SparseX: Synergizing GPU Libraries for Sparse Matrix Multiplication on Heterogeneous Processors

Ruifeng Zhang*, Xiangwei Wang*, Ang Li†, Xipeng Shen*
*Department of Computer Science, North Carolina State University, Raleigh, NC, USA
†Pacific Northwest National Laboratory and University of Washington, Seattle, WA, USA
{rzhang38, xwang258, xshen5}@ncsu.edu*, ang.li@pnnl.gov†

***Abstract*—Sparse Matrix-Matrix Multiplication (SpMM) on GPU is critical to applications ranging from scientific simulations to Graph Neural Networks (GNNs) and Deep Neural Networks (DNNs). Modern GPUs offer diverse processing units, such as CUDA cores, Tensor Cores, and Sparse Tensor Cores. Many SpMM libraries have been built to harness those different types of processors. Although impressive performance has been reported by each, including cuSparse, Sputnik, CLASP, and Jigsaw, a systematic study in this work shows that no single library is a clear winner across all matrices and scenarios. Based on the empirical observations, this work proposes the first solution to synergize the various libraries to best harness the heterogeneous processors and the array of cutting-edge libraries. The solution is an extensible framework, namely SparseX, that can automatically select the best matrix multiplication library (and types of processors) on the fly for a given sparse matrix on a GPU through an agile accurate predictive model. Experiments show that SparseX can speed up sparse matrix multiplications on thousands of real-world matrices significantly over the SOTA GPU libraries, achieving significant speedups (e.g., as much as 95.34x over cuSparse). Its extensible design makes it easy to be extended to cover new libraries and hardware architectures.**

***Index Terms*—Sparsity, GPU, Sparse Matrix-Matrix Multiplication (SpMM)**

I. Introduction

SpMM (Sparse Matrix-Matrix Multiplication) is crucial for various applications, ranging from scientific simulations [1], [2] to Graph Neural Networks (GNNs) [3], [4], and Deep Neural Networks (DNNs) [5]–[7]. For these applications, GPUs have become the mainstream choice due to their high throughput and massive parallelism—both of which are essential for achieving optimal performance. Maximizing the execution speed of SpMM on GPUs has therefore become a critical challenge, drawing increasing interest from researchers and industry alike. This is evident from the numerous recent research efforts on the topic [7]–[15] and the significant investments made by industry to develop high-performance SpMM libraries for GPUs, such as cuSPARSE library [16], Sputnik [8], and Deep Graph Library (DGL) [4]. Despite extensive research and development, no single SpMM library performs optimally across all matrices and scenarios. The fundamental challenge lies in the fact that SpMM optimization is highly dependent on multiple factors, making it nearly impossible for a single library to achieve the best performance in every case. These influencing factors can be categorized into three main areas: (i) Matrix characteristics – including size, shape, sparsity patterns, and other structural properties. (ii) Usage scenarios – the sparse matrix could be used repeatedly or only once. For example, adjacency matrices in GNNs tend to be sparse and repeatedly reused, whereas sparsity introduced by activation functions [17]–[19] is dynamic and varies with different model inputs. (iii) Hardware architecture – not only differences between GPU models but also the heterogeneity within a single GPU. Modern GPUs incorporate a variety of heterogeneous computing units, each specialized for different matrix sizes, sparsity patterns, and execution scenarios. For example, an NVIDIA A100 contains CUDA Core units, Tensor Core units, and Sparse Tensor Core units [20]. Due to architectural differences, each computing unit works better for some sparse matrices.

Given these complexities, state-of-the-art (SOTA) SpMM libraries are typically designed to leverage a particular type of computing unit and target specific problem settings. For instance, Jigsaw [12] reports superior SpMM performance compared to existing solutions for a subset of sparse matrices. However, a deeper examination reveals that Jigsaw is only effective when the same sparse matrix is used repeatedly, and its optimizations are tailored for Sparse Tensor Cores on NVIDIA GPUs only. Because Jigsaw employs a reordering technique that transforms sparse matrices into a 2:4 sparsity pattern to enable execution on Sparse Tensor Cores. However, this process is computationally expensive, taking thousands to millions times longer than a single SpMM operation. As a result, Jigsaw is impractical for scenarios where SpMM is performed only a few times on a given matrix. In such cases, other libraries that don't incur large preprocessing overhead may be more efficient. This illustrates a common limitation of prior SpMM optimization efforts—whether they focus on data layout transformations [12], [21], [22], algorithmic refinements [7], [8], or hardware-specific tuning [10], [23]—none have successfully created a universal solution that excels across all scenarios, as Figure 1 shows.More importantly, all these methods incur different preprocessing overheads, which are non-negligible in scenarios where the sparse matrix is not repeatedly reused. So directly applying them generally fails to improve performance with dynamic sparsity, which are commonly observed in modern DNNs [19] and large language models (LLMs) [18]. Other predictor-based approaches [24]–[27] attempt to synergize different sparse methods, but their

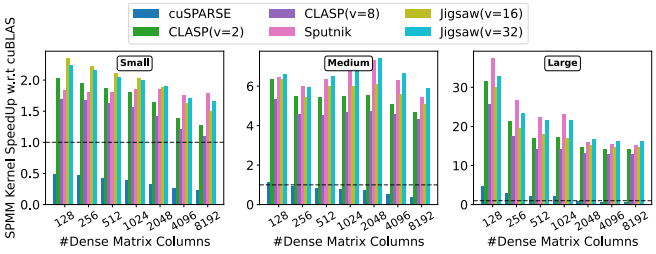


Fig. 1: SpMM Kernel's average speedups over cuBLAS on small, medium, and large matrices from SuiteSparse on RTX 4090 GPU; the involved libraries represent the state-of-the-art GPU libraries for SpMM (detailed in Section V-A). The x-axis is the number of columns of the output matrix. No library excels in all cases.

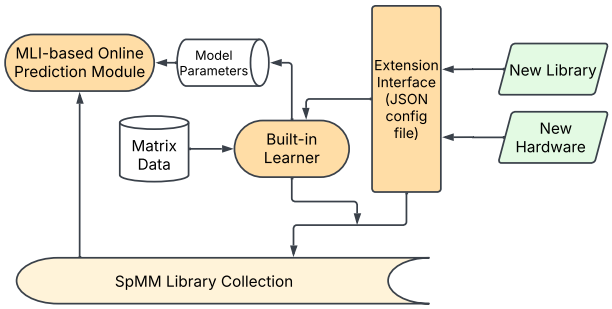


Fig. 2: Framework of SparseX. Key components include the *MLI-based online prediction module* for selecting the best libraries on the fly, the *built-in learner* for automatic model updates, and the *extension interface* for integrating new libraries or adapting to new hardware platforms.

feature extraction and model inference still introduce considerable overhead.

So, instead of proposing yet another SpMM optimization targeted at a specific use case or hardware configuration, this work takes a different approach. We aim to develop a solution that effectively synergizes multiple existing matrix-multiplication libraries, leveraging their strengths in different scenarios to achieve the best possible performance across diverse matrices and computing environments. By doing so, we seek to overcome the limitations of individual libraries and provide a flexible, adaptive framework for SpMM execution on GPUs. To make the solution able to serve as a drop-in replacement of existing libraries, it must meet several requirements: (1) maintaining high accuracy in predicting the optimal library to maximize performance gains, (2) coming with an extensible design that facilitates the integration of new (or upgraded) matrix multiplication libraries, as well as support for different hardware platforms such as AMD and NVIDIA GPUs. (3) ensuring that the runtime overhead of selection is negligible, which is essential for the solution to work in various scenarios, especially for dynamic sparsity. Examples include Sparse Mixture-of-Experts [28] and activation sparsity [17], [18], where, sparse matrices are input-dependent and generated on the fly during inference, causing frequent SpMM operations on varying structures.

To meet these objectives, we propose **SparseX**, the first extensible framework designed to dynamically select the optimal GPU matrix multiplication library and processing unit for any given input on the fly. Our approach builds upon a multi-faceted research strategy. We begin by conducting an in-depth benchmarking study of leading matrix multiplication libraries, systematically evaluating their performance across a diverse set of GPUs and workloads. From these insights, SparseX constructs a hybrid predictive model to ensure accurate and efficient selection. The framework employs an augmented two-stage lazy-and-light scheme, and leverages novel efficient features of sparse matrices (**memory latency indicators (MLI)**) and a hybrid agile predictor to maximize selection efficiency while minimizing overhead. To further enhance adaptability, SparseX is designed with a modular architecture and built-in learners, enabling a seamless plug-and-play experience for incorporating new matrix multiplication libraries.

Experimental results demonstrate that SparseX delivers significant speedups (e.g., as much as 95.34× over cuSparse) over state-of-the-art (SOTA) matrix multiplication libraries across thousands of real-world sparse matrices from the SuiteSparse dataset [29],and is capable of utilizing sparse activation to achieve end-to-end speedup for LLMs. By introducing a dynamic and synergistic approach to SpMM, SparseX offers a new way to enhance sparse matrix operations. By bridging the gap between hardware diversity and software efficiency, SparseX unlocks the potential of heterogeneous GPU computing for sparse matrix multiplication, and achieves speedup in almost all scenarios. The key contributions of this work are:

- The first extensible, cross-platform framework **SparseX** that best harness various GPU processors by synergizing matrix multiplication libraries.
- Introduction of **Skipping Granularity** and **Memory Latency Indicator (MLI)**, two key concepts for analyzing and efficiently predicting SpMM kernel performance.
- A novel, overhead-aware and highly agile performance model with an on-the-fly decision module that achieves near-optimal selection across diverse input scenarios.
- Empirical demonstrations of the effectiveness of SparseX in speeding up sparse matrix multiplications on thousands of real-world matrices and modern LLMs.

II. Overview of SparseX

SparseX is the first known framework that on-the-fly selects the best library to use for a given SpMM problem to maximize the performance. Figure 2 illustrates the modular architecture of SparseX. Its key components include the *MLI-based online prediction module* (MLI detailed in Section III), *built-in learner*, and *extension interface*. The framework adopts a plug-and-play design that supports a wide variety of matrix multiplication libraries, making it robust across diverse input scenarios. We explain each of the key components next. For easy reference, we list the common notations in Table I.

TABLE I: Common notations.

Notation	Description
A	Left input matrix (sparse)
B	Right input matrix (dense)
C	Output matrix (dense)
M	Number of rows of matrix A and C
K	Number of columns of matrix A and rows of B
N	Number of columns of matrix B and C
BM	Block size along dimension M in tiling
BN	Block size along dimension N in tiling
BK	Block size along dimension K in tiling

III. MLI: A KEY LIGHTWEIGHT FEATURE

The *Memory Latency Indicator (MLI)* is one of the key reasons for SparseX to enable the efficient on-the-fly selection of SpMM kernels. There are some prior studies [23], [26], [27], [30] that have used features of sparse matrices to make predictions, such as predicting the best storage format to use [26], [27]. They, however, all rely on features (e.g., number of non-zeros in the diagonals and difference between number of non-zeros of adjacent rows) that require one or more traversals of the sparse matrix, or computationally expensive models (e.g. Convolutional Neural Networks), making the overhead too large for SparseX to use. To address this, we propose MLI, a lightweight traversal-free feature that effectively captures a key aspect of a sparse matrix in determining the performance of an SpMM kernel.

A. L2-Cache-Bound Behavior

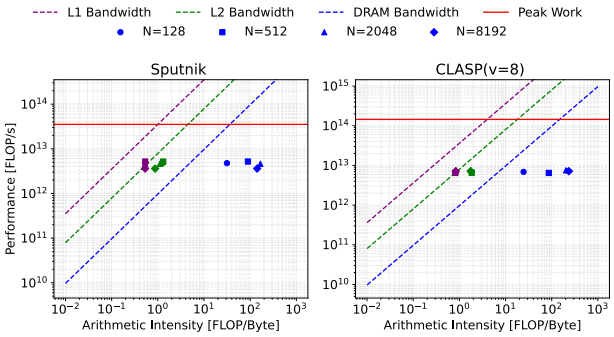


Fig. 3: Sputnik and CLASP's roofline charts of SpMM. The sparse matrix dimensions are 4096×4096 and the dense matrix dimensions are $4096 \times N$. The color of dots indicates the memory hierarchy of the result. The L2 cache bound (green line) closely aligns with the achieved results, indicating that the kernels are predominantly limited by L2 cache performance.

MLI design is motivated by our analysis of GPU SpMM. SpMM kernels are strongly memory-bound—especially with FP16—so we profiled them with Nsight Compute (NCU). Figure 3 shows roofline models for Sputnik and CLASP: across all N, arithmetic intensity tracks the L2 bandwidth ceiling, indicating an L2-bound regime. This holds across libraries and sizes (less so for very small matrices where L1 effects appear). Consistently, L2 cache read volume is a strong predictor of runtime, particularly at larger sizes.

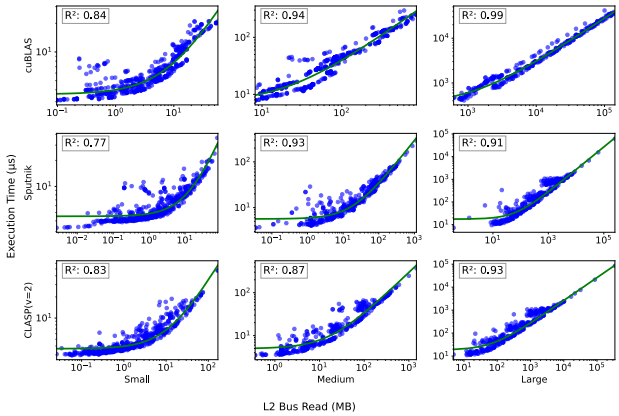


Fig. 4: Linear regression for execution time and L2 bus read count. The green curve is the linear regression. It's curved due to using logrithm as axis scales. R-Square (R^2) is the coefficient of determination for the linear regressor, with values closer to 1 indicating better fit.

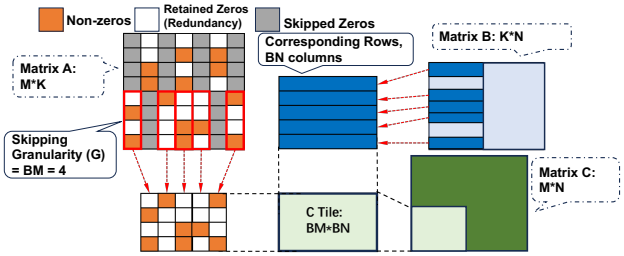


Fig. 5: Typical computing pattern of SpMM. The example is CLASP(v=4). Skipping granularity (G) defines the smallest unit that SpMM libraries consider for skipping zero elements.

Although the L2 cache read traffic serves as a semi-linear indicator of kernel performance, its exact value cannot be determined prior to execution. Therefore, we aim to estimate the L2 cache read traffic using features of the input matrices. Additionally, while DRAM reads are not as critical, they still have a significant impact on kernel execution time. Thus, we also seek to estimate DRAM read traffic. We found that certain input matrix characteristics can serve as linear indicators of these memory traffic values, which we refer to as MLIs.

Figure 5 illustrates a typical SpMM computation pattern, where thread blocks are launched on the GPU. We introduce the concept of *Skipping Granularity (G)*, which defines the smallest unit that SpMM libraries consider when skipping zero elements. A library either skips an entire block or retains it, meaning that only if all elements within a block are zero can it be skipped. Different libraries and kernel functions adopt different values for G. For example, Sputnik skips individual elements, so $G_{\text{Sputnik}} = 1$, whereas CLASP, which operates on column vectors, has $G_{\text{CLASP}} = v$, where v is the column vector length. In this example $G = 4$.

The result matrix is partitioned into tiles of size $BM \times BN$, where each GPU thread block is responsible for one tile in the output. Assuming that each thread block reads its data directly

from L2 cache, disregarding potential L1 cache hits, and that non-zeros are uniformly distributed across the sparse matrix, we can compute the total L2 cache reads given the input matrix size, sparsity, and the skipping granularity G of the library. Note that all these factors are attainable without traversal of the matrix. The input matrix size is usually specified in the metadata of the input matrix, the sparsity can be easily calculated based on the size of the sparse representation in the input and the input matrix size, and the skipping granularity G is known for a given library.

We next show how MLI is calculated. Without the loss of generality, we assume that the skipping granularity is a column vector consisting of G rows and one column. Under this assumption, we have $BM = G$. We first define the *Estimated Density* as $D = (1 - S^G)$, where S is the matrix sparsity. D represents the probability that a block contains nonzero elements and is therefore retained.

As Figure 5 shows, the total number of tiles in C is: $\frac{M}{G} \times \frac{N}{BN}$. For each tile in C, the corresponding A tile has:

$$D \times K \times G \times Const_1 \quad \text{(Bytes)},$$

where $Const_1$ is the number of bytes required to store one element. For example, for CSR, each element consists of an FP16 value and a short integer column index, making $Const_1 = 4$. Similarly, the corresponding tile in B has:

$$D \times K \times BN \times Const_2 \quad \text{(Bytes)},$$

where $Const_2$ is the size of a dense matrix element. Since dense matrices store values only, $Const_2 = 2$ for FP16.

Thus, the total L2 cache read volume is:

$$(D \times K \times G \times Const_1 + D \times K \times BN \times Const_2) \times \frac{M}{G} \times \frac{N}{BN} \quad \text{(Bytes)}.$$

It can be written as:

$$M \times K \times D \times N \times \left(\frac{Const_1}{BN} + \frac{Const_2}{G} \right) \quad \text{(Bytes)}.$$

Here, the left-hand term, $M \times K \times D \times N$, depends only on the sparse matrix, while the right-hand term, $\frac{Const_1}{BN} + \frac{Const_2}{G}$, is a kernel-specific constant. Therefore, we define $M \times K \times D \times N$ as an MLI, a feature that serves as a linear indicator of L2 cache reads. Instead of collecting the exact L2 cache read volume, we use this feature in the predictive models.

A similar approach allows us to derive a feature representing DRAM read traffic. This estimation is simpler, as it only requires calculating the total number of processed elements:

$$\textit{DRAM\_read}_A = M \times K \times D \times Const_1,$$

$$\textit{DRAM\_read}_B = K \times N \times Const_2,$$

$$\textit{DRAM\_read}_{\text{total}} = M \times K \times D \times Const_1 + K \times N \times Const_2.$$

Based on these derivations, we define two MLIs:

$$MLI_{L2} = RB \times N, \quad MLI_{DRAM} = RB \times G \times Const_1 + K \times N \times Const_2,$$

$$RB = (1 - S^G) \times \frac{M \times K}{G}.$$

Here, RB stands for the number of retained blocks in A. We introduce RB as an auxiliary variable because it is the number of non-empty blocks in A, which can be precisely counted. However, directly extracting this feature from data incurs overhead. To mitigate this, we approximate RB under the assumption of uniform nonzero distribution across A. Our experimental results demonstrate that this approximation is both effective and efficient.

Since M, K, and N are input dimensions, S represents sparsity, and G is the skipping granularity, all these features can be extracted from the input matrices without traversal of the matrix, having an $O(1)$ time complexity. This ensures that our method introduces negligible feature extraction overhead. For GEMM libraries like cuBLAS, which do not skip nonzero elements, we set $S = 0$ and $G = 1$.

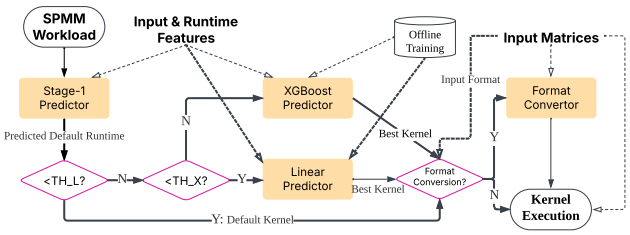


Fig. 6: Workflow of SparseX's MLI-based online prediction module. *TH_L*, *TH_X*, and *default kernel* are automatically decided and updated by the Built-in learner (details in IV-C). The *Predicted Default Runtime* is the default kernel's end-to-end execution time.

IV. Prediction Module and Extensible Design

A. MLI-Based Online Prediction Module

In addition to the use of MLIs, the prediction module features an augmented two-stage lazy-and-light scheme and a combination of an agile linear model and an XGBoost model.

1) Augmented Two-stage Lazy-and-Light Scheme: Inspired by [27], we employed an overhead-aware, two-stage lazy-and-light scheme, which augments the lazy-and-light scheme in the previous work with automatic threshold determination and accommodations of hybrid predictive models.

As shown in Figure 6, the module consists of two prediction stages. Stage-1 estimates the execution time of the default kernel, which is the kernel fastest for small problems– which are especially sensitive to runtime overhead. In our experiments, cuBLAS is chosen as it is highly optimized for dense matrices even if they are small and sparse but converted into dense forms; the conversion into the dense form is much more quicker than into other sparse formats for small matrices. Since this predictor only estimates the execution time of the default kernel, it is a lightweight linear regressor based on only the problem size. If the estimated execution time is smaller than a threshold *TH_L* (automatically determined; discussed later), the framework directly assigns it to the default kernel; otherwise, it proceeds to the next stage.

The Stage-2 predictors utilize hybrid predictive models, including a **Linear predictor** and an **XGBoost predictor**. The non-linear XGBoost predictor is employed only when the estimated execution time is sufficiently large to ensure that the

overhead of invoking the predictive model is negligible. Once the kernel selection is done, the sparse matrix undergoes the preprocessing needed by the kernel, and the selected kernel function executes the SpMM.

This design minimizes the impact of overhead. The Stage-1 predictor requires only basic input features and executing in mere nanoseconds—resulting in negligible overhead even for small problem sizes. The thresholds *TH_L* and *TH_X* are set by the *Built-in Learner* based on the measured overhead of predictive models and feature extraction. Specifically, the lower bound of *TH_L* is defined as 25 times the inference overhead of the linear predictor, while *TH_X* is set to 50 times the inference overhead of the XGBoost model. This ensures that the overhead of using the linear predictor remains below 4% and the XGBoost predictor remains below 2%. XGBoost models are more sophisticated and overall more accurate than the linear model, but also more costly. In many cases, the linear predictor can yield the same results but more quickly.

2) Prediction Targets Definition: Let $T(\text{lib}, A, f, I)$ be the time taken by a library lib to multiply sparse matrix A with I dense matrices of the same shape and size. The sparse matrix A is stored in format f. This time is given by:

$$T(\text{lib}, A, f, I) = L_f(\text{lib}, A) + I \times t(\text{lib}, A, *) \quad (1)$$

where:

- $L_f(\text{lib}, A)$ is the preprocessing time of the library lib on sparse matrix A, which is in format f.
- $t(\text{lib}, A, *)$ is the time taken by the library lib to perform one matrix multiplication on matrix A with dense matrix.
- I is the number of invocations of SpMM on A.

Our goal is to predict the best library lib that minimizes $T(\text{lib}, A, f, I)$ using a predictor.

3) Features: In addition to the MLI features mentioned above, several additional features are used in the predictors, as listed in Table II. All features can be directly extracted from the input matrices in $O(1)$ time.

4) MLI-Based Agile Linear Predictor: The linear predictor is a simple and efficient linear regression model designed for fast execution. It is a regression-based predictor that estimates the execution time of each library kernel independently. As shown in Figure 7, it consists of two components: an overhead regressor that predicts the preprocessing overhead of the kernel and runtime regressors that estimate the kernel's execution time on the GPU. The linear predictor is also used by Stage-1 to estimate the default kernel's runtime. Linear models are in form of a weighted linear combination of relevant features.

TABLE II: Feature set used in Linear and XGBoost predictors. I = included; E = excluded.

Feature	Description	Linear	XGBoost
MLI_{L2}	MLI for L2	I	I
MLI_{DRAM}	MLI for DRAM	I	E
`M, K, N`	Matrix Sizes	I	I
`NNZ`	Number of non-zeros	I	I
`Density`	Density of sparse matrix	I	I
`NNZ * N`	Product of NNZ and N	I	E
`M * Density`	Product of M and density	I	E
`Iteration`	Repetition count	E	I

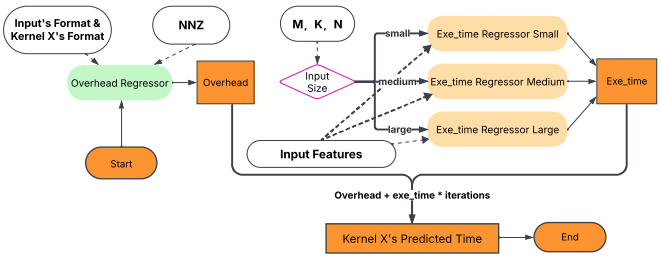


Fig. 7: SparseX's agile linear predictor. The $Overhead$ is the preprocessing overhead of the SpMM library.

Figure 7 shows the workflow of the overall linear predictor. Given an SpMM workload, it first uses the overhead regressor to predict the preprocessing overhead based on the number of non-zeros in the sparse matrix. This prediction is facilitated by a dictionary that maps input and target format pairs to corresponding linear coefficients and intercepts, ensuring efficient estimation of preprocessing costs. Next, the execution time regressor is invoked.

To address the impact of input size variations, we employ a piecewise regression model with distinct weights for small, medium, and large datasets. A key challenge for linear modeling is that memory throughput varies depending on whether the input fully utilizes all SMs. To accommodate this, we categorize inputs into three size groups and assign each group a dedicated regressor. By default, SparseX automatically partitions the dataset based on problem size. Once categorized, the appropriate regressor processes the input features and computes the execution time. The final prediction for the kernel's runtime is obtained by summing the overhead and execution time estimates, and the kernel with the shortest predicted runtime is selected.

The linear predictor is highly efficient. Under our experiments with eight kernel functions, the inference overhead is just 2.52 microseconds. However, it does not guarantee a performance gain in every case. Thus, we introduce a threshold *TH_L* to determine when the linear predictor should be used. Despite this precaution, experiments indicate that the linear predictor is the optimal choice in most scenarios. Although the overhead of SparseX may increase as more libraries are incorporated, several factors mitigate this concern. First, with more libraries, the likelihood of the default kernel being suboptimal increases, thereby more potential benefits from using the predictor. Second, only libraries that perform well in specific input scenarios are included. If a new library outperforms an existing one across most cases, it replaces the older library. Third, the user may easily control the maximum number of libraries for SparseX to consider if the overhead becomes significant.

5) XGBoost Model: XGBoost [31] is a scalable and efficient machine learning model based on gradient boosting decision trees. It builds an ensemble of weak learners—typically decision trees—by sequentially adding models that correct the residual errors of prior models. As a non-linear model, it has

a higher capacity and flexibility in modeling relations, and hence potentially more predictive power than linear models can offer. Some libraries (e.g., cuBLAS) internally dispatch a matrix multiplication call to one of multiple versions of kernels depending on undisclosed heuristics. The non-linearity of XGBoost makes it possible to model such behaviors.

Despite its efficiency, the decision tree models are still computationally expensive, introducing 0.8 seconds of overhead in our experiments. To optimize inference speed, we leverage parallel inference across decision trees. We find that increasing the number of trees while reducing the depth of each tree can help balance accuracy and efficiency, so we set the tree count to 480 and depth to 6, which maintains high precision while minimizing inference time. We employ XGBoost as a classification model that directly outputs the optimal kernel id. This approach is more efficient than constructing separate regression models for each kernel, further reducing computational overhead while maintaining accurate selection.

B. Extension Interface

SparseX is designed to enable seamless integration of matrix multiplication libraries through a well-structured extension interface. This interface consists of two components: the *library protocol*, which defines a general protocol for a new library to follow in its top-level SpMM function definition, and the *user configuration file*, which provides a user-friendly interface for adding or modifying the library collection effortlessly.

1) Library Protocol: For a new library to be integratable into SparseX, it should provide a top-level SpMM function interface that accepts the following parameters: (i) M, K, N: Sparse matrix dimensions; (ii) device pointers for matrices A, B, and C; (iii) row and column index arrays of matrix A; (iv) Row permutation indexes for workload balancing.

Additionally, the function should use opt_dict to store any library-specific arguments. For example, Jigsaw requires $d\_block\_sparsity\_metadata$ to store the position code required by the 2:4 format in SpTCs.

If the library uses some special sparse formats, it should provide functions that can convert common Sparse formats to the special format; the definitions of those functions should adhere to a predefined form. Some libraries have certain hyperparameters. For instance, CLASP explicitly requires the column vector length as a hyperparameter. Some other libraries (e.g., Sputnik) automatically determine the needed hyperparameters. The way SparseX handles this is to include a user-defined $hyper\_dict$ and maintain a list of available choices. For example, CLASP supports column vector lengths of 2, 4, and 8. These choices should be reflected in the user configuration file (detailed next), allowing users to explicitly specify the desired options or include all supported values.

2) User Configuration File: SparseX is equipped with a configuration file in JSON format, through which, the user can easily add or modify the candidate libraries and other settings. For a library, the relevant fields in the configuration file include (i) library name; (ii) library hyperparameter list; (iii) path of the library object file; (iv) name of the top-level SpMM function; (v) list of the format conversion functions.

3) Linking: SparseX offers both C and Python packaging. To use SparseX for SpMM, the user just needs to include the corresponding header file or module into their application and replace the original SpMM calls with calls to SparseX_spmm.

C. Built-in Learner

To enable automatic adaptation to new libraries and hardware, SparseX features a built-in learner module. It can be invoked manually or get triggered automatically when certain fields in the configuration file are updated.

When the library collection changes, SparseX links new libraries and recompiles the framework. Its built-in learner executes the new libraries on training datasets, updates the predictive models, and measures the overhead. It automatically updates the thresholds *TH_L* and *TH_X* according to the following formulas:

$$TH_L = N_1 \times \text{InferenceTime}_{\text{Linear}},$$
$$TH_X = N_2 \times \text{InferenceTime}_{\text{XGBoost}}.$$

where N_1 and N_2 are either user-defined or determined via binary search, with default values $N_1 = 25$ and $N_2 = 50$.

Next, the learner selects the default kernel. With the threshold *TH_L*, the learner identifies all input cases where at least one kernel achieves end-to-end execution time below this threshold. From these cases, it compiles a list of the best-performing kernels and selects as the default the kernel that appears most frequently as optimal. Finally, it retrains the stage-1 predictor based on the default kernel choice.

D. Example: Adding Sputnik to SparseX

This subsection illustrates how to integrate Sputnik via the extension interface. The workflow has three parts: implement the top–level SpMM wrapper (library protocol), declare the library in the user configuration file, and rely on the built–in learner to adapt automatically.

1) Top–level SpMM wrapper (Library Protocol): Following the protocol, the entry function should take the form below:

```
cudaError_t sparsex_spmm_sputnik(
  int M, int K, int N, // input size
  void* dA_values,     // CSR value
  int*  dA_rowptr,     // CSR rows
  int*  dA_colind,     // CSR columns
  void* dB, void* dC,  // Dense matrices
  int*  d_row_permute, // Row permutation
  OptDict& opt_dict    // Additional Args
  HyperDict& hyper_dict); // Kernel Choices
```

The library should also expose a conversion function for format conversion and preprocessing:

```
cudaError_t coo_to_csr_with_row_swizzle(
    // matrix shape and sizes
    int M, int K, int nnz,
    // COO inputs (device)
    int* d_coo_rows, d_coo_cols, void* d_coo_val,
    // CSR outputs (device)
    int* d_csr_rows, d_csr_colind, void* d_csr_val,
    // row permutation
    output int* d_row_permute);
```

2) User Configuration File (JSON): The next step is to declare Sputnik in the JSON configuration by adding it to the list of libraries (`hyperparameters` is empty since Sputnik only has one configuration):

```
{
  "name": "sputnik",
  "hyperparameters": [],
  "library_path": "lib/libsparsex_sputnik.a",
  "spmm_entry": "sparsex_spmm_sputnik",
  "conversions": ["coo_to_csr_with_row_swizzle"]
} // Sputnik
```

3) Linking and Learning: After the configuration update, SparseX recompiles and links the new static library. The built–in learner benchmarks Sputnik on the training set, measures preprocessing/launch overheads, retrains the predictive models, and updates the thresholds. Because Sputnik exposes no discrete hyperparameters, it is treated as a single candidate kernel. No additional user effort is required.

V. Evaluation

A. Experiments Setup

1) Dataset:

TABLE III: SuiteSparse graph collection.

		#Vertices	#Edges	Sparsity	#Graphs
Small	*Avg*	426	4.97k	86.56%	444
	Med	430	2.19k	91.48%	
Medium	*Avg*	3.6k	93.2k	99.33%	724
	Med	2.6k	24.1k	99.61%	
Large	*Avg*	22.6k	878k	99.78%	188
	Med	20.5k	229k	99.91%	

a) SuiteSparse: We tested the libraries across a wide range of matrix sizes and sparsity levels. The sparse matrices include 1356 sparse matrices from the SuiteSparse Matrix Collection [29], a widely used repository of real-world sparse datasets. These matrices are used as they cover a broad range of matrices in sizes and sparsity patterns. They are derived from real-world graphs in form of the adjacency matrices of those graphs, which fall into small, medium, and large groups as shown in Table III. The matrices come in COO sparse format in the default collection, which is the input format used in all the experiments.

b) Large Language Model with Activation Sparsity: We also extracted a dataset from the Open Pre-trained Transformer(OPT) models [32], which include a ReLU activation between the up-projection and down-projection layers of each Feed Forward Network (FFN). Due to this sparse activation, the down-projection can naturally be formulated as an SpMM operation. We applied SparseX to replace the original GEMM kernel and measured the performance gains. Specifically, we use OPT-2.7B and OPT-6.7B models, with $D_{\text{model}} = 2560$ and 4096, respectively. We use the open-source checkpoints from HuggingFace, applying only the original ReLU activations without additional modifications.

It's worth noting that other LLMs such as LLaMA2 [33] and Falcon [34], while not originally implemented with sparse activation, have been shown to support relufied variants with minimal accuracy degradation [18].

2) Datatype: We use FP16 precision for all our experiments in this paper, as FP16 serves as a faster alternative to FP32 and is widely adopted in SpMM libraries [9], [10], [12], [16].

3) Hardware Platforms: We conducted the measurements on two popular NVIDIA GPU models: the RTX 4090 and the A100-40G.

4) Major Candidate Libraries: **cuBLAS** (CUDA Basic Linear Algebra Subroutines) is NVIDIA linear algebra library. It is not specifically designed for sparse matrices, treating sparse matrices the same as dense matrices in the computation. Nonetheless, it is a highly tuned library, still offering a meaningful reference point.

cuSPARSE (CUDA Sparse Basic Linear Algebra Subroutines) [16] is an NVIDIA library optimized for sparse operations, performing best at high sparsity levels.

Sputnik [8] is an SpMM library designed for fine-grained sparsity in deep learning. It operates using FMA instructions on CUDA cores. Sputnik stores sparse matrices in CSR format and employs a 1D tiling approach combined with subwarp tiling, mapping output tiles into groups of four rows.

CLASP (Column-Vector Pruning-Aware SpMM Kernel) [10] is an SpMM library optimized for the column-vector sparsity pattern and designed specifically for TCUs on GPU. CLASP utilizes column-vector format, and assigns each thread block to process a $v \times 64$ tile in the output matrix, where v is the column vector size. We included $v = 2, 4, 8$ in our study.

Jigsaw [12] is an SpMM library designed for SpTCs. It exploits the 2:4 sparsity format supported by SpTCs. Jigsaw provides a utility that reorders the rows and columns in each tile of a given sparse matrix, in order to conform to the required pattern. The reordering is time-consuming, far exceeding the SpMM kernel execution time, intended for offline preprocessing only. Moreover, the reordering may fail, depending on the non-zero distributions of the sparse matrix. Jigsaw also supports different column vector lengths (v). We use $v = 16, 32$ in our study.

Because Jigsaw cannot work on the whole dataset, we use Sputnik—the overall most performant library—as the fallback option in Jigsaw for a matrix that does not conform to the required 2:4 sparse pattern. We use **Jigsaw*** to distinguish this variant from the original Jigsaw. In the original dataset, only about 5% of the matrices conform to the 2:4 sparse pattern; in order to have a meaningful mix of matrices with structured and unstructured sparsity for a comprehensive evaluation, we offline reordered another 25% of the matrices randomly into the 2:4 sparse pattern.

B. Performance on SuiteSparse Dataset

Table IV presents the speedups achieved by SparseX compared to individual libraries and an oracle. The oracle represents the upper bound of the speedups produced by perfect selections of libraries without any overhead. The reported speedups are averaged over the entire dataset, accounting for all overheads introduced by SparseX's MLI-based online prediction module. Each column in the table corresponds to a specific workload setting.

TABLE IV: Speedup of SparseX compared to individual matrix-multiplication libraries and other predictor based methods on RTX 4090. N is the number of columns of the output matrix. "Oracle" represents the theoretical upper bound.

Method	N=128			N=512			N=8192		
	Iter=1	Iter=100	Iter=10k	Iter=1	Iter=100	Iter=10k	Iter=1	Iter=100	Iter=10k
Sputnik	20.76	1.11	1.44	13.24	1.08	1.28	3.83	1.06	1.27
cuBLAS	4.01	3.71	6.77	2.55	4.49	7.03	2.39	5.14	5.85
cuSPARSE	1.14	4.04	7.85	1.62	7.31	11.03	7.07	18.05	20.70
CLASP-2	19.64	0.99	1.29	12.82	1.06	1.30	3.81	1.27	1.56
CLASP-4	19.74	1.15	1.59	13.10	1.34	1.74	4.07	1.73	2.13
CLASP-8	19.86	1.14	1.59	13.25	1.21	1.56	3.97	1.39	1.71
Jigsaw*-16	18.86	0.94	1.90	12.59	1.03	1.62	3.75	1.18	2.04
Jigsaw*-32	18.86	0.96	1.04	13.44	1.02	1.09	3.90	1.02	1.08
WISE	81.70	2.92	1.47	48.91	2.75	1.16	13.78	1.52	1.28
SEER	3.14	1.80	5.41	2.45	1.69	2.02	1.39	1.23	1.87
Oracle*	0.98	0.85	0.94	0.96	0.90	0.99	0.90	0.94	0.99

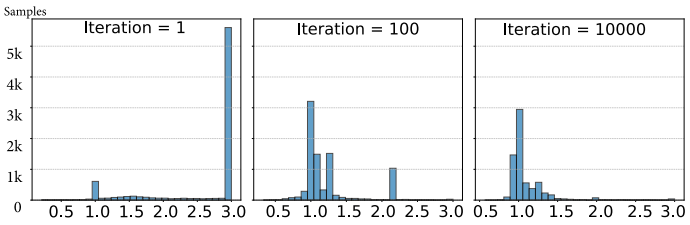


Fig. 8: Distribution of speedup of SparseX compared to Sputnik, under different iterations. X-axis is the speedups over Sputnik, averaged across all input matrices.

Across nearly all tested configurations, SparseX consistently outperforms individual libraries, with the highest observed speedup reaching 20.7× over cuSPARSE. When $Iter = 1$ and $N = 128$, SparseX achieves more than 10× speedup over CLASP, Jigsaw, and Sputnik. This is because these libraries incur significant preprocessing overhead in this setting, whereas selecting alternatives like cuBLAS and cuSPARSE avoids this cost. Conversely, when $Iter = 10k$ and $N = 8192$, the problem size is large enough that preprocessing overhead becomes negligible. In this case, Jigsaw-32 with Sputnik as an alternative performs best in most scenarios. Notably, in this scenario, SparseX already achieves 99% of the theoretical upper bound, implying that even Oracle can achieve at most a 1% additional speedup.

SparseX demonstrates its effectiveness by delivering substantial speedup across various configurations. The total problem size increases as N and Iter increase, and it's clear that on larger problems, SparseX better taps into the potential. When $Iter = 100$, from $N = 128$ to $N = 8192$, SparseX achieves 85%, 90%, 94% of Oracle's performance; when $N = 8192$, from $Iter = 1$ to $Iter = 10k$, SparseX achieves 90%, 94%, 99% of Oracle's performance. The reason is that, as the problem size increases, more complex but accurate models can be used (evidence in Table V, discussed later). Also, the results indicate that for $N \geq 512$ and $Iter \geq 100$, SparseX achieves over 90% of the Oracle speed. This suggests that, even under ideal conditions, there is a theoretical limit of approximately 10% further improvement. In scenarios where $N \geq 512$ and $Iter \geq 10k$, the remaining gap is less than 1%, demonstrating that SparseX is approaching the optimal performance. Given

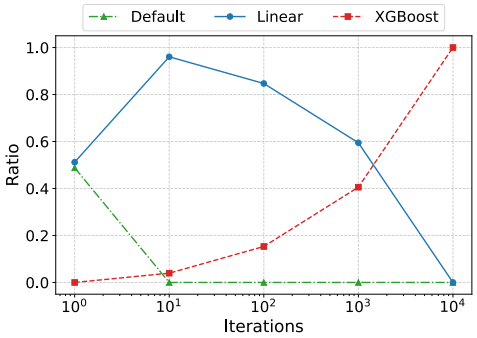


Fig. 9: Each predictor's selection ratio across iterations. The predictors are selected based on the two-stage scheme of SparseX, detailed in Figure 6. The optimal ratio reflects how often it is selected across all inputs.

that large SpMM workloads dominate execution time in real-world machine learning applications, these results highlight the practical effectiveness of the proposed approach.

There are only a few cases where SparseX slightly underperforms a library. When $N = 128$ and $Iter = 100$, CLASP-2 achieves slightly better results, and Jigsaw is faster on compatible input. For such scenario of $N = 128$ and $Iter = 100$, which represents an extremely small workload, the execution is highly sensitive to preprocessing overhead. As shown in Figure 9, when the iteration count is low, many inputs default to the baseline kernel. Given that $N = 128$ is the smallest tested workload, SparseX often selects the default kernel, leading to a slight performance disadvantage. In the second case, CLASP-2 slightly outperforms SparseX because it is already the best choice for most inputs in this setting. Consequently, the predictive model has little room for improvement, and the additional inference overhead, although minimal, results in a slight slowdown. Another factor contributing to the suboptimal performance at $N = 128$ is the role of the L1 cache, which has a larger impact on performance at small input size. More failure case studies are detailed in Section V-G1.

To provide more detailed view of the results, Figure 8 shows the speedup distribution over Sputnik. At low iteration counts, SparseX outperforms Sputnik significantly by selecting lower-overhead libraries than Sputnik which has a high preprocessing cost. As the iteration count increases, Sputnik becomes more competitive and often emerges as the optimal choice. However, many scenarios still favor alternative selections with substantial performance gains. Meanwhile, thanks to its overhead-aware design, SparseX rarely incurs substantial slowdowns even when its prediction is suboptimal.

As shown in Figure 9, at low iteration counts the MLI-based linear predictor handles most cases and is key to performance. Table V reports its accuracy in choosing the optimal kernel: 88.83% (iter 1), 65.09% (iter 100), and 70.97% (iter 10k). The predictor models preprocessing overhead and kernel time separately. Overhead is easy to estimate—largely a function

of nnz—but as iterations grow, its share of total time shrinks, reducing overall accuracy. However, this does not pose a significant problem, as the linear predictor primarily operates in scenarios where the problem size is small.

C. Overhead Breakdown

TABLE V: Comparison of overhead and prediction accuracy among predictive models in SparseX. Overhead is defined as the inference time divided by the kernel execution time.

Iter	SparseX-Linear		SparseX-XGBoost		SparseX-Combined	
	Overhead	Accuracy	Overhead	Accuracy	Overhead	Accuracy
1	5.67%	88.83%	1589%	89.85%	**2.67%**	**73.92%**
10	0.8732%	76.71%	244.62%	87.62%	**1.93%**	**80.80%**
100	0.1638%	65.09%	45.89%	89.70%	**1.92%**	**79.17%**
1k	0.0291%	69.00%	8.15%	96.54%	**1.28%**	**95.20%**
10k	0.0053%	70.97%	1.47%	96.84%	**1.47%**	**96.84%**

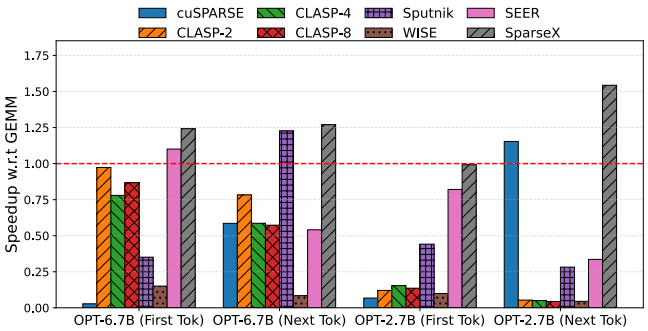


Fig. 10: Speedup of static SpMM libraries, SparseX, and other predictor-based methods compared to original GEMM in OPT model's downprojection layer. "First Tok": the first token generation; "Next Tok": the following token generation.

SparseX uses the hybrid predictors and the two-stage lazy-and-light prediction scheme to achieve a balance between prediction accuracy and efficiency. To evaluate the effectiveness of this design, we analyzed the selection ratios of different predictors within SparseX. As Figure 9 shows, the selection behavior dynamically shifts as the iteration count increases. When SpMM is executed only once, SparseX uses the default kernel or the Linear model. For iteration counts between 10 and 1,000, the linear model becomes dominant.

Figure V quantifies the accuracy and overhead of different predictors across iteration settings. The two-stage scheme allows SparseX to adaptively balance predictor involvement, achieving an accuracy exceeding 73.92% while keeping the overhead consistantly below 2.67%. As the iteration count increases and the more accurate XGBoost model is used, accuracy reaches as high as 96.84%. Note that we deem a prediction correct when the selected kernel's runtime is within 2% of the true optimum—reasonable given measurement noise and the negligible gains from near-tie micro-optimizations.

For iteration counts of 1,000 or lower, on more than half of the inputs, SparseX uses the linear predictor. This confirms that the linear model provides a favorable balance of accuracy and efficiency. The ability to maintain high accuracy while minimizing overhead ensures that SparseX remains practical even in scenarios with moderate iteration counts.

D. Comparison with Other Predictor-Based Models

We further compare SparseX with other predictor-based models. There are no mature work directly on developing predictors for SpMM library selections. The closestly related work in recent efforts are WISE [24] and Seer [25]. Both are for SpMV rather than SpMM. WISE [24] extracts features from sparse matrices and employs decision tree models to predict the performance of different SpMV methods. Seer [25] addresses a similar problem but adopts a two-stage decision tree design: a selector model determines whether feature extraction is required; if not, it falls back on trivially known features such as matrix size and sparsity. To adapt these approaches to the SpMM setting, we extend their models by incorporating dense matrix size and iteration count as additional features, and use GPU for faster feature extraction.

Table IV reports the average speedup of SparseX over these two predictor-based methods. Compared with WISE, SparseX achieves an average 81.7× speedup at the smallest problem size. This large margin comes from SparseX's overhead-aware design: WISE incurs significant costs from feature extraction and decision-tree prediction, which dominate in small problem regimes. At larger problems, WISE's additional features do not substantially improve predictive accuracy beyond what our MLI-based XGBoost model already captures, and SparseX still delivers around 30% speedup even in the largest setting.

Seer improves upon WISE by explicitly incorporating feature-extraction overhead into its design. However, it faces two limitations. First, although Seer claims its small decision tree model has negligible runtime cost, this overhead is still higher than that of SparseX's lightweight linear model. In addition, mispredictions by Seer's selector model can trigger unnecessary feature extraction, leading to considerable overhead. Second, Seer's "Known Predictor" relies only on trivial features such as matrix size and sparsity, missing MLI-based features. Even with feature extraction enabled, its simple tree model restricts accuracy compared to SparseX's XGBoost model. As a result, even in large problem sizes where feature-extraction overhead becomes negligible, SparseX still provides over 23% average speedup.

Other predictor-based methods [26], [27] also tend to rely on costly feature extraction or heavy models; without our lightweight MLI features and fully overhead-aware design, it is difficult to balance accuracy and efficiency across workloads.

E. Performance on LLMs

We also test SparseX on the Open Pre-trained Transformer(OPT) LLM models. We use a batch size of 16 on 100 groups of random prompts from the WikiText-103 validation set [35]. The runtime comparison of the down-projection layer (the second fully connected layer in OPT's FFN) is shown in Figure 10. Although the ReLU activation often

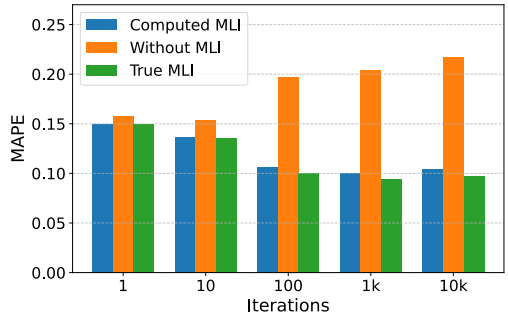


Fig. 11: Median MAPE of the linear predictors. "Computed MLI" is what SparseX uses, "Without MLI" is when MLIs are excluded from SparseX, "True MLI" is when the true MLI features extracted from inputs are used.

yields input matrices with over 90% sparsity, the overhead of format conversion and preprocessing greatly limits the effectiveness of existing SpMM libraries. Moreover, the fastest SpMM kernel varies significantly across different scenarios. In contrast, SparseX achieves consistent speedup in most cases. For challenging settings such as OPT-2.7B during first-token generation, where overall sparsity is lower and hence no much potential for SpMM libraries, unlike the over 50% performance loss by other SpMM libraries, SparseX successfully limits the performance loss to a small percentage.

The down-projection operation itself accounts for approximately 25%–30% of the first-token generation time, and about 20% of the next-token generation time. Under our setting with a sequence length of 500 input and 500 generated tokens, SparseX achieves an end-to-end speedup of 4.55% on OPT-6.7B and 7.24% on OPT-2.7B. It should be noted that the overall performance gain is bounded by the proportion of sparsified operations in the model; if more sparse operations are introduced, the benefits of SparseX would increase further.

F. Benefits Brought by MLI

To evaluate the precision of the predicted runtime of kernels by the linear predictor, we calculate the Mean Absolute Percentage Error (MAPE), a common metric in regression [36]. It measures the average deviation of predicted values ($\hat{y}_i$) from actual values (y_i) across all observations: MAPE $= \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i-\hat{y}_i}{y_i}\right| \times 100\%$, where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of observations. Figure 11 presents the median MAPE results, showing that the linear predictor maintains a relative error below 15% at low iteration counts and around 10% when iterations increase. Considering measurement fluctuations due to system noises, this level of accuracy is sufficient for most scenarios.

Figure 11 further demonstrates the effectiveness of the MLI features. SparseX computes MLI features using M, K, N, NNZ, and uses other features detailed in Table II. Its MAPE is as represented by the blue bars in the graph. If only alternative features— $M, K, N, NNZ, Density, NNZ*$ $N, M * Density$—are used, a linear regressor can still be constructed. However, as the orange bars indicate, excluding MLI features results in significantly higher MAPE, particularly when iteration counts are high enough that the overhead regressor alone is insufficient. Meanwhile, we also consider using true MLI features by extracting the number of retained blocks in the sparse matrix, as detailed in Section III. While these true MLI features should theoretically provide more structural insights, the green bars indicate that their impact is minimal. This is likely because, while large-scale sparse matrices may exhibit uneven distributions, at the level of each miniblock, assuming an even distribution gives reasonable approximations. Additionally, the accuracy of the XGBoost model suggests that such features are not always essential.

TABLE VI: Speedup of SparseX compared to individual matrix-multiplication libraries on NVIDIA A100. N is the number of columns of the output matrix. Oracle represents the theoretical upper bound of speedup.

MM Library	N=128			N=512			N=8192		
	Iter=1	Iter=100	Iter=10k	Iter=1	Iter=100	Iter=10k	Iter=1	Iter=100	Iter=10k
Sputnik	10.15	1.41	1.53	7.88	1.34	1.36	3.99	1.45	1.55
cuBLAS	1.11	3.18	5.90	1.16	4.53	6.97	2.69	8.91	8.44
cuSPARSE	2.68	13.59	21.49	5.43	23.93	31.21	20.89	70.39	95.34
CLASP-2	9.05	1.10	1.15	7.28	1.17	1.21	3.88	1.63	1.82
CLASP-4	8.86	1.27	1.44	7.23	1.38	1.51	3.93	1.87	2.05
CLASP-8	8.94	1.28	1.44	7.27	1.30	1.40	3.83	1.51	1.64
Jigsaw*-16	9.94	1.60	1.59	7.75	1.26	1.32	3.92	1.20	1.39
Jigsaw*-32	9.94	1.36	1.57	7.75	1.24	1.32	3.90	1.06	1.12
Oracle*	0.83	0.87	0.98	0.91	0.90	0.98	0.94	0.93	0.99

G. Portability Analysis

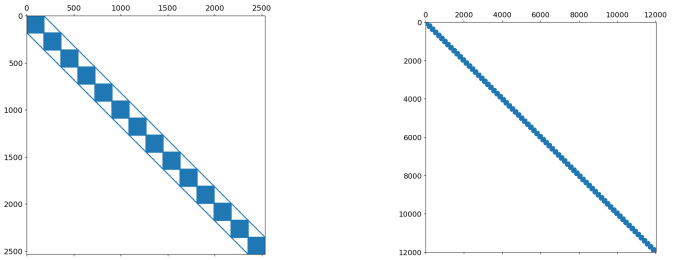
To demonstrate the portability of SparseX across different platforms, we applied the same settings and re-evaluated its performance on an NVIDIA A100 GPU. As shown in Table VI, SparseX achieves even greater speedup, up to 95.34x over cuSPARSE, on the A100 while maintaining the same trend—larger problem sizes yield near-optimal performance. Notably, the inference overhead on the A100 machine is slightly higher due to its lower-end CPU. However, SparseX's overhead-aware mechanism dynamically adjusts the selection thresholds, effectively balancing accuracy and overhead to maintain strong performance.

1) Failure Cases: **(a) Block structure.** One matrix, qc2534, contains a special pattern in its non-zero elements as shown in Figure 12(a), which creates a gap between computed (with random non-zero distribution assumption) and true MLI values and leads to a miss: oracle selects CLASP (v=8) vs. CLASP (v=2) as chosen by SparseX.

(b) Short-range correlation & L1 reuse. Another matrix, ex19, contains a diagonal distribution of non-zeros as shown in Figure 12(b). Again the regular sparsity pattern causes disparity from the random distribution assumed by the performance model in SparseX. Sputnik gets a higher L1 hits (36.15%) than SparseX estimates, leading to a miss in prediction: oracle selects Sputnik vs. CLASP (v=4) as chosen by SparseX.

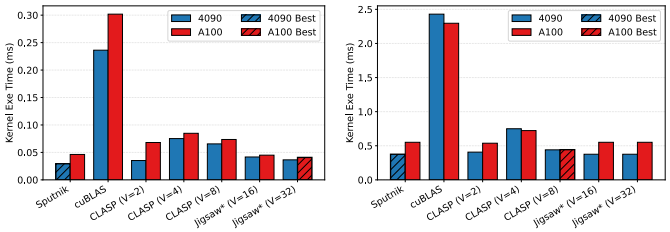
The miss predictions in both cases stem from the regular sparsity patterns in the matrices, suggesting an opportunity for SparseX to be further enhanced if it can be made conscious

of regular sparsity patterns efficiently. However, capturing all kinds of patterns would add significant overhead. For many applications (e.g., sparse activations in DNN), such patterns are rare. Even for graph-structured data in SuiteSparse, which can exhibit distinct patterns, our experiments reported earlier in this section show that SparseX consistently achieves near-optimal speedup on average.



(a) qc2534, N=128 , Iter=1k (b) ex19, N=2048, Iter=1

Fig. 12: Sparsity of two matrices forms regular patterns.



(a) ACTIVSg2000, N=2048 (b) Goodwin_023, N=8192

Fig. 13: Kernel time of matrices `ACTIVSg2000` and `Goodwin_023` on RTX 4090 and A100.

H. Case Studies

1) Variant Behavior Cross Platform: We present two cases to illustrate cross-platform variability. As shown in Figure 13, though the relative ranking of libraries is broadly consistent, a key divergence is that A100 favors kernels with larger thread-block granularity (e.g., Jigsaw (V=32) on `ACTIVSg2000`, CLASP (V=8) on `Goodwin_023`), whereas small-block kernels (e.g., Sputnik, CLASP (V=2)) show large performance gaps than their counterparts on RTX4090. We attribute this to A100's lower CUDA/Tensor-core count and different scheduling/clock characteristics, which make large numbers of small thread blocks harder to saturate both the memory and compute resources. Despite these hardware-specific effects, SparseX accounts for platform differences during learning, selects the best kernels and achieves speedups on both GPUs.

VI. Related Work

Previous sections have already covered the studies on SpMM libraries most closely related to this work. We briefly summarize some other work on optimizing SpMM.

Apart from research efforts that focus on utilizing specific computing units, there are also efforts aimed at enabling collaboration between heterogeneous processors. TC-GNN [14] utilizes CUDA core threads for data loading and Tensor Core Units (TCUs) for computation. SparTA [7] decomposes the sparse matrix into segments that are assigned to different types of compute units to better exploit hardware capabilities. In addition, there have been studies on building compilers that generate optimized, input-specific kernel functions. The Tensor Algebra Compiler (TACO) [37] is a compiler-based framework for high-performance sparse linear algebra and tensor computations; accelerators such as ExTensor [38] are built atop this infrastructure to further boost performance. Xin et al. [39] proposed a method to explore SpMM loop implementations in a three-dimensional design space defined by *Tiling*, *Parallelism*, and *Scheduling*. EC-SpMM [23] and LO-SpMM [30] search for optimal tiling configurations and generate efficient kernel functions tailored to specific input matrices and hardware constraints. Although SparseX shares some conceptual similarity with these compiler-based approaches in exploring configuration spaces, its objectives are fundamentally different. These works focus on optimizing the kernel's configuration for specific input, whereas SparseX addresses the broader challenge of synergizing entirely different library implementations, each potentially leveraging different GPU compute units. We ran small scale tests on 100 randomly selected matrices from the SuiteSparse dataset we use, showing that the kernels produced by TACO are on average 8.2x slower than the ones selected by SparseX. Also, it reveals that the compilation overhead of TACO is on avearge 4,527x of the preprocessing time of the libraries (e.g., Sputnik) included in our study, making such compiler-based approaches not practical for dynamic sparsity. Meanwhile, SpMM kernels produced by compiler-based tools can serve as candidates in SparseX to further boost performance.

VII. Conclusions

This paper introduces SparseX, the first end-to-end, extensible framework for synergizing libraries for GPU-based SpMM. Experiments demonstrate that SparseX achieve significant performance improvements across diverse use cases. Moreover, the framework offers easy extensibility and cross-hardware portability, making it a robust and adaptable solution for optimizing SpMM computations. Additional data and artifacts are available at 10.6084/m9.figshare.30639536.v1 [40].

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Award No. OAC-2417850, USDA NIFA under Award No. P24-001771, NIH under Award No. 1R01HD108473-01, and U.S. Department of Energy (DOE), under Award No. 78284. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, USDA, NIH or DOE.

REFERENCES

[1] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[2] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *arXiv preprint arXiv:1503.07241*, 2015.

[3] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–12.

[4] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[5] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational dropout sparsifies deep neural networks," in *International conference on machine learning*. PMLR, 2017, pp. 2498–2507.

[6] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization," *arXiv preprint arXiv:1712.01312*, 2017.

[7] 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 *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 213–232.

[8] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[9] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 359–371.

[10] 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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 135–147.

[11] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[12] K. Zhang, X. Liu, H. Yang, T. Feng, X. Yang, Y. Liu, Z. Luan, and D. Qian, "Jigsaw: Accelerating spmm with vector sparsity on sparse tensor core," in *Proceedings of the 53rd International Conference on Parallel Processing*, 2024, pp. 1124–1134.

[13] J. Xin, X. Ye, L. Zheng, Q. Wang, Y. Huang, P. Yao, L. Yu, X. Liao, and H. Jin, "Fast sparse deep neural network inference with flexible spmm optimization space exploration," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[14] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding, "{TC-GNN}: Bridging sparse {GNN} computation and dense tensor cores on {GPUs}," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 149–164.

[15] R. Fan, W. Wang, and X. Chu, "Dtc-spmm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 253–267.

[16] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, vol. 12, 2010.

[17] M. Federici, D. Belli, M. Van Baalen, A. Jalalirad, A. Skliar, B. Major, M. Nagel, and P. Whatmough, "Efficient llm inference using dynamic input pruning and cache-aware masking," *arXiv preprint arXiv:2412.01380*, 2024.

[18] I. Mirzadeh, K. Alizadeh, S. Mehta, C. C. Del Mundo, O. Tuzel, G. Samei, M. Rastegari, and M. Farajtabar, "Relu strikes back: Exploiting activation sparsity in large language models," *arXiv preprint arXiv:2310.04564*, 2023.

[19] M. Kurtz, J. Kopinsky, R. Gelashvili, A. Matveev, J. Carr, M. Goin, W. Leiserson, S. Moore, N. Shavit, and D. Alistarh, "Inducing and exploiting activation sparsity for fast inference on deep neural networks," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5533–5543.

[20] NVIDIA Corporation, "Nvidia a100 tensor core gpu architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020, accessed: 2025-03-24.

[21] A. Mehrabi, D. Lee, N. Chatterjee, D. J. Sorin, B. C. Lee, and M. O'Connor, "Learning sparse matrix row permutations for efficient spmm on gpu architectures," in *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2021, pp. 48–58.

[22] Z. Xue, M. Wen, Z. Chen, Y. Shi, M. Tang, J. Yang, and Z. Luo, "Releasing the potential of tensor core for unstructured spmm using tiled-csr format," in *2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 2023, pp. 457–464.

[23] J. Lin, H. Zhang, X. Shi, J. Sun, X. Yu, J. Yao, and G. Sun, "Ec-spmm: Efficient compilation of spmm kernel on gpus," in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 21–30.

[24] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Wise: Predicting the performance of sparse matrix vector multiplication with machine learning," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023, pp. 329–341.

[25] R. Swann, M. Osama, K. Sangaiah, and J. Mahmud, "Seer: Predictive runtime kernel selection for irregular problems," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2024, pp. 133–142.

[26] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, 2018, pp. 94–108.

[27] W. Zhou, Y. Zhao, X. Shen, and W. Chen, "Enabling runtime spmv format selection through an overhead conscious method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 80–93, 2019.

[28] T. Gale, D. Narayanan, C. Young, and M. Zaharia, "Megablocks: Efficient sparse training with mixture-of-experts, 2022," *URL https://arxiv.org/abs/2211.15841*.

[29] T. A. Davis, "Algorithm 1000: Suitesparse:graphblas: Graph algorithms in the language of sparse linear algebra," vol. 45, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3322125

[30] J. Lin, J. Sun, X. Shi, H. Zhang, X. Yu, X. Wang, J. Yao, and G. Sun, "Lo-spmm: Low-cost search for high-performance spmm kernels on gpus," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 4, pp. 1–25, 2024.

[31] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 785–794.

[32] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[33] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[34] E. Almazrouei, H. Alobeidli, A. Alshamsi, A. Cappelli, R. Cojocaru, M. Debbah, É. Goffinet, D. Hesslow, J. Launay, Q. Malartic *et al.*, "The falcon series of open language models," *arXiv preprint arXiv:2311.16867*, 2023.

[35] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," *arXiv preprint arXiv:1609.07843*, 2016.

[36] A. Botchkarev, "Performance metrics (error measures) in machine learning regression, forecasting and prognostics: Properties and typology," *arXiv preprint arXiv:1809.03006*, 2018.

[37] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[38] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[39] J. Xin, X. Ye, L. Zheng, Q. Wang, Y. Huang, P. Yao, L. Yu, X. Liao, and H. Jin, "Fast sparse deep neural network inference with flexible spmm optimization space exploration," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7.

[40] [n. d.] SparseX: Synergizing GPU Libraries for Sparse Matrix Multiplication on Heterogeneous Processors - Research Paper Artifact. https://doi.org/10.6084/m9.figshare.30639536.v1.