will be presented in Section V-A.

D. EXPLORATION MODULE: NORMALIZING THE SEARCH SPACE

Solving the Performance as a Classification Task problem with the aforementioned scenario is extremely difficult because of the large size of $\mathbb{TS}$. Such a vast number of classes allows having very fine-grained kernel configurations and adapting to the wide range of possible inputs. However, it also promotes the proliferation of multiple "best configurations", that is, configurations that achieve the best results in at least one scenario.

Figure 8, top side, shows this effect on a set of $\sim 600k$ input matrices sharing the same density (x-axis) and the M-dimension (y-axis), which is tightly-coupled to the pruning algorithm configuration, as we have already seen. In this example, the total number of configurations that were selected as the best one at least in one input problem was $\phi \approx 200$. The figures in the cells stand for the number of best configurations in at least one scenario for an specific M and density, normalized to ϕ.

First of all, the denser and the larger the problem, the easier it is to select the correct configuration. This perfectly matches with the conclusions drawn in Section III, demonstrating why existing autotuners work well under these circumstances. However, the sparser and smaller, the more unstable it turns, and thus the more difficult to predict. This way, for problems with density $d = 0.05$, and $M = 64$, 47% of ϕ (i.e., ~ 100) has been selected as the best configuration at least once.

In view of this, we implemented a normalization algorithm which aims to enhance data regularity for classification, and it will be used as a pre-processing stage to the LBPMs training. To that end, it replaces the ideal configurations for alternative ones with a certain relative configurable performance but, in contrast with the ideal, it preserve the state of best alternative configuration across more samples of IS, thus enhancing its regularity. Figure 8, lower side, shows $\mathbb{TS}$ after a normalization considering a threshold of $\geq 0.98\times$ w.r.t. the Oracle. As a consequence, the total number of configurations is reduced to $\phi \approx 120$.

The pseudocode of the normalization algorithm is shown in Algorithm 1. First, it computes, for each input problem, the relative speedup of each $t \in \mathbb{TS}$ w.r.t. the Oracle. Then, it selects the configurations that have achieved the minimum (threshold) relative speedup we configured (i.e., in our case *threshold*=$0.98\times$), and computes the popularity of each t, that is, the number of times it is repeated. Finally, it selects the most popular configuration and assigns it to those inputs $i \in \mathbb{TS}$ where it has fulfilled the threshold condition. This last step is repeated until all the inputs have been assigned a kernel configuration.

This pre-processing had a big impact on the accuracy of our classifiers, with no to little change in performance ($\leq 0.02x$). If more homogeneity is needed, the relative performance selected ($\geq 0.98\times$) can be reduced, but this can be in detriment of the real speedup finally achieved, so the desired trade-off must be found depending on the application.

Algorithm 1 : $\mathbb{TS}$ Normalization Pseudocode. The Word 'by' Means 'grouped By'

```
1:  relative_sp = compute_relative($\mathbb{O}$.time/$\mathbb{TS}$.time, by=$\mathbb{TS}$.classes)
2:  $\mathbb{TS}$_norm = $\mathbb{TS}$.norm(relative_sp, threshold, by=$\mathbb{TS}$.classes)
3:  $\mathbb{TS}$_norm.compute_popularity()          ▷ repetitions per configuration
4:
5:  result = { }
6:  while $\mathbb{TS}$_norm $\neq$ { } do
7:      sort($\mathbb{TS}$_norm, by='popularity')
8:      cfg = $\mathbb{TS}$_norm.head()             ▷ most repeated configuration
9:      r = filter($\mathbb{TS}$_norm, cfg)         ▷ Select inputs $i \in \mathbb{IS}$ matching cfg
10:     result = concat(result, r)                  ▷ Update result
11:     $\mathbb{TS}$_norm.update(r)                ▷ Remove inputs $i \in \mathbb{IS}$ selected
12: end while
```

V. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our method, we carried our experiments in two different NVIDIA GPUs from different architectures, the NVIDIA T4-16GB from the Turing architecture, and the NVIDIA A100-40GB from Ampere. We build our $\mathbb{IS}$ as described in Section IV-C. For every kernel instantiated with $t \in \mathbb{TS}$, we conducted 50 execution iterations, locking the GPU memory and SM clock frequency to ensure consistency in the measurements. On top of that, we iterated this procedure 5 times for each data point $i \in \mathbb{IS}$, and collected the average results for each input problem.

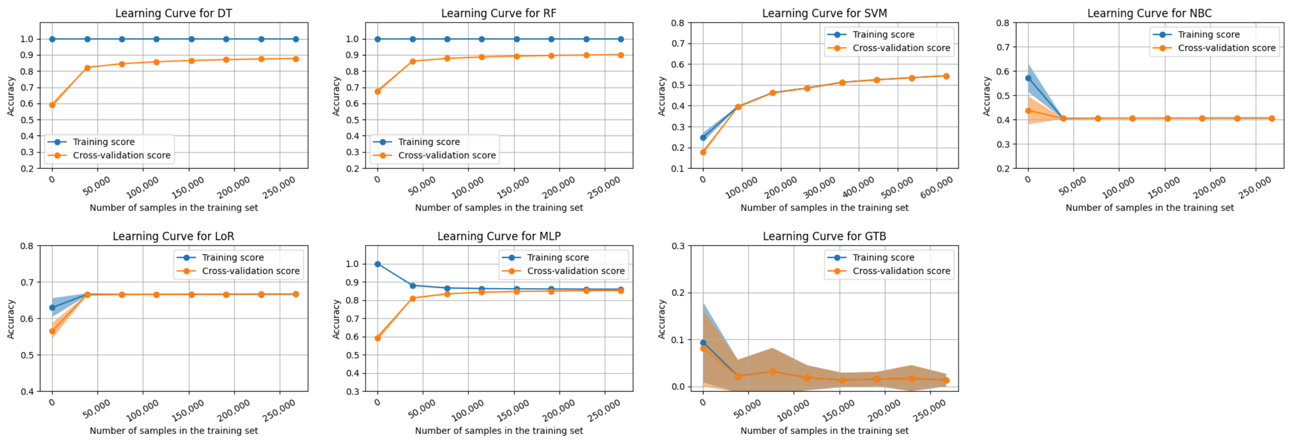

FIGURE 9. LBPMs performance 267,541 training points from *go*2 dataset, except SVM, with 623,848. The coloured regions illustrate the variability observed across 32 evaluations for every data point, while the line indicating the corresponding average value. DT: Decision Tree; RF: Random Forest; GTB: Gradient Tree Boosting; NBC: Naive Bayesian Classifier; LoR: Logistic Regression; KNN: K-Nearest Neighbours; SVM: Support Vector Machine; MLP: Multi-Layer Perceptron.

In order to constitute our baseline $\mathbb{IS}$ history data, we defined three different datasets, one formed by real-world matrices, and two from synthetic ones, inspired in previous works for dense computation with LBPMs [28]:

1) **Real-world** dataset: sparse matrices with shapes from popular DL architectures, such as: Convolutional Neural Networks (e.g., MobileNetV1 [23], AlexNet [26], GoogleNet [54]) and transformers (e.g., BERT$_{base}$, BERT$_{large}$ [12]).
2) **Power of two** (*po2*): sparse problems with shapes ($M \times N \times K$) considering all the combinations of values that are powers of 2 ranging from 64 to 8,192.
3) **Grid of two** (*go2*): sparse problems with shapes ($M \times N \times K$) considering all the combinations of values ranging from 256 to 8,192 with a step of 256, plus 64 and 128. This set of matrices is the largest and it contains 786,080 samples.

Note that the shapes included in the synthetic datasets have been chosen based on their prevalence in Machine Learning architectures. As the synthetic matrices are intended for benchmarking purposes, they have been populated with random values. For each of these inputs sets, different density levels ($d \in \{0.05, 0.10, 0.20, 0.30, 0.50, 0.09\}$), and pruning configurations of vector-wise [24] ($l \in \{16, 32, 64, 128\}$) and VENOM [5] ($V \in \{32, 64, 128\}$) formats were considered.

A. LEARNING-BASED PREDICTIVE MODELS EVALUATION

In order to evaluate the selected LBPMs, first, we partitioned $\mathbb{IS}$ into training, validation and test sets to train, tune the hyperparameters, and evaluate the selected models, respectively. All the entries $i \in \mathbb{IS}$ belonging to the test set have been removed from the train and validation sets, so that model evaluation has been conducted on unseen inputs. The accuracy evaluation (e.g., learning curves) has been performed through 25-fold cross-validation. This method is commonly used to assess model's generalization performance. The accuracy of each LBPM is calculated w.r.t. the Oracle which always selects the best configuration according to the objective function. It is important to highlight that achieving a correct prediction in this context entails identifying the exact optimal configuration, a notably ambitious objective. However, it is worth noting that in many instances, even selecting a configuration that deviates slightly from the optimal choice results in minimal to negligible performance impact, as we will see in the following sections.

Figure 9 shows the learning curves for each model. At the beginning, the training score starts at a higher accuracy in comparison with the cross-validation score because the number of samples is low, and it is easier to memorize a small dataset. As more samples are added to the training set, the training score generally starts to decrease because it becomes more challenging to fit a larger dataset. The cross-validation score has the opposite behavior. Generally, it starts at a lower accuracy, because models are not able to generalize well to unseen data when there is only a small amount of samples to learn from. But when the sample size increases, the curve also does, indicating that the model's ability to generalize is improving with the training set size. The first conclusion here for all the models considered is that $\approx$ 200,000 points seems to be enough to make them converge, and none of them suffers from underfitting, so adding more samples will not significantly improve the accuracy obtained.

More specifically, DT, RF, and MLP converge to high accuracy on predicting new data (around $0.85 - 0.9$), and the distance between training and cross-validation curves is small. However, we should remind that the number of classes in $\mathbb{TS}$ is much larger than the number of features in $\mathbb{IS}$. The complexity of this classification problem is reflected in NBC, LoR, and SVM classifiers, demonstrating that there is not a simple way of cutting the input feature space to properly separate the classes. Regarding NBC, LoR, and GTB models, there are no performance improvements by increasing the number of samples beyond 50,000. On the opposite, SVM

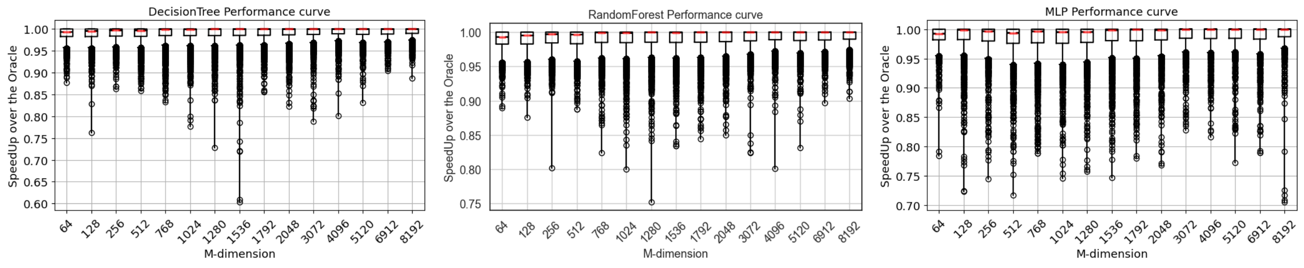

FIGURE 10. LBPMs performance 267,541 training points from *go*2 dataset grouped by the M-dimension of the product ($M \times K \times N$).

shows a slight increase in the accuracy achieved with the number of samples, so we decided to increment them to 623,848. However, both training and cross-validation scores plateau around 0.55, so adding more data will not improve its performance significantly beyond this point, considering the training time and also the cost of generating such amount of samples.

This study presents a significant finding by contrasting the requisite number of samples for classifier convergence with those reported in previous dense computation research for linear algebra kernels [28]. Whereas prior studies in dense computation demonstrated that 4,000 samples were sufficient to achieve convergence in models like DT, RF, MLP, our investigation reveals a starkly different scenario. There is a noticeable gap between training and cross-validation scores until the sample size reaches 50,000, were both start to approach. *This substantial disparity underscores the heightened complexity inherent to this classification challenge*.

Overall, in terms of accuracy, DT, RF and MLP seem the most promising alternatives for this case study, and thus they are the ones selected from this point on.

1) MICROBENCHMARKS

The next step is to evaluate how the accuracy numbers previously described translate into computational time, but also the *impact of misclassifications*, that is, the cost of a wrong kernel configuration selection. Once again, this evaluation has been carried out on inputs from the test set, which the models have not seen before. The boxplot in Figure 10 shows, for the three selected models, the relative speedup, normalized between $0 \sim 1$, of the configuration predicted for each classifier against the performance using the configuration selected by the Oracle. The results in this plot are grouped by the M-dimension of the problem, the outermost of the LHS operand. For the sake of readability, we have represented a subset of the results for the M-dimension. However, it is important to note that results for other values of M exhibit similar trends. The overall result numbers of the whole dataset will be presented in Tables 6, and 7. For each M-value, we are evaluating matrices with all the combinations of different N, K, densities d, and pruning configurations (l, in the case of vector-wise pruning, and V, in the case of VENOM).

As we can see, on average terms (red line), the three methods provide good results, achieving relative speedups very close to $1\times$ (the Oracle). The interquartile range (box extremes) is generally between $\sim 0.98\text{-}1\times$. The minimum of the boxplot (lower whisker) for both DT and RF is always $> 0.95\times$, while for MLP it is sometimes in the range $0.90 \sim 0.95\times$. The outliers are anomalous single data points (matrices) represented by circles that are out of the main probability distribution of the results. We can see that, for both DT and RF, the outliers are generally above 85% relative performance. That would mean that even for those worst case scenarios, we would be able to achieve 85% of the peak performance. For MLP, in general terms, the outliers are above $0.80\times$. The number of samples that fall bellow such limits represent $< 1\%$ of the total number of samples for the three classifiers.

These results are in accordance with the previous accuracy numbers. Overall, the three methods provide good predictions and, as a consequence, good relative performance numbers. However, MLP is slightly less accurate than DT and RF, which translates into slightly worse relative speedups and more unstability, which can result in a higher impact of missclassification.

2) RELATIVE IMPORTANCE OF FEATURES IN THE SPMM TASK

One relevant aspect to analyze is the importance of each feature in $\mathbb{IS}$. Since this is a new input classification problem, different from those previously covered in dense and sparse computation for HPC problems, this is still an open question. The feature importance is defined as a metric that quantifies the contribution of each feature to the prediction of the model, for which a number of techniques can be applied [46]. Table 5 shows an example of this for the two most accurate models previously shown, DT and RF. Feature importance has been calculated following the Gini importance [33]. In practice, and particularly for these two tree-based models, it represents the influence that each feature had in the decision-making process considering that both focus on traversing `if-then-else` rules.

As we can see, *format-specific parameters*, such as l or V for vector-wise and VENOM-based sparse matrices, respectively, *have the greatest influence*, followed by the N and M dimensions of the product, which are the outer

TABLE 5. **Feature importance. Format-specific parameters refers to parameters tailored to a specific sparse matrix format, such as *l* (vector-wise) or *V* (VENOM).**

Feature	M	N	K	d	Format-specific
DT	22%	25%	16%	8%	29%
RF	20%	23%	15%	7%	35%

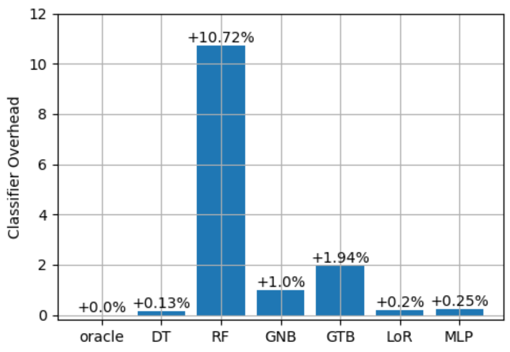

FIGURE 11. **Average inference overhead of each model.**

dimensions. On the contrary, the density of the model is the input feature with the smallest influence overall. This represents a very interesting finding, since it enlightens one of the reasons why general methods do not fit well sparse computational problems for DL. The influence that the configuration of the DL-friendly sparse formats can have in the classification problem demonstrates why general methods for sparse computation or unstructured formats cannot fit DL workloads.

3) INFERENCE OVERHEAD

The overhead of the model's execution is a key factor to be taken into account. This overhead includes the time spent between the feature extraction and the selection of the best kernel configuration.

In order to measure the inference overhead of each classifier, we have used the same matrices referenced in Figure 10 and calculated the relative amount of time that the model's inference represents in the overall SpMM execution time. Figure 11 shows the results obtained in average terms, where the Oracle is the ideal scenario, with no overhead. DT has the lowest inference overhead since it only focuses on traversing `if-then-else` rules. In general, all the models show a low overhead with the exception of RF, which in average requires 10% of the total training time, its maximum overhead being 135%. This overhead depends on the number of trees that compose the model, which in this case provided the best accuracy for 300 estimators (trees), with a total size of 44GB. While the number of trees of RF can be pruned to find a performance-accuracy trade-off, the excellent performance of DT both in terms of accuracy and performance makes this model the most suitable one for this case study. A second noteworthy aspect is the disparity in training duration among the models: MLP can require days, RF minutes, and DT models seconds. This significant variation in training time facilitates the retraining process with new historical data as needed, streamlining updates and enhancements to the models with efficiency.

TABLE 6. **DTs statistics on the NVIDIA T4 from the Tesla architecture, and trained on different input datasets independently.**

Dataset Name	Dataset size	Best DT- Raw dataset				Best DT - Norm dataset			
		#classes	accy	HIM	h, L	#classes	accy	HIM	h, L
real-world	5,139	130	0.35	0.49×	$\infty, 1$	89	0.52	0.50×	8, 2
po2	7,939	135	0.49	0.54×	$\infty, 1$	92	0.64	0.59×	$\infty, 1$
go2	86,695	166	0.74	0.67×	$\infty, 1$	**111**	**0.87**	**0.81×**	$\boldsymbol{\infty, 4}$

TABLE 7. **DTs statistics on the NVIDIA A100 from the Ampere architecture, and trained on different input dataset independently.**

Dataset Name	Dataset size	Best DT- Raw dataset				Best DT - Norm dataset			
		#classes	accy	HIM	h, L	#classes	accy	HIM	h, L
real-world	5,139	175	0.31	0.54×	$\infty, 1$	66	0.46	0.60×	$\infty, 1$
po2	7,939	218	0.55	0.57×	$\infty, 1$	76	0.7	0.61×	$\infty, 1$
go2	267,541	278	0.8	0.67×	$\infty, 4$	**134**	**0.89**	**0.83×**	$\boldsymbol{\infty, 4}$

4) CONCLUSIONS

DTs have emerged as the most robust, precise, and lightweight approach for sparse GPU kernels in DL. Detailed results are presented in Tables 6, and 7 for the Turing and Ampere architectures, respectively. Each row in these tables represents the number of samples, and the performance evaluation of the best DT model found when trained *independently* on one of the three-defined datasets: real-word, *po*2, and *go*2. This evaluation intends to demonstrate the importance of the synthetic datasets defined. The "Raw dataset" set of columns represents the usage of raw historical data for training, without any preprocessing, while the "Norm dataset" ones denotes results achieved after normalizing the data before the training process (Exploration Module, Section IV-D). For each of these scenarios, the column "#classes" denotes the total number of kernel configurations selected as the best one at least once, while the "accy" column indicates the accuracy achieved by the best DT configuration. Such configuration is represented in the "h, L" column, where L represents the minimum number of samples required for a kernel configuration (class) to be a leaf node, and h is the maximum height of the DT. In practical terms, regarding L, split points in the DT are considered at any depth only if they ensure a minimum of L training samples in both resulting branches. This criterion serves to potentially enhance model smoothness. With regard to h, if $h = +\infty$, nodes are expanded until either all leaves are pure, which means that all feature values within the node belong to a single class, or until they contain fewer than $2L$ samples. Following previous studies about LBPMs for dense linear algebra kernels [28], we defined $h \in \{1, 2, 4, 8, +\infty\}$, and $L \in \{1, 2, 4, 0.1, 0.2, 0.4, 0.5\}$. Finally, the "HIM" column represents the Highest Impact of Misclassification, that is, the lowest relative performance of the predicted configuration w.r.t. the Oracle ($0 \sim 1\times$, the higher the better), which has been achieved by averaging the biggest misclassification impact for each M problem dimension present in the test set (in-depth analysis of Figure 10 evaluations).

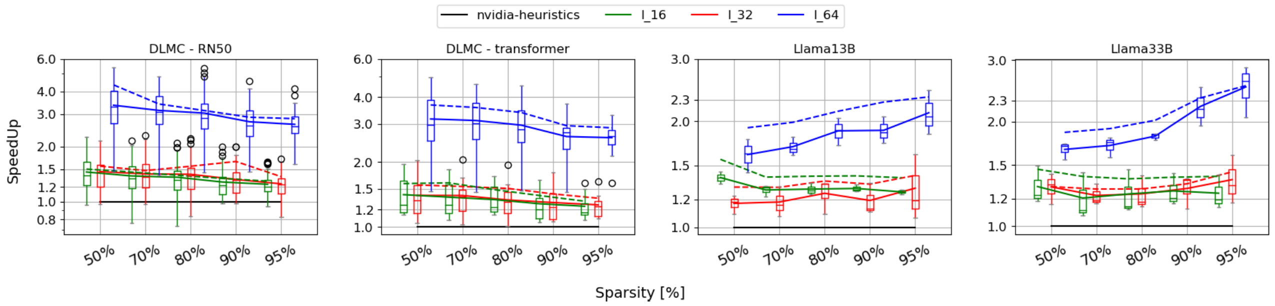

FIGURE 12. **Speedup results obtained by using the kernel configurations provided by STuning-DL w.r.t. the configurations extracted from the NVIDIA-heuristics on matrices from DLMC dataset and two different Llama 2 models. The benchmarks have been performed on the NVIDIA T4-16GB GPU. The kernels used belong to the Shfl-BW implementations. The notation *l_x* represent the vector-length *x* used for the vector-wise sparse matrices.**

In accordance with the results shown in Figure 9, the number of historical data samples used for training is crucial in the performance of the classifier. In general terms, the number of samples in the real-world and *po*2 datasets is insufficient, and the models trained just using one of these datasets suffer from poor generalization. A remarkable aspect in that sense is the number of classes, which increases the larger the dataset, demonstrating why existing auto-tuners fail to adapt to this kind of DL workloads, since the total number of classes defined is, generally, very limited. However, a wide range of kernel configurations helps to adapt each implementation in accordance with the properties of the input matrix, which can vary drastically. One important aspect to remark here is the difference in the number of classes before normalization between the Ampere and the Turing architecture. This is due to the difference between both architectures in terms or resources and tunability (e.g., the Ampere architecture supports more instructions shapes). Another important reason for this is that VENOM is only supported in the Ampere architecture (and newer ones), so the configurations associated with this format are not included in the Turing total count of classes. Finally, it is worth noting the boost in terms of accuracy that the proposed normalization module introduces in the final results, as not only the models are more accurate, but also the impact of misclassification is significantly reduced. In these terms,in the worst possible scenario, our predictor will be able to provide more than 80% of the ideal performance achievable. However, as showed in Figure 10, the average performance is always $> 0.97\times$.

B. PERFORMANCE EVALUATION ON REAL USE-CASES

In this section we evaluate STuning-DL on real use-cases, and compare the performance achieved w.r.t. the corresponding kernel configurations selected by the NVIDIA heuristics for the same inputs. For vector-wise sparsity (i.e., Shfl-BW [24] kernels), we selected the baseline configurations from the NVIDIA cuSparse+BlockedELL implementation, both using the same block-length l. In the case of V:N:M sparsity (i.e., VENOM [5] kernels), we extracted the configurations from the vendor-library implementation for N:M sparsity, NVIDIA cuSparseLt. To extract such information we have used NVIDIA Nsight Compute [43] to profile, for each input problem, the selected SpMM kernel and extract the configuration information from the demangled method name .[2] Such information includes the following parameters: $(BM \times BN \times BK)$, $(WM \times WN \times WK)$, $(MM \times MN \times MK)$, $NSTAGE$ (or Batch size). Both STuning-DL and NVIDIA-based configurations must be constrained in the BM value, which cannot exceed the vector length (l or V, depending on the sparse format selected) of the sparse input matrix in the third-party implementations selected.

In order to evaluate the performance of STuning-DL, we have used the sparse matrices extracted from the models included in the DLMC dataset [17], and two different versions of the Llama 2 model (13B and 33B parameters). We conducted the experiments on NVIDIA T4-16GB and NVIDIA A100-40GB GPUs. The sparse matrices have been pruned to three different vector lengths of $l = 16$, 32, and 64 in the case of Shfl-BW, and $V = 32$, 64, and 128 for VENOM. Sparsity levels of 50%, 70%, 80%, 90%, and 95% have been considered.

The DLMC dataset is formed by sparse matrices from extensively used models, such as ResNet-50 [19] and the Transformer architecture [60] on the WMT 2014 English-to-German dataset. The problem shapes $(M \times K)$ in this dataset (weights) go from small to medium-size. In the second use-case, the Llama 2 model is an LLM whose layers represent an scenario with large matrices, with dimensions that largely exceed the ones considered before (e.g., 13,824 elements). Note that STuning-DL has not been trained on any matrix with such size, since the maximum value considered was defined in the synthetic dataset with 8,192 elements.

This evaluation allows us to asses the models' performance across previously unseen combinations of $(M, N, K, d, l/V)$, akin to scenarios encountered with DLMC extracted matrices. Furthermore, it enables the evaluation of the model's behavior when encountering significant disparities with the cases considered during the LBPMs training. Hence, one of the objectives of this study is to showcase our detailed and

[2] An example of such kernel name is the following: sm80_xmma_sparse_-gemm_f16f16_f16f32_f32_tt_t_tilesize128 $\times$ 64x64_stage4 _warpsize2 $\times$ 2x1_sptensor16 $\times$ 8x32_execute_kernel_cusparse

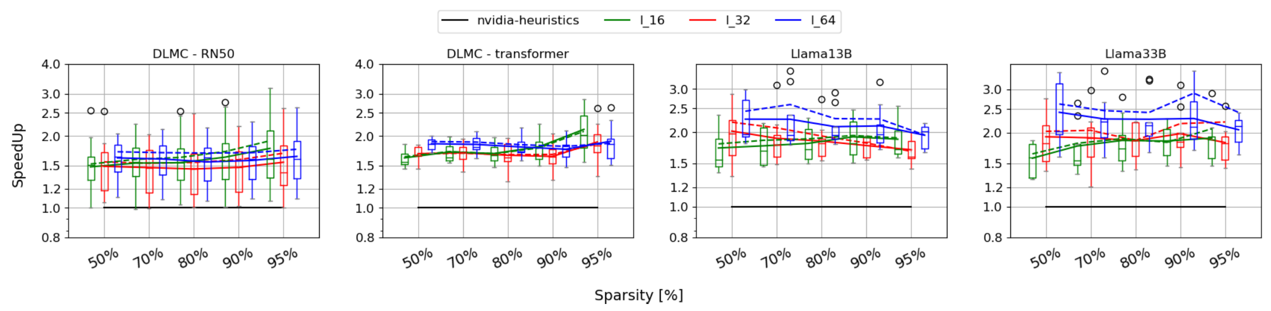

FIGURE 13. Speedup results obtained by using the kernel configurations provided by STuning-DL w.r.t. the configurations extracted from the NVIDIA-heuristics on matrices from DLMC dataset and two different Llama 2 models. The benchmarks have been performed on the NVIDIA A100-40GB GPU. The kernels used belong to the Shfl-BW implementations. The notation *l_x* represent the vector-length *x* used for the vector-wise sparse matrices.

transparent guideline for creating LBPM-based autotuners for third-party implementations tailored to sparsity in Deep Learning. This is particularly important as existing autotuners often lack the capability to effectively adapt to the unique characteristics of sparse DL problems, or *operate generally as black boxes*.

Figures 12, 13, and 14 show the speedups of the configurations selected by STuning-DL w.r.t. to those selected by the NVIDIA libraries heuristics on the selected datasets for Shfl-BW kernels in the NVIDIA T4-16GB, and in the NVIDIA A100-40GB, and for VENOM in the NVIDIA A100-40GB, respectively. Remind that VENOM cannot be executed on the NVIDIA T4-16GB because it requires Sparse Tensor Cores (SPTC), available since the Ampere architecture. The solid lines represent the average performance of the configuration provided by STuning-DL, while the dotted lines represent the average performance of the configurations provided by the Oracle. To attain such optimal configurations, we benchmarked all configurations within $\mathbb{TS}$ for each input problem. It is worth noting that the entire $\mathbb{T}$uning $\mathbb{S}$pace comprises hundreds of potential kernel configurations (see Tables 6, and 7). For instance, in Figure 12, even constraining the search process to the matrices included in these datasets, it required approximately 2-3 days for each sub-figure, that is, one of the four defined datasets, for a single kernel implementation and GPU architecture. This underscores the challenge and resource-intensive nature of maintaining specialized databases like SAMPL [52], which are prohibitively expensive and impractical for sparse Deep Learning problems. In contrast, STuning-DL offers a promising approach to swiftly deliver near-optimal configurations in real-time.

Starting with the results on the T4, Figure 12, we can see that, in average terms, STuning-DL outperforms NVIDIA-heuristics in all the scenarios. In general terms, there are some input problems in DLMC-RN50 where it selects a worse configuration. However, this only represents a residual 2% of the matrices present in the DLMC-RN50 testset. Overall, the NVIDIA-heuristics in this GPU architecture behave significantly better for $l = 16$, and 32 than $l = 64$. Hence, despite STuning-DL nearly matching the performance of the Oracle for $l = 16$, and 32, the speedups achievable have an upper limit (Oracle), typically ranging from 20% to 50% on average, depending on the sparsity level and testset. It is worth noting also that these speedups compare the performance of the same template kernel using two different hyperparameter configurations. Thus, considering the speedup relative to the dense version (i.e., cuBLAS), a performance improvement of 20% can determine whether the sparse implementation is faster or slower than its dense counterpart.

As a real example extracted from these evaluations, for an input problem of size (5,120 $\times$ 13,824 $\times$ 2,048), with 70% sparsity, and $l = 32$, the NVIDIA-heuristics configuration achieves an speedup w.r.t. cuBLAS of 0.94$\times$, whereas the STuning-DL configuration achieved a 1.20$\times$ improvement. This enhancement could potentially allow the use of lower sparsity levels, which may better preserve the original accuracy. To provide further context, a performance improvement of 3$\times$ implies that a kernel running 2$\times$ faster than its dense counterpart would now run 6$\times$ faster, so small improvements here can have a huge overall impact.

However, the STuning-DL's performance improvement becomes evident in scenarios where the NVIDIA-heuristics fail to align well with the sparse input matrix format. In such cases, where the Oracle diverges notably from the baseline, we observed speedups of up to 5.42$\times$ for $l = 64$. It is important to note that the most influential parameter in a sparse problem, as extracted from previous studies (Table 5), was the format-specific feature; in this case, the l value.

These results strongly support our hypothesis that while *general kernel configurations* for sparse matrices may work in certain scenarios, *they lack robustness* when confronted with the diverse range of custom sparse formats. STuning-DL emerges as a solution that offers the potential to achieve near-optimal configurations, elevating performance even in situations where generic configurations may suffice. However, its primary strength lies in its robustness, capable of significantly and consistently enhancing performance in cases where generic configurations struggle to adapt to the intricacies of the input problem.

Continuing our analysis, Figure 13 repeats the previous evaluations but on the A100-40GB GPU. In this GPU architecture, there is more room for performance

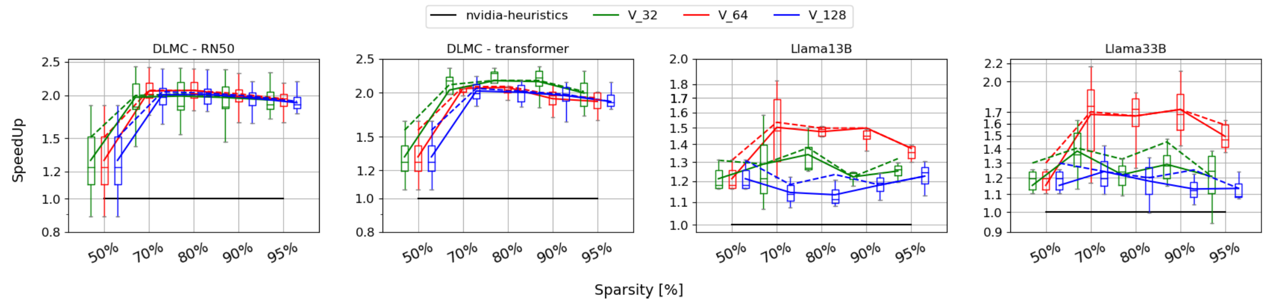

FIGURE 14. **Speedup results obtained by using the kernel configurations provided by STuning-DL w.r.t. the configurations extracted from the NVIDIA-heuristics on matrices from DLMC dataset and two different Llama 2 models. The benchmarks have been performed on the NVIDIA A100-40GB GPU. The kernels used belong to the VENOM implementations. The notation V_x represent the vector-length x used for the V:N:M sparse matrices.**

improvement through tuning compared to the T4-16GB GPU (as evidenced by the classes in Table 6 vs. Table 7). Overall, the performance of configurations provided by STuning-DL surpasses that of NVIDIA-heuristics in all cases, with only a few exceptions in the DLMC-RN50 dataset, accounting for less than 1% of the total matrices in this set of problems. More specifically, the average speedup ranges from approximately $1.50 - 2\times$ faster, with some cases reaching up to about $3\times$. Similarly, for the DLMC-transformer dataset, the average performance improvement ranges between approximately $1.75 - 2.2\times$. As problem sizes increase (Llama2-13B, and 33B), there is also an increment in the average speedups obtained, with improvements of up to $3.5\times$.

In comparison with the results on Figure 12, in this architecture, the performance improvement w.r.t. the NVIDIA-heuristics occurs at approximately the same ratio for all three l configurations. This uniformity is likely due to the broader range of tunability available in this architecture, as mentioned earlier.

Once again, across all four datasets, the predicted configurations achieve speedups that closely approach those of the Oracle, indicating that STuning-DL is capable of providing near-optimal configurations on the A100 GPU as well.

Finally, we have evaluated STuning-DL on the VENOM format, depicted in Figure 14. It is important to note that cuSparseLt, the NVIDIA library from which the heuristics for this format were extracted, demonstrated the most robust behavior in Section III (Figure 4). So we should expect a good kernel configuration selection in this library. However, cuSparseLt is designed for sparsity in DL but is limited to the 2:4 format, equivalent to 50% sparsity. In contrast, VENOM is an adaptation of the 2:4 format that overcomes this limitation, allowing for the utilization of arbitrary sparsity levels. The parameter V, configurable in this format, is only applied for sparsities $> 50\%$.

During our evaluations, we observed that cuSparseLt heuristics, overall, do not adapt properly to small input problems, such as those present in the DLMC dataset, probably because of the low tunability we saw in Table 2. In average terms, STuning-DL provides better configurations in all the scenarios. There are instances in the DLMC-RN50 dataset where STuning-DL selects a worse configuration than cuSparseLt heuristics. However, this accounts for only 4% of the matrices in this sub-dataset. However, there is a significant performance improvement for sparsities greater than 50%, precisely where the V value is applied. Since this is a custom variable related to VENOM, STuning-DL demonstrates better adaptability to such input problems, achieving speedups of up to $2.4\times$.

For larger problem sizes, where cuSparseLt exhibits its best behavior, the performance improvements becomes more limited. Similarly to what occurred with $l = 16$ and 32 on the T4, NVIDIA heuristics select decent configurations for $V = 32$ and 128 in the case of Llama-2 models. However, for $V = 64$, we once again observe a significant performance improvement, in average terms, of around $1.5\times$ for Llama-2 13B and $1.7\times$ for Llama-2 33B, specifically for sparsities exceeding 50%, where V can be used. Crucially, STuning-DL consistently provides near-optimal configurations.

In addition to these performance improvements, it is worth to remind, as detailed in Section III-B, that the `cusparseLtMatmulSearch` autotuning function within cuSparseLt can take hours to find the optimal NVIDIA kernel configuration for just one matrix. In contrast, STuning-DL can operate in real-time, as demonstrated in Figure 11, with negligible overhead (i.e., $\sim 0.01\%$). This characteristic broadens its applicability to various domains, including AutoML.

1) CONCLUSIONS

Overall, STuning-DL has proven to be a robust autotuner capable of maximizing the performance of sparse kernel implementations for Deep Learning, approaching the near-optimal performance (Oracle) achieved by tuning their hyperparameters. It consistently outperforms generic existing approaches, showcasing its effectiveness across various GPU platforms, sparse implementations, compression formats, configurations, problem sizes, and sparsity levels. Therefore, this paper represents a detailed and transparent guideline for developing or customizing LBPM-based autotuners tailored to sparsity in Deep Learning, particularly for third-party implementations lacking autotuning capabilities.

VI. CONCLUSION

The high variability in the matrices present in modern Machine Learning models can significantly undermine the efficacy of existing GPU kernels if they are not meticulously tuned.This challenge is further compounded in scenarios involving sparse computations, requiring adaptations of kernels to diverse sparsity levels, non-zero distributions, and sparse compression formats, among other factors. In this paper we have presented STuning-DL, which as far as we know, is the first autotuner uniquely tailored to CUDA C++ template-based sparse linear-algebra kernels for Deep Learning workloads, spanning from high-level aspects down to GPU-native instructions specifics. It stands as the first solution that accounts for pruning dependency while being aware of hardware details down to *ptx*-code level. STuning-DL has demonstrated its effectiveness in adapting each kernel configuration to the input dynamics across various GPU platforms, kernel implementations, sparse formats, and a broad spectrum of sparsity levels. STuning-DL achieves speedups of up to 5.42× on T4-16GB GPU and 4.98× on the A100-40GB GPU compared to the configurations extracted from the NVIDIA sparse libraries cuSparse and cuSparseLt on the same input problems. This underscores the capability of STuning-DL both to deliver high-performance kernels as well as to consistently approximate near-optimal configurations across diverse scenarios. Furthermore, this paper delineates a detailed guideline that can be easily applied to new implementations, sparse compression formats and hardware architectures that may arise within the community. By doing so, STuning-DL not only presents a robust solution but it also *seeks to contribute to the broader advancement and accessibility of optimized sparse linear-algebra kernels for Deep Learning*.

REFERENCES

[1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, San Francisco, CA, USA, Tech. Rep., 2019, vol. 1, no. 8, p. 9.

[2] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–42, Sep. 2019.

[3] T. B. Brown, "Language models are few-shot learners," in *Proc. NIPS*, 2020, pp. 1877–1901.

[4] R. L. Castro, D. Andrade, and B. B. Fraguela, "Probing the efficacy of hardware-aware weight pruning to optimize the SpMM routine on ampere GPUs," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2022, pp. 135–147.

[5] R. L. Castro, A. Ivanov, D. Andrade, T. Ben-Nun, B. B. Fraguela, and T. Hoefler, "VENOM: A vectorized N:M format for unleashing the power of sparse tensor cores," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, New York, NY, USA, Nov. 2023, pp. 1–14.

[6] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018.

[7] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, "Efficient tensor core-based GPU kernels for structured sparsity under reduced precision," in *Proc. SC21, Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2021, pp. 1–13.

[8] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 115–126, May 2010.

[9] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, Sep. 1995.

[10] G. Dai, G. Huang, S. Yang, Z. Yu, H. Zhang, Y. Ding, Y. Xie, H. Yang, and Y. Wang, "Heuristic adaptability to input dynamics for SpMM on GPUs," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, New York, NY, USA, Jul. 2022, p. 595.

[11] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, Nov. 2011.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[13] Y. Ding, Y. Wu, C. Huang, S. Tang, F. Wu, Y. Yang, W. Zhu, and Y. Zhuang, "NAP: Neural architecture search with pruning," *Neurocomputing*, vol. 477, pp. 85–95, Mar. 2022.

[14] A. Dosovitskiy, "An image is worth 16×16 words: Transformers for image recognition at scale," in *Proc. Int. Conf. Learn. Represent.*, 2020.

[15] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "AlphaSparse: Generating high performance SpMV codes directly from sparse matrices," in *Proc. SC, Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2022, pp. 1–15.

[16] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse GPU kernels for deep learning," in *Proc. SC, Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–14.

[17] *Deep Learning Matrix Collection*, Google Research, Atlanta, GA, USA, 2020.

[18] G. Huang, S. Liu, L. van der Maaten, and K. Q. Weinberger, "CondenseNet: An efficient DenseNet using learned group convolutions," 2017, *arXiv:1711.09224*.

[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[20] J. Hestness, S. Narang, N. Ardalani, G. Diamos, H. Jun, H. Kianinejad, M. M. A. Patwary, Y. Yang, and Y. Zhou, "Deep learning scaling is predictable, empirically," 2017, *arXiv:1712.00409*.

[21] T. K. Ho, "Random decision forests," in *Proc. 3rd Int. Conf. Document Anal. Recognit.*, Aug. 1995, pp. 278–282.

[22] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, no. 241, pp. 1–124, 2021.

[23] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[24] G. Huang, H. Li, M. Qin, F. Sun, Y. Ding, and Y. Xie, "Shfl-BW: Accelerating deep neural network inference with tensor-core aware weight pruning," in *Proc. 59th ACM/IEEE Design Autom. Conf.*, New York, NY, USA, Jul. 2022, pp. 1153–1158.

[25] J. Kaplan, "Scaling laws for neural language models," 2020, *arXiv:2001.08361*.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 25, 2012.

[27] E. Kurtic, D. Campos, T. Nguyen, E. Frantar, M. Kurtz, B. Fineran, M. Goin, and D. Alistarh, "The optimal BERT surgeon: Scalable and accurate second-order pruning for large language models," 2022, *arXiv:2203.07259*.

[28] P. S. Labini, M. Cianfriglia, D. Perri, O. Gervasi, G. Fursin, A. Lokhmotov, C. Nugteren, B. Carpentieri, F. Zollo, and F. Vella, "On the anatomy of predictive models for accelerating GPU convolution kernels and beyond," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, pp. 1–24, Jan. 2021.

[29] F. Lagunas, E. Charlaix, V. Sanh, and A. M. Rush, "Block pruning for faster transformers," 2021, *arXiv:2109.04838*.

[30] M. Li, M. Zhang, C. Wang, and M. Li, "AdaTune: Adaptive tensor program compilation made efficient," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 14807–14819.

[31] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, Feb. 2019, pp. 229–241.

[32] M. Lin, Y. Zhang, Y. Li, B. Chen, F. Chao, M. Wang, S. Li, Y. Tian, and R. Ji, "1xN pattern for pruning convolutional neural networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 4, pp. 3999–4008, Apr. 2023.

[33] W.-Y. Loh, "Classification and regression trees," *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 1, no. 1, pp. 14–23, 2011.

[34] R. Malouf, "A comparison of algorithms for maximum entropy parameter estimation," in *Proc. 6th Conf. Natural Lang. Learn. (COLING)*, 2002, pp 1–7.

[35] A. Mishra, J. A. Latorre, J. Pool, D. Stosic, D. Stosic, G. Venkatesh, C. Yu, and P. Micikevicius, ''Accelerating sparse deep neural networks,'' 2021, *arXiv:2104.08378*.
[36] T. Mohammed and R. Mehmood, ''Performance enhancement strategies for sparse matrix-vector multiplication (SpMV) and iterative linear solvers,'' 2022, *arXiv:2212.07490*.
[37] I. Nisa, C. Siegel, A. S. Rajam, A. Vishnu, and P. Sadayappan, ''Effective machine learning based format selection and performance modeling for SpMV on GPUs,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2018, pp. 1056–1065.
[38] C. Nugteren, ''CLBlast: A tuned OpenCL BLAS library,'' in *Proc. Int. Workshop OpenCL*, New York, NY, USA, May 2018, pp. 1–10.
[39] *NVIDIA PTX*, NVIDIA, Santa Clara, CA, USA, 2024. Accessed: Apr. 1, 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
[40] *Exploiting NVIDIA Ampere Structured Sparsity With CuSPARSELt*, NVIDIA, Santa Clara, CA, USA, 2020.
[41] NVIDIA. (2023). *The Cusparse Library*. Accessed: Oct. 6, 2023. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html
[42] *Exploiting NVIDIA Ampere Structured Sparsity With cuSPARSELt*, NVIDIA, Santa Clara, CA, USA, 2024. Accessed: Apr. 1, 2024.
[43] NVIDIA. (2024). *Nsight Compute*. Accessed: Apr. 1, 2024. [Online]. Available: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html
[44] S. M. Omohundro, *Five Balltree Construction Algorithms*. Berkeley, CA, USA: International Computer Science Institute Berkeley, 1989.
[45] S. K. Pal and S. Mitra, ''Multilayer perceptron, fuzzy sets, and classification,'' *IEEE Trans. Neural Netw.*, vol. 3, no. 5, pp. 683–697, Jan. 1992.
[46] 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,'' *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.
[47] A. Peste, E. Iofinova, A. Vladu, and D. Alistarh, ''AC/DC: Alternating compressed/decompressed training of deep neural networks,'' in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 34, 2021, pp. 8557–8570.
[48] A. Rasch and S. Gorlatch, ''ATF: A generic directive-based auto-tuning framework,'' *Concurrency Comput., Pract. Exper.*, vol. 31, no. 5, Mar. 2019, Art. no. e4423.
[49] I. Rish, ''An empirical study of the naive Bayes classifier,'' in *Proc. IJCAI Workshop Empirical Methods Artif. Intell.*, vol. 3, no. 22, 2001, pp. 41–46.
[50] J. Ryu and H. Sung, ''MetaTune: Meta-Learning based cost model for fast and efficient auto-tuning frameworks,'' 2021, *arXiv:2102.04199*.
[51] S. R. Safavian and D. Landgrebe, ''A survey of decision tree classifier methodology,'' *IEEE Trans. Syst. Man, Cybern.*, vol. 21, no. 3, pp. 660–674, May 1991.
[52] SAMPL. (2024). *Tophub Autotvm Log Collections*. Accessed: Mar. 31, 2024. [Online]. Available: https://github.com/tlc-pack/tophub
[53] B. Singer and M. M. Veloso, ''Learning to predict performance from formula modeling and training data,'' in *Proc. 17th Int. Conf. Mach. Learn.*, 2000, pp. 887–894.
[54] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, ''Going deeper with convolutions,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
[55] S. Tavarageri, A. Heinecke, S. Avancha, B. Kaul, G. Goyal, and R. Upadrasta, ''PolyDL: Polyhedral optimizations for creation of high-performance DL primitives,'' *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, pp. 1–27, Mar. 2021.
[56] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta, ''CUTLASS,'' NVIDIA, Santa Clara, CA, USA, Jan. 2023. [Online]. Available: https://github.com/NVIDIA/cutlass
[57] P. Tillet, H. T. Kung, and D. Cox, ''Triton: An intermediate language and compiler for tiled neural network computations,'' in *Proc. 3rd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, New York, NY, USA, Jun. 2019, pp. 10–19.
[58] J. O. Tørring and A. C. Elster, ''Analyzing search techniques for autotuning Image-based GPU kernels: The impact of sample sizes,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2022, pp. 972–981.
[59] H. Touvron, M. Cord, M. Douze, F. Massa, A. Sablayrolles, and H. Jégou, ''Training data-efficient image transformers & distillation through attention,'' in *Proc. Int. Conf. Mach. Learn.*, 2021, pp. 10347–10357.
[60] A. Vaswani, ''Attention is all you need,'' in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017.
[61] Z. Wang, ''SparseRT: Accelerating unstructured sparsity on GPUs for deep learning inference,'' in *Proc. ACM Int. Conf. Parallel Architectures Compilation Techn.*, Sep. 2020, pp. 31–42.
[62] J. Won, C. Mendis, J. S. Emer, and S. Amarasinghe, ''WACO: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program,'' in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 2, Jan. 2023, pp. 920–934.
[63] C. Wong, N. Houlsby, Y. Lu, and A. Gesmundo, ''Transfer learning with neural AutoML,'' in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 31, 2018.
[64] J. Xing, L. Wang, S. Zhang, J. Chen, A. Chen, and Y. Zhu, ''Bolt: Bridging the gap between auto-tuners and hardware-native performance,'' *Proc. Mach. Learn. Syst.*, vol. 4, pp. 204–216, Apr. 2022.
[65] Z. Xu, J. Xu, H. Peng, W. Wang, X. Wang, H. Wan, H. Dai, Y. Xu, H. Cheng, K. Wang, and G. Chen, ''ALT: Breaking the wall between data layout and loop optimizations for deep learning compilation,'' in *Proc. 18th Eur. Conf. Comput. Syst.*, May 2023, pp. 199–214.
[66] D. Yan, W. Wang, and X. Chu, ''Simplifying low-level GPU programming with GAS,'' in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, New York, NY, USA, Feb. 2021, pp. 469–471.
[67] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng, ''Stochastic gradient boosted distributed decision trees,'' in *Proc. 18th ACM Conf. Inf. Knowl. Manage.*, Nov. 2009, pp. 2061–2064.
[68] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze, ''SparseTIR: Composable abstractions for sparse compilation in deep learning,'' in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, vol. 3, New York, NY, USA, Mar. 2023, pp. 660–678.
[69] M. Zhang, M. Li, C. Wang, and M. Li, ''DynaTune: Dynamic tensor program optimization in deep neural network compilation,'' in *Proc. Int. Conf. Learn. Represent.*, 2020.
[70] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, and K. Sen, ''Ansor: Generating $*high − performance*$ tensor programs for deep learning,'' in *Proc. 14th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2020, pp. 863–879.
[71] N. Zheng, B. Lin, Q. Zhang, L. Ma, Y. Yang, F. Yang, Y. Wang, M. Yang, and L. Zhou, ''Sparta: Deep-learning model sparsity via tensor-with-sparsity-attribute,'' in *Proc. 16th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2022, pp. 213–232.

ROBERTO L. CASTRO received the bachelor's degree in computer science and the master's degree in high-performance computing from the University of A Coruña, Spain, in 2019 and 2020, respectively, where he is currently pursuing the Ph.D. degree with the Computer Architecture Group. His research interests include high-performance computing, sparse computation for machine learning (ML), GPGPU programming, and automated machine learning (AutoML).

DIEGO ANDRADE received the M.S. and Ph.D. degrees in computer science from the University of A Coruña, A Coruña, Spain, in 2002 and 2007, respectively. He is currently an Associate Professor with the Departamento de Enxeñaría de Computadores, University of A Coruña, where he has been a Faculty Member, since 2006. His research interests include performance evaluation and prediction, analytical modeling, and compiler transformations.

BASILIO B. FRAGUELA received the M.S. and Ph.D. degrees in computer science from the University of A Coruña, Spain, in 1994 and 1999, respectively. He is currently a Professor with the Departamento de Enxeñaría de Computadores, University of A Coruña, where he has been a Faculty Member, since 1995. His primary research interests include programmability, high-performance computing, heterogeneous systems, and code optimization.