AlphaSparse: Generating High Performance SpMV Codes Directly from Sparse Matrices

Zhen Du*‡, Jiajia Li†, Yinshan Wang*, Xueqi Li*, Guangming Tan*, Ninghui Sun*
* *Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
† *North Carolina State University, Raleigh, NC, USA*
‡ *University of Chinese Academy of Sciences, Beijing, China*
{duzhen18z,wys,lxq,tgm,snh}@ict.ac.cn, jli256@ncsu.edu

***Abstract*—Sparse Matrix-Vector multiplication (SpMV) is an essential computational kernel in many application scenarios. Tens of sparse matrix formats and implementations have been proposed to compress the memory storage and speed up SpMV performance. We develop AlphaSparse, a superset of all existing works that goes beyond the scope of human-designed format(s) and implementation(s). AlphaSparse automatically *creates novel machine-designed formats and SpMV kernel implementations* entirely from the knowledge of input sparsity patterns and hardware architectures. Based on our proposed Operator Graph that expresses the path of SpMV format and kernel design, AlphaSparse consists of three main components: Designer, Format & Kernel Generator, and Search Engine. It takes an arbitrary sparse matrix as input while outputs the performance machine-designed format and SpMV implementation. By extensively evaluating 843 matrices from SuiteSparse Matrix Collection, AlphaSparse achieves significant performance improvement by 3.2× on average compared to five state-of-the-art artificial formats and 1.5× on average (up to 2.7×) over the up-to-date implementation of traditional auto-tuning philosophy.**

***Index Terms*—auto-tuner, sparse matrix-vector multiplication, SpMV, GPU, code generator, sparse data structures**

I. Introduction

Sparse Matrix-Vector multiplication (SpMV, $y=Ax$) is one of the most computational kernels in many domains, such as climate simulation [1], computer graphics [2], molecular dynamics [3], [4], data analytic [5], [6], machine/deep learning [7], [8], to name a few. In the past decades, many efforts have been conducted to improve SpMV performance through proposing sparse matrix formats, leveraging various performance optimization methods, and automatic performance tuning (auto-tuning).

Dozens of sparse matrix formats have been proposed to efficiently compress sparse matrices in memory on contemporary architectures: multi-core CPUs [9], Graphics Processing Units (GPUs) [10], Intel Xeon Phi accelerators [11], and Field-Programmable Gate Array (FPGAs) [12]. These formats are designed for diverse goals: reducing memory access, improving load balance, reducing GPU thread divergence, etc. They store only non-zero elements and ignore zeros which take a major portion of a sparse matrix. (Refer to the work [13] and [14] for good summaries of them). We categorize sparse matrix formats into three groups: Root Formats, Derived Formats, and Hybrid Formats.

Four formats are generally considered as basic formats [14]–[16], or *Root Formats*, which consists of `COOrdinate (COO)`, `Compressed Sparse Row (CSR)`, `ELLPACK (ELL)`, and `DIAgonial (DIA)`. To handle more irregular matrices, better memory compression, or higher performance of sparse kernels, plenty of *Derived Formats* have been proposed. We refer to a derived format as a format manually designed based on only ONE root format, such as `Blocked COORdinate (BCOO)` [17], derived from `COO`; `CSR5` [18], derived from `CSR`; `Sliced ELLPACK (SELL)`, derived from `ELL`, to name a few. Beyond root and derived formats, *Hybrid Formats* flexibly use multiple formats for different portions of a sparse matrix. It could be a mix among root and derived, such as `HYBrid (HYB)`, `COCKTAIL` [19], and `Compressed Sparse eXtended (CSX)` [20].

Because of the diversity of sparsity patterns and close association between input matrix features, architecture characteristics, and SpMV performance, it is unrealistic to find a one-fits-all format or optimization method. Thus, SpMV auto-tuners such as SMAT [15], clSpMV [19], Zhao et al. [21], have been designed to select the most appropriate format for a given matrix from a set of candidate artificial formats.

Despite the efforts of all researches mentioned above, this classic but stubborn kernel is still largely behind its attainable performance from analysis, especially for highly irregular sparse data [22]. We observe three problems in state-of-the-art researches preventing SpMV from achieving higher performance.

Problem 1: Limited human practices meet an ever-growing number of sparse matrices. Generally, a matrix format could handle only a specific matrix pattern and perform this type of matrix well. Thus, the patterns not covered in this format lead to low performance. According to our experiments in SuiteSparse Matrix Collection [23], there is an approximate 10× maximum-minimum performance gap observed from mainstream formats `ELL`, `HYB`, `ACSR` [24] and `CSR-Adaptive` [22]. From another point of view, SuiteSparse Matrix Collection has gradually collected 2893 matrices from 91 domains. As domains and data emerge from real-world problems, most probably, we will face unseen sparse data and patterns in the future, which would need new formats and kernel implementations. It is not effective, practical, or even possible for researchers to keep designing new formats

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

for any incoming matrices.

Problem 2: Challenge of irregular sparsity. Irregularity is almost the biggest challenge in nowadays SpMV program design. It brings a great diversity of distributions for row lengths[1] and row positions, which causes enormous difficulties for efficient parallelism and memory access [22]. In this paper, we define sparse matrices where the variances of its row lengths are more than 100 as irregular matrices, according to target matrices of recent format studies [25], [26]. Irregular matrices occupy more than 35% of SuiteSparse Matrix Collection. General sparse matrix formats cannot accommodate irregular sparsity well due to highly redundant computing, unbalanced load, memory access hot-spots, etc. Though some new formats [18], [24], [27] have been proposed, particularly for irregular sparsity, they still have limited applicability (they only focus on 10-20 matrices in their evaluation).

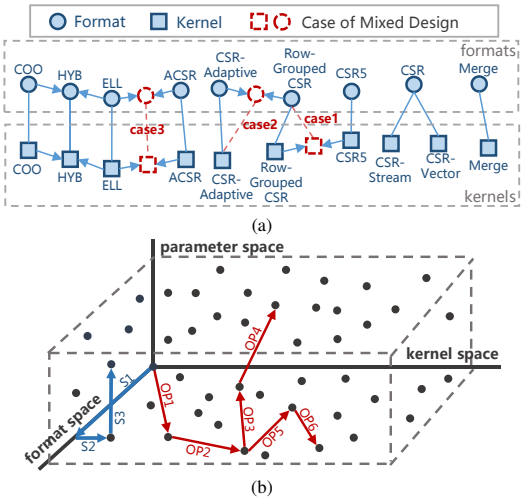

Fig. 1. (a) Search space of a traditional auto-tuner. (b) Searching methodologies of format-selection auto-tuner and AlphaSparse in origin design space.

Problem 3: Limitation of existing auto-tuners. Existing auto-tuners are modeled as coarse-grained format selectors [15], [19] and are limited by human experience and implementations. We depict a small set of artificial formats and SpMV implementations (blue circles and squares, respectively) in Figure 1a. Each format has one or more coupled kernel implementations. Traditional auto-tuners essentially choose a format-kernel combination (represented as solid blue lines), where other potentially existing but humanly undiscovered formats, kernel implementations, and the connections in-between (shown in cases 1-3 in red) have been overlooked. Take case 1 as an example, it uses the existing `row-grouped CSR` format [28] but with a new implementation by combining thread-level reduction of `CSR5` [18] and global memory reduction of `row-grouped CSR`. These omitted cases cause an auto-tuner to miss opportunities for potential performance improvement. To make things worse, complexity from irregular sparsity amplifies this shortcoming of format selectors.

AlphaSparse solves these problems by targeting one ultimate goal: **creating machine-designed SpMV programs that surpass the scope of human practices and outperform both artificial formats and traditional auto-tuners.** We achieve this goal by directly searching in the original design space of the SpMV program, which contains three dimensions: 1) format, the data layout in memory; 2) kernel, the way that data is calculated; 3) parameter, the quantitative details of the first two dimensions (illustrated in Figure 1b). Every position of the design space represents an SpMV program. The blue path shows the selection strategy of traditional auto-tuners that can only take steps in parallel with any of the three directions. In contrast, AlphaSparse proposes a new model, named *Operator Graph*, which simulates the SpMV design philosophy to exploit much larger space. An Operator Graph is a "path" to a specific location of the design space by connecting arbitrary numbers of operators (detailed in Section IV). An operator, a vector in design space, represents a design strategy of the SpMV program and can simultaneously "move" in three dimensions. This more flexible and integrated model enables AlphaSparse to reach designs inaccessible to existing human works and gain more opportunities for higher performance.

TABLE I
COMPARISON OF ALPHASPARSE TO STATE-OF-THE-ART WORKS.

Work		Sparsity	Irregularity	Creativity[2]
Artificial Format Designs	CSR, ELL, COO,etc.	✓	✗	✗
	CSR5 [18], Merge [27], ACSR [24],etc.	✓	✓	✗
Traditional Auto-tuners	SMAT [15], clSpMV [19], Zhao et al. [21]	✓	✗[3]	✗
Compiler Technologies	TVM [29]	✗	✗	✗
	TACO [30]	✓	✗	✗
Intelligent Auto-tuner	AlphaSparse	✓	✓	✓

Table I compares AlphaSparse with mainstream related works from angles of sparsity, irregularity, and creativity. Compared with artificial format designs and traditional SpMV auto-tuners, AlphaSparse shows its novelty in creativity and irregularity. It is the first work that creates completely novel machine-designed formats along with their SpMV implementations to pursue high performance. Some compilers seem to be more flexible, especially TACO [30]. However, its general IR (intermediate representation) hides details of algorithms and hardware architectures, which covers only basic optimizations for general sparse problems and misses many optimization opportunities. Besides, TACO still explores limited artificial formats by leveraging the "level formats" concept for each dimension [31], [32], same as format selectors.

However, three challenges need to be conquered to build the intelligent AlphaSparse. The first one is a much larger search space. Let $\mathcal{A}$ be the number of all known artificial formats and

[1]For sparse matrix, row length is the number of non-zeros in a row.

[2]The ability to create new machine-designed SpMV formats and kernels.

[3]Zhao et al. partly solves the irregularity by including `CSR5` format.

assume each of them provides a unique format or kernel design strategy. By only comparing the format-kernel subset of search space, its size of traditional auto-tuning is $O(\mathcal{A})$, while $O(\mathcal{A}^p)$ theoretically in AlphaSparse with an Operator Graph including p Operators. The second is integrated modeling. Extracting design strategies of SpMV from a large number of existing works and expressing them in a unified IR is non-trivial. The last challenge is projecting positions in the origin design space to three dimensions to obtain corresponding SpMV programs.

AlphaSparse has three main components to solve these challenges: Designer, Format & Kernel Generator, and Search Engine, to accomplish design space's expression, projection, and exploration. Designer and Format & Kernel Generator accept Operator Graphs as input and generate formats with corresponding kernel implementations. Search Engine aims at finding an Operator Graph with high performance. While searching, SpMV performance corresponding to Operator Graph can be obtained by directly running the generated SpMV program. We implement AlphaSparse in more than 110,000 lines of C++ codes that will be released. Although we only focus on SpMV in this paper, the methodology of AlphaSparse can even adapt to more sparse problems by defining new corresponding operators and backends.

Our main contributions are summarized as follows:

- We first show potential high-performance SpMV programs overlooked in existing works and the necessity and feasibility foundations for AlphaSparse (Section II).
- We develop AlphaSparse, which is easy to use by taking Matrix Market files as input and outputting high-performance SpMV codes generated by the machine. AlphaSparse can be considered as a counterpart of *AlphaFold* [33], which predicts the protein structure from the beginning, in high-performance sparse problems; while traditional auto-tuners correspond to traditional template-based methods in protein structure prediction. (Section III)
- The design space is expressed by a newly proposed graph-based modeling, called Operator Graph (Section IV); projected by format and kernel generators to generate compressed data representation and high-performance implementation (Section V); and explored by a three-level search and pruning strategies (Section VI) in AlphaSparse.
- We evaluate AlphaSparse on 843 large matrices from SuiteSparse Matrix Collection. AlphaSparse largely improves SpMV performance by up to 22.2× (3.2× on average) compared to five human-designed state-of-the-art formats. We also compare AlphaSparse with an up-to-date implementation of format selector, where AlphaSparse achieves up to 2.7× (1.5 × on average) performance improvement. (Section VII)

II. Motivation

The motivation of AlphaSparse comes from two observations, which separately show its necessity and feasibility.

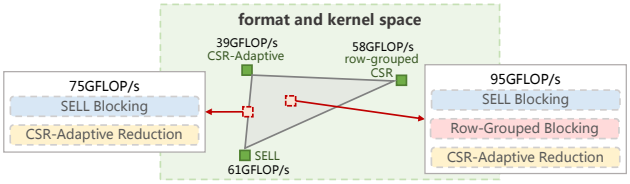

Fig. 2. Mixed designs found by AlphaSparse on the matrix 2D_27628_bjtcai in the space of format and kernel.

Observation 1: Artificial formats and their sparse kernel algorithms are limited by human experience and narrow search space, which misses the potential for higher performance. Newly proposed artificial sparse matrix formats and auto-tuners have covered increasing sparse patterns. However, human practice ignores a large number of potential formats and kernels. As shown in Figure 2, on matrix 2D_27628_bjtcai from SuiteSparse Matrix Collection, `CSR-Adaptive` [34], `row-grouped CSR` [35], `SELL` [36] separately achieves 39 GFLOPS, 58 GFLOPS and 61 GFLOPS. By combining the blocking strategy of `row-grouped CSR` with the reduction strategy of `CSR-Adaptive`, the performance of the mixed format is higher as 75 GFLOPS. Similarly, by mixing formats and kernels from all these source formats, the performance could be even higher as 95 GFLOPS.

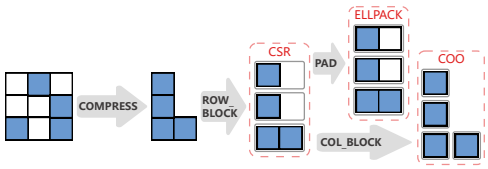

Fig. 3. The steps of converting a tiny sparse matrix to `CSR`, `COO` and `ELL`. Blue blocks are non-zeros, while blank ones are zeros.

Observation 2: Sparse formats are converted from the source matrix with common steps, making creating new formats feasible from more combinations of these common steps. This observation has been proved by other work [37], although they underscored the conversion among existing artificial formats. Usually, when a new artificial format is designed, the conversion routine will also be provided from the original matrix. We take the conversion of three root formats as examples, shown in Figure 3. In the beginning, the original input matrix is compressed by ignoring all zeros. By blocking the matrix in each row, `CSR` format can be obtained. Furthermore, by further padding in each block or by blocking in each column, `ELL` or `COO` can be generated. These four steps commonly exist in other format conversions [35], [38], [39]. Thus, it is feasible to generate or even create a format automatically by taking more common conversion steps.

III. Overview

AlphaSparse proposes an integrated model named Operator Graph. Operator Graph describes and explores the origin three-dimensional design space of format, kernel and parameter simultaneously with operators (shown in Figure 1). It provides a meticulous search to handle the complexity brought by sparsity patterns that are highly associated with SpMV performance. **An operator uniformly expresses the information of kernel**

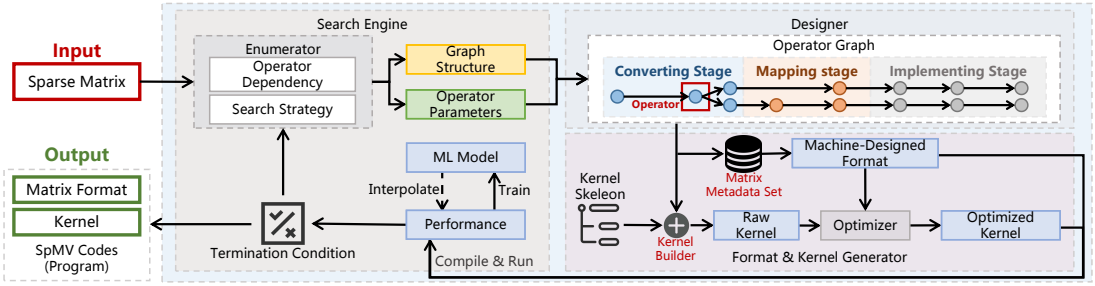

Fig. 4. Overview of AlphaSparse.

and format design, including the configurations of their parameters. Through transforming Operator Graphs, not only high-performance but also new machine-designed formats and kernels could be generated.

AlphaSparse consists of a *Search Engine* (Section VI), a *Designer* (Section IV), and a *Format & Kernel Generator* (Section V). As shown in Figure 4, the Search Engine first enumerates Operator Graphs by generating graph structures and corresponding parameters for their operators under a given search strategy. The enumerated Operator Graph will be sent to the Designer. The Designer executes these operators in order to modify the Matrix Metadata Set, which includes all details of the matrix state (detailed in Section V-A). At last, Format & Kernel Generator produces the kernel and format according to the Operator Graph and Matrix Metadata Set, with several optimizations (detailed in Section V). For a specific structure of Operator Graph, AlphaSparse first gets its performances by directly running the SpMV program of each parameter combination on a coarse-grained grid. To further achieve a detailed search in parameter space with low overhead, AlphaSparse uses a lightweight machine learning (ML) cost model to interpolate parameters to a fine-grained grid. Till the termination condition based on simulated annealing is satisfied, the search process stops and outputs the best SpMV codes found by it.

AlphaSparse has already provided high out-of-the-box performance and is easy to use for top-level users. Users only need to input a Matrix Market file of a sparse matrix, and AlphaSparse will output a matrix stored in a specific format and a kernel implementation. Essentially, apart from traditional auto-tuners, AlphaSparse is moving forward a significant step by acting as a substitute for algorithm researchers in developing new SpMV formats and kernels. Usually, this kind of algorithm work not only highly depends on individual inspiration but also costs time of either months or years. AlphaSparse only takes hours to greatly outperform almost all artificial designs. From this aspect, AlphaSparse is not a traditional online performance tuner but a tool for the SpMV algorithm research or an extremely optimized library generator, which narrows the focus from the entire algorithm to a particular operator(s). The generated codes can be directly called in real-world applications. The artifact description of this paper shows its usage.

IV. Designer

The *Designer* maintains the *Operator Graph*, the key data structure of AlphaSparse. We are the first to break existing formats and kernel implementations [13], [14] into finer-grained design strategies and use them to model the SpMV program (shown in Table II). As the combination of operators, Operator Graph opens a wider integrated space of format and kernel designs. Compared to existing format selectors, AlphaSparse possesses higher flexibility for performance tuning, thus obtaining outperforming SpMV codes in larger probabilities.

A. Operator

Given a sparse matrix, we summarize that its SpMV program is generally developed in three steps: 1) defining a compressed memory layout (i.e., format) of the matrix; 2) mapping (distributing) it to hardware units of different parallelism levels; 3) designing kernel implementation, mainly SpMV reduction strategies. These stages are *converting*, *mapping* and *implementing*. Each stage consists of multiple design or optimization strategies, called *operators*[4]. Defining operators is non-trivial and challenging, which needs plenty of preparatory work to abstract optimizing strategies from existing works and validate their effectiveness in the final performance. For prototyping purposes, AlphaSparse currently only considers operators for GPUs. We list all the operators in AlphaSparse in Table II. Almost all of them are derived from existing research, as shown in the "Source" column. At the level of the overall design of the SpMV program, AlphaSparse has covered the whole design process by the three stages of Operator Graph. At the level of design strategies (so-called operators), it is not easy to get their quantitative and theoretical coverage. As far as we know, AlphaSparse has covered almost all popular formats with high performances.

Operators in the converting stage define compressed memory layout. ROW(COL)_DIV divides the whole matrix into striped sub-matrices in a row or column direction, which branches in the Operator Graph. Each sub-matrix can be treated separately in the following designs (shown on the upper right of Figure 4) that help handle highly irregular matrices. SORT, SORT_SUB, and BIN reorder matrix rows according to their lengths. COMPRESS ignores all zeros of a sparse matrix for storage.

[4]Operators in AlphaSparse represent designs of format and kernel implementation, different from mathematical operators.

TABLE II
OPERATORS CONSIDERED IN ALPHASPARSE.

Stage	Operator	Source	Description
Converting	ROW(COL)_DIV	[40], [41]	Divide a matrix in rows/columns
	SORT	[36], [42]	Sort rows in decreasing order of #non-zeros per row
	SORT_SUB	[36], [42], [43]	Sort rows in decreasing order of #non-zeros per row with in a submatrix
	BIN	[24], [44]	Put rows into different bins according to #non-zeros per row
	COMPRESS	[45]	Ignore all zeros of the sparse matrix
Mapping	BMTB(BMW,BMT)_ROW(COL)_BLOCK	[39], [43], [46], [47]	Split a matrix in row/column dimension, each of which mapped to a thread block/warp/thread
	BMT_NNZ_BLOCK	[18], [25], [41]	Map continuous non-zeros to threads
	BMTB(BMW,BMT)_PAD	[35], [46], [47]	Zero padding to BMTB/BMW/BMT
	SORT_BMTB	[39]	Sorting rows in decreasing order of #non-zeros per row within a BMTB
Implementing	SET_RESOURCES	/	Set runtime configurations
	GMEM_ATOM_RED	[35]	Atomically add intermediate results to global memory
	SHMEM_OFFSET_RED	[22], [27], [34]	Reduce intermediate results from multiple rows to shared memory, according to row offset
	SHMEM_TOTAL_RED	[22], [24]	Reduce intermediate results of the same row in shared memory
	WARP_TOTAL_RED	[48], [49]	Reduce all the intermediate results per warp to one row
	WARP_BITMAP_RED	[47]	Reduce all the intermediate results per warp by bitmap
	WARP_SEG_RED	[18]	Reduce all the intermediate results per warp by segment sum
	THREAD_TOTAL_RED	[24], [47], [50]	Reduce all the intermediate results per thread to one row
	THREAD_BITMAP_RED	[18], [25]	Reduce intermediate results per thread by bitmap

BMTB/BMW/BMT is abbreviation of "a block mapped to a thread block" or "warp" or "thread".

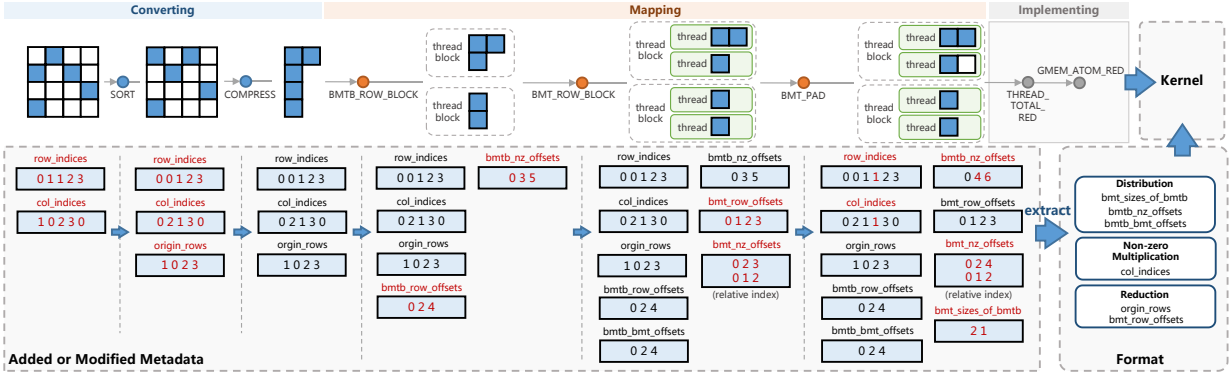

Fig. 5. An Example of format generation. The upper is an Operator Graph; the lower is a subset of Matrix Metadata Set.

The mapping stage always begins after the COMPRESS operator. Operators suffixed by _BLOCK cut adjacent non-zeros of the matrix into blocks and map them to different levels of parallelism. The other operators in this stage further trim the memory layout inside of blocks. Operators suffixed by _PAD add zeros to specific positions of a matrix to get more regular indices for higher performance. SORT_BMTB reorders rows of each BMTB, which can reduce the range of sorting and create opportunities to decrease the padding rate..

Operators in the implementing stage are more relevant to kernel implementation. Except for SET_RESOURCES, all the operators are suffixed by _RED, which are different reduction strategies for intermediate results of BMTB, BMW, or BMT in an SpMV kernel. GMEM_ATOM_RED directly and atomically adds intermediate results to vector y in global memory. Operators prefixed by SHMEM_ are strategies for thread-block-level reduction in shared memory. SHMEM_TOTAL_RED fits for the condition where all intermediate results in a BMBT come from the same row. It adds up all intermediate results of a thread block to a result. SHMEM_OFFSET_RED includes CSR-like row offset indices [51] that record the position of the first intermediate result of each row in BMBTs. It reduces the intermediate results of each row in parallel. Three operators prefixed by WARP_ represent three mainstream strategies of warp-level reduction. WARP_TOTAL_RED is a classic strategy from CSR-Stream [22]. For irregular matrices containing both short and long rows, WARP_BITMAP_RED and WARP_SEG_RED use *bitmap* [47] and *segment sum* [52] to reduce results of BMW by rows. To gain more optimization opportunities from low-level details of the hardware, operators utilize a series of unique features of the GPU. In warp-level operators, hardware-level *Warp Shuffle Functions* [53] are used to achieve high performance of reduction. Operators prefixed by THREAD_ are thread-level reductions in registers. THREAD_TOTOAL_RED is similar to other operators suffixed by _TOTAL_RED. THREAD_BITMAP_RED serially reduces the results of each row, using a bitmap to mark row boundaries.

There is still a huge search space behind an operator that contains parameters of its details (parameter space in Figure 1b), such as sorting granularity, the parallelism of reduction algorithms, blocking size, etc. Some design strategies derived from formats such as `HYB`, `CSB` [54] are also critical to SpMV performance but have not been supported by AlphaSparse. AlphaSparse allows users to implement operators by themselves.

B. Operator Graph

An Operator Graph is generated by connecting operators in order. The upper part of Figure 5 shows an elementary example. A real high-performance Operator Graph could be much deeper and sometimes include branches. This example mainly combines design philosophies of `SELL-P` [38] and `CSR-Scalar`. COMPRESS, BMTB_ROW_BLOCK, BMT_ROW_BLOCK, BMT_PAD, THREAD_TOTAL_RED, GMEM_ATOM_RED are from `SELL-P`, while COMPRESS, BMT_ROW_BLOCK,

THREAD_TOTAL_RED, GMEM_ATOM_RED are from `CSR-Scalar`. SORT is from other formats, like JAD [55].

Dependencies exist between operators. They usually come from operators' semantics. Take the Operator Graph in Figure 5 as an example, BMT_ROW_BLOCK and BMT_PAD cannot be followed by BMTB_ROW_BLOCK. Because when a data block has already been mapped to a thread, it cannot be further split and mapped to a thread block as higher-level parallelism in CUDA. Dependencies can also be defined by users for search pruning (detailed in Section VI).

V. Format & Kernel Generator

Given an Operator Graph, we can move to a specific position of SpMV design space. To get the corresponding format and kernel implementation, Format & Kernel Generator projects this position to format, kernel, and parameter space. Unlike traditional source code generators [56], which are based on a static template and only focus on the kernel implementation, Format & Kernel Generator needs to handle flexible combinations of format and kernel by two components: *Matrix Metadata Set* and *Kernel Builder*.

A. Matrix Metadata Set

Matrix Metadata Set includes multi-perspective descriptions of the current matrix state, recording how the matrix is converted (detailed in observation 2 of Section II). It is a huge key-value memory database whose contents are used to generate formats and kernels. Matrix Metadata Set contains basic matrix information (matrix size, number of columns and rows, length of each column and row), basic non-zero information (parent-block index, row index, column index), and information of blocks distributed to different levels of parallelism (block size, first non-zero index, first row index, first sub-block index), reduction information (row index of intermediate result, etc.), and so on. In an Operator Graph, operators convert a matrix by modifying the Matrix Metadata Set in order. After an Operator Graph has been iterated, Matrix Metadata Set will include all effects of operators to the original matrix cumulatively. A simple example of matrix metadata is shown on the lower part of Figure 5. The red text represents where the metadata is added or modified. Take row_indices and col_indices as examples. They are added by the input matrix, recording the row and column indices of non-zeros, and operator BMT_PAD further modifies them by adding an index of a zero element in a specific position (1, 1).

B. Format Construction

All arrays of a format are extracted from Matrix Metadata Set by choosing the metadata needed by the kernel (determined by kernel fragment detained in Section V-C). In the final format shown in Figure 5, bmtb_nz_offsets, bmt_row_offsets, and bmtb_bmt_offsets record the indices of the first non-zero, sub-block, row in each BMT or BMTB. They are generated by operator BMT_ROW_BLOCK and BMTB_ROW_BLOCK, defining how the matrix is distributed to each thread and thread block. bmt_sizes_of_bmtb is non-zero numbers of BMT in each BMTB, generated by BMT_PAD. origin_rows (generated by SORT) and bmt_row_offsets record the original row indices of intermediate results from non-zero multiplications, which are needed by reduction of GMEM_ATOM_RED (in vector *y*).

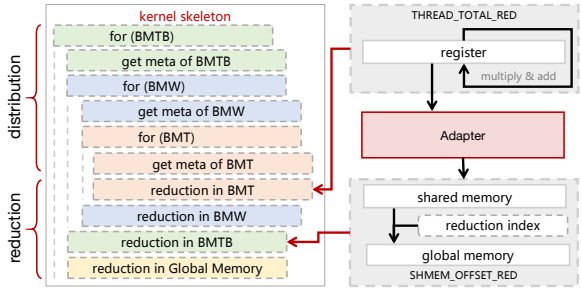

Fig. 6. Example of kernel generation by splicing kernel fragment. The reduction strategies of this case are THREAD_TOTAL_RED and SHMEM_OFFSET_RED.

C. Kernel Builder

The construction process of the SpMV kernel includes two parts: 1) Distribution, mainly determined by the mapping stage. It gets metadata for each block in different levels of parallelism, which mainly includes information for task distribution and reduction strategy. 2) Reduction, mainly determined by the implementing stage. It multiplies the non-zeros of the matrix with the vector elements and reduces their results by row.

According to the commonality of SpMV programs, the template of Kernel Builder includes two key components: *kernel skeleton* and *kernel fragment*. The left of Figure 6 shows the kernel skeleton, which is the root symbol containing multiple nested loops. Each loop traverses blocks distributed to different levels of parallelism(thread block, warp, thread), including a series of slots for kernel fragments. Kernel fragments marked as "get meta of BMX" read metadata arrays needed by other kernel fragments of the same loop, which constitute the format. It can be easily and automatically generated by analyzing data dependency. For the strategy to reduce current intermediate results, kernel fragments prefixed by "reduction in" are determined by operators in the implementing stage. Non-orthogonal factors could appear in combinations of different reduction strategies. To solve this issue, special kernel fragments called *Adapter* need to be pre-defined, which only includes several assignment expressions. Shown on the right of Figure 6, intermediate results from thread-level reduction (THREAD_TOTAL_RED) are further reduced in thread-block-level reduction (SHMEM_OFFSET_RED). The former reduction puts its output in the register group. The latter accepts input only in shared memory, which makes these two reduction strategies cannot be connected directly. An Adapter is needed to copy results from registers to shared memory in an accepted layout.

Figure 7 shows an example kernel of Operator Graph shown in Figure 5. In this case, the whole matrix is divided into BMTBs and BMTs in the row direction. Each thread reduces its contents into one result. These results from threads are further reduced in the global memory. Lines 3-6 and 11-12

```
__global__ void spmv(.....){
.....
1  //reverse all BMTBs
2  for (int BMTB_id = 0; BMTB_id < BMTB_num;
                                  BMTB_id = BMTB_id + thread_block_num){
3  int bmtb_nz_offset = bmtb_nz_offsets[BMTB_id];
4  int bmt_begin_in_bmtb = 2 * BMTB_id;
5  int bmt_end_in_bmtb = 2 * (BMTB_id + 1);          get meta of BMTB
6  int bmt_size_of_bmtb = bmt_sizes_of_bmtb[BMTB_id];
7
8  //reverse all BMTs in each BMTB
9  for (int BMT_id = bmt_begin_in_bmtb; BMT_id < bmt_end_in_bmtb;
                                  BMT_id = BMT_id + thread_block_size){
10
11 int bmt_row_offset = BMT_id;
12 int bmt_nz_offset = bmtb_nz_offset + bmt_size_of_bmtb * BMT_id;   get meta of BMT
13
14 float temp_result = 0;                              THREAD_TOTAL_RED
15 //reverse all non-zeros in each BMT
16 for (int nz_id = bmt_nz_offset; nz_id < bmt_nz_offset + bmt_size_of_bmtb; nz_id++){
17 temp_result = temp_result + val_arr[nz_id] * x[col_indices[nz_id]];
18 }
19
20 int real_index = origin_rows[bmt_row_offset];
21 atomicAdd(&(y[real_index]), temp_result);        Adapter of GMEM_ATOM_ADD
22 } } }
```

Fig. 7. Example of generated kernel of Figure 5 after optimizing. Underlined codes are optimized by Model-Driven Format Compression.

get format(metadata) arrays of BMTB and BMT. Lines 14-18 multiply non-zeros of each BMT by elements of vector x and reduce them in one register represented by temp_result. Lines 20-21 further reduce the intermediate result of each thread by atomic addition in the global memory.

D. Optimizer

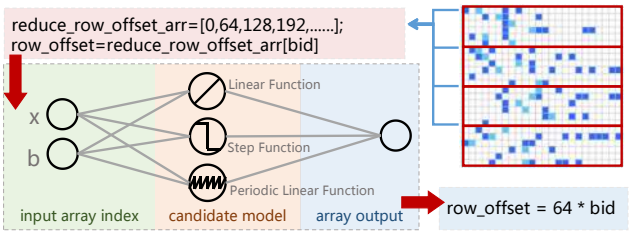

Fig. 8. Example of memory access optimization for a format array.

To improve kernel performance, Kernel Builder supports a series of optimizing strategies, such as removing unnecessary codes, combining multiple short-data-type arrays, etc. Since kernel optimizations have been well studied before, we leverage several state-of-the-art techniques in our tuning system. One prominent optimization is *Model-Driven Format Compression* (derived from [57]), which is especially efficient to memory-access optimization. It reduces the number of memory accesses by transforming array type data (in memory) to models and replacing memory access with calculation.

As shown in Figure 8, an operator named BMTB_ROW_DIV divides the matrix into row bands every 64 lines, and each row band is mapped to a thread block. It adds an array, named reduce_row_offsets, to the format, which includes the first-row index of BMTBs for thread-block-level reduction. row_offset can be calculated directly from the index of BMTB (row_offset=64*bid), by fitting array index and value to a linear model, which makes global memory access (row_offset=reduce_row_offsets[bid]) unnecessary. In manually written codes, programmers can naturally discover the regularity of data structures and directly write the optimized implementation. Because AlphaSparse is entirely automatic, we need to perform this optimization explicitly to achieve competitive performance with human-written codes. In addition to linear functions, other functions, such as step function and periodic linear function, are also supported. Users can also extend the hypothesis function. Unlike normal regression problems of data analysis, any errors in the model would cause incorrect SpMV implementation. To improve the success rate of this optimization, a small number of errors can be tolerated by adding *if* statements to separately assign values for the specific array index that the model cannot fit.

Figure 7 includes example optimizations: Accesses of bmtb_bmt_offsets, bmt_row_offsets are eliminated by Model-Driven Compression; and the optimizer also eliminates the warp-level loop for a cleaner code structure.

VI. Search Engine

Search Engine drives AlphaSparse by enumerating Operator Graphs and choosing the best one of them. To deal with a huge search space (as a challenge detailed in Section I) consisting of the parameters and structure of the Operator graph, Search Engine provides multi-level search from coarse to fine. It exploits the experience of coarse-grained search to accelerate the fine-grained search by an ML model.

A. Operator Graph Search

The search strategy of the Search Engine includes three steps (levels). In the first step, graph structures are enumerated by randomly choosing empty operators and connecting them at the end of the existing Operator Graph. The second step searches node (operator) parameters in a coarse-grained grid and gets the performance of Operator Graphs by directly running corresponding SpMV programs. In the third step, the test results from step two are further interpolated to a fine-grained parameter grid by an ML model. We do not directly do the fine-grained search because the overhead of running SpMV programs is extremely high, even occupying almost all the searching overhead. In comparison, the overhead of the ML model is negligible. To further and reasonably reduce the executions of SpMV programs, the first two steps could be terminated early by simulated annealing. Moreover, we also limit the search time to no more than 8 hours, as a mandatory termination condition, according to our experience.

According to our practice, XGBoost [58] performs very well in interpolation, which is also confirmed to be practical by TVM [29]. It achieves a mean absolute deviation of 5%, which is even less than the performance volatility of GPU. Because of the memory hierarchy of the architectures, we speculate that the cost model of memory-bound programs includes linear decision boundaries, which fits a tree-type model. The third step significantly reduces the overhead of the parameter search. Assume there is an Operator Graph with q parameters. Reducing the search step size by half would increase search space by 2^q times, finally increasing the search time from several hours to several weeks. XGBoost can achieve the same effect by incurring relatively negligible overhead.

B. Pruning

Although AlphaSparse provides a three-level search to accelerate the searching process, the overhead from the first two steps is still expensive because of the huge search space of AlphaSparse. So, in addition to coarse-grained parameter search and simulated annealing, more pruning strategies are needed.

Pruning the search of the parameter. Parameters indicate quantifiable details of an operator. The biggest challenge is the array type parameter. For example, ROW_DIV includes an array type parameter containing the positions where the matrix is divided in the row direction. Assuming the input matrix has 10^5 rows, the search space size of just this single parameter is $10^5!$, which is impossible to grasp. One or more parameter discretization strategies are included in each operator to handle array type parameters. *Parameter discretization strategies* can reduce the parameter space, especially spaces of array-type parameters. In this case, a parameter discretization strategy named DIV_IN_ROW_LEN_MUTATION can be used to divide the matrix where row length mutates. It converts the array type parameter to just several integer parameters describing the degree of such mutation, which can be easily enumerated.

Pruning the search of the graph structure. Pruning strategies for graph structure are added when we find operators are unnecessary for specific matrix sparsity patterns. For example, matrices with short rows do not need to try operators for long row reduction. Users can add their pruning strategies. AlphaSparse provides a ban list for pruned operators, according to already existing operators of graph and sparsity patterns of input matrices.

VII. Evaluation

Our evaluation shows that AlphaSparse provides the highest overall performance among the most advanced artificial formats and the up-to-date implementation of traditional auto-tuning.

A. Experimental Setup

Platform. The experiments are conducted on NVIDIA A100 and RTX 2080. The former is based on Ampere architecture, with 6912 CUDA cores, 40GB HBM2 memory (1.5TB/s), and 19.49 TFLOPS peak performance. The latter is based on Turing architecture, with 2944 CUDA cores, 8GB GDDR6 memory (448GB/s), 10.07 TFLOPS peak performance. We use single-precision for floating-point values in experiments.

Testset. The experiment includes 843 matrices (most of them are irregular) from SuiteSparse Matrix Collection [23] whose features satisfy the three conditions: 1) row number is larger than 9K, 2) number of non-zeros is between 50K and 60M, 3) no empty rows [5]. We ignore matrices with extremely large sizes because they are difficult to grasp. Small matrices are also ignored because they are not suitable for GPU.

[5]Our prototype has not included operators to handle empty rows.

B. Baselines

The baselines are classified into three kinds according to the degree of coupling with SpMV. **Artificial format** represents the special library achieved by hand. **Format selector** represents the traditional auto-tuning framework. **Tensor algebra compiler** represents the more general compiler that considers SpMV as one of many objects.

Artificial format. To compare with artificial formats, we choose several popular state-of-the-art formats with high performance and irregularity-specific design as follows: 1) `ACSR` [24], implemented by us because so far we have not found its high-quality implementation. 2) `CSR-Adaptive` [22], from ViennaCL 1.7.1 [59], [60]. 3) `CSR5` [18][6], 4) `Merge-based CSR(Merge)` [27][7]. 5) `HYB`, from cuSPARSE 9.2.

Format selector. It is unrealistic to fairly compare AlphaSparse with the traditional auto-tuning philosophy based on format selection. The most state-of-the-art auto-tuners, Zhao et al. [21], SMAT(ER) [61] and clSpMV, have historical limitations: 1) They contain only out-of-date formats, which sometimes cannot handle irregularity and cannot take advantage of new GPU features. 2) They have not been actively maintained for a long time. [8] For a reasonable comparison, we implement a Perfect Format Selector (PFS) as a representative to the up-to-date auto-tuner as the baseline.

As a performance-first auto-tuner, PFS does not rely on probabilistic models for format selection. To achieve the highest accuracy(100%), PFS can certainly select the best formats by directly running SpMV of all candidate formats. For an up-to-date implementation, PFS consists of five aforementioned state-of-the-art formats: `ACSR`, `CSR-Adaptive`, `CSR5`, `Merge`, `HYB` [62]; three root formats from the widely-used cuSPARSE library: `ELL` (from v9.2), `COO`, `CSR` (from v11.6); and two derived formats: `SELL`, `row-grouped CSR`, for a comprehensive comparison.

Tensor algebra compiler. Compiler focuses more on code-level optimizations instead of algorithm-level designs. For a more sufficient comparison, we add TACO [30][9] as a baseline of the tensor algebra compiler.

C. Performance Comparison with Artificial Formats

Figure 9a shows the overall performance of AlphaSparse and state-of-the-art formats in 843 matrices. The x-axes are matrix size and we use floating point operations per second(GFLOPS) to represent performance as the y-axes. AlphaSparse achieves the highest performance among all artificial formats. On A100, AlphaSparse achieves an average 3.2× speedup and the maximum 22.2× speedup (in matrix

[6]https://github.com/weifengliu-ssslab/Benchmark_SpMV_using_CSR5

[7]https://github.com/dumerrill/merge-spmv

[8]clSpMV has not been maintained for seven years, and we could not deploy it on our platform due to the error appearing. We believe AlphaSparse can outperform clSpMV because AlphaSparse gains better performance than `ACSR` which outperforms clSpMV.

[9]https://github.com/tensor-compiler/tacowe use CUDA code fully automatically generated by it.

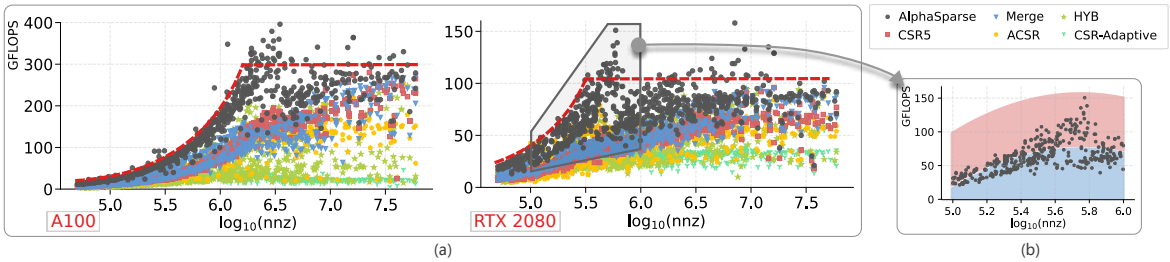

Fig. 9. SpMV overall performance of matrices with different sizes. (a) All test results on RTX 2080 and A100, while the red dashed lines show a trend of achieved highest performances. (b) Parts of the AlphaSparse results on RTX 2080. The region colored by red includes cases providing higher performance in specific matrix size, while the other is colored by blue.

TSOPF_RS_b300_c2) over all artificial formats. In particular, it achieves average 2.3×, 5.7×, 2.0×, 2.0× and 3.9× speedup over ACSR, CSR-Adaptive, CSR5, Merge and HYB, respectively. AlphaSparse outperforms Merge, ACSR, CSR-Adaptive and CSR5 in all 843 matrices, while it outperforms HYB in 841 matrices (because AlphaSparse has not included the matrix decomposition strategy of HYB). On RTX 2080, AlphaSparse achieves an average 2.0× speedup and the maximum 8.3× speedup (in matrix TSOPF_RS_b2052_c1). In particular, it achieves average 2.0×, 2.3×, 2.0×, 1.7× and 2.4× speedup over ACSR, CSR-Adaptive, CSR5, Merge and HYB, respectively.

Merge and CSR5 provide the highest overall performance among all artificial formats, because they benefit from thread-level load balance by allocating a balanced number of non-zeros or rows to each thread. The overall performance of CSR-Adaptive is the lowest. It performs well in relatively small matrices by achieving higher parallelism. However, it suffers from giving up the reduction in registers, making it perform the worst on remaining matrices. ACSR and HYB are based on matrix decomposition, providing mild performance.

In Figure 9a, maximum performances of AlphaSparse in each matrix size make up a trend of flat-tail shape, represented by red dashed lines. As a memory-bound program, the SpMV performance can be raised by the increasing occupy of memory bandwidth when the matrix size is not too large. When the memory bandwidth is sufficiently used, the performance will not further increase [63]. In our evaluation, only AlphaSparse approaches this trend.

To show how input matrices affect performances, we take samples of RTX 2080 test results and divide them into two parts in Figure 9b. We choose this range of results because it shows clear upper and lower borders and makes us easy to split in the middle of them. Although these two parts of cases correspond to the same matrix sizes, the performance of the upper part (red) is up to 5.0× (average 1.4×) higher than the lower. According to our further observation, we suspect that the two matrix features cause this performance gap. One is average row length($\frac{nnz}{n}$), which in the upper part is 1.9× higher than the lower. We speculate that a higher average row length improves performance by increasing the ratio of calculation to memory access and decreasing the proportion of reduction operations (which require synchronization) in the SpMV program. The other is row variance(degree of regularity, $\frac{\sum(row\_len-\frac{nnz}{n})^2}{n}$), which in the upper part is 20× lower than the lower. Lower regularity can usually achieve higher reduction performance, better load balance, and less computation waste.

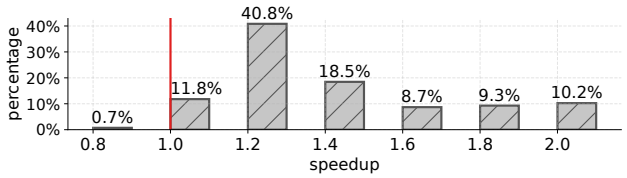

Fig. 10. The frequency distribution of AlphaSparse's speedup over PFS on A100.

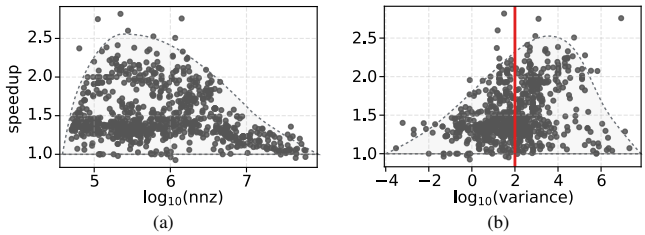

Fig. 11. Speedups of AlphaSparse over PFS corresponding to (a) matrix sizes and (b) variances of row lengths on A100.

D. Performance Comparison with Format Selector

Figure 10 illustrates the frequency distribution of AlphaSparse's speedup over PFS on A100. In 99.3% cases, the performance of AlphaSparse is higher. In the remaining 0.7% matrices, AlphaSparse performs worse because some design strategies of formats in PFS are not included in AlphaSparse (detailed in Section VII-H). Most (40.8%) cases achieve the speedup between 1.2× and 1.4×.

Figure 11 further demonstrates speedups of AlphaSparse over PFS along with matrix sizes and row variances(to show the influences of the irregular sparsity). Figure 11a shows impressive speedups can be achieved in cases where the matrix fits into the 40 MB L2 cache of the A100, and large matrices ($\geq 10^7$ non-zeros) provide lower speedups. In Figure 11b, the red line shows the boundary of the regularity and irregularity (10^2 as mentioned). The peak of speedup is 2.7×, appearing in the middle degree of matrix size and irregularity, which shows the fine-grained trade-off provided by Operator Graph is suitable for moderate sparsity patterns. On the contrary, designs of most artificial formats are based on human observations of specific extreme sparsity patterns from matrices such as Webbase, mip1, FullChip, as shown

in their papers. They ignore matrices with moderate sparsity patterns. Moreover, we find irregular matrices benefit more from AlphaSparse: the average speedup is 1.4× for regular sparsity, while for irregular sparsity, the average speedup goes up to 1.6×.

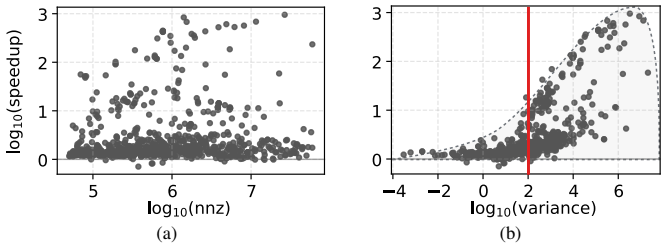

Fig. 12. Speedups of AlphaSparse over TACO corresponding to (a) matrix sizes and (b) variances of row lengths on A100.

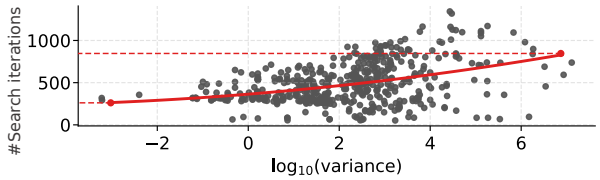

Fig. 13. Numbers of searching iterations along with row variances on A100.

TABLE III
SEARCH TIME AND PERFORMANCES WITH AND WITHOUT PRUNING ON A100

Matrix	Search Time (hour)		Performance (GFLOPS)	
	no pruning	pruning	no pruning	pruning
pdb1HYS		3.1	273.4	303.2
windtunnel_evap3d		2.3	286.1	343.3
consph		1.9	339.8	356.0
Ga41As41H72		3.4	193.6	242.1
Si41Ge41H72		4.8	175.1	236.9
ASIC_680k		0.9	121.8	169.6
mip1	8.0	4.9	227.0	226.0
Rucci1		1.9	218.9	223.7
boyd2		2.4	61.3	80.2
rajat31		5.1	189.2	226.0
transient		3.0	127.7	153.0
ins2		3.4	101.9	152.8
bone010		3.3	193.2	235.4
Average	8.0	3.2	198.6	231.0

E. Performance Comparison with TACO

AlphaSparse greatly outperforms TACO. On A100, AlphaSparse achieves an 18.1× average speedup and the maximum 950.8× speedup over TACO. As shown in Figure 12a, speedups are insensitive to matrix sizes, unlike PFS. Figure 12b shows the peak of speedup appearing in highly irregular matrices. Two reasons cause its relatively lower performance. The first reason is that TACO is not tailored for SpMV. Its three key features, index compression, loop optimization, and automatic parallelism, only target general sparse problems. None of them can handle problems brought by SpMV, especially the irregularity. The second reason is that TACO lacks the utilization of GPU features, which even lacks competitiveness with human-designed programs.

F. Searching Overhead

Since the first two search steps occupy almost all the searching overhead (as mentioned in Section VI), we use the number of iterations in the first two steps to represent the performance of search strategies. Figure 13 shows search iterations along with degrees of matrix irregularity (so-called row variances). The regression line of test results illustrates a positive correlation between the search overhead and matrix irregularity: regular matrices need 3.5× fewer iterations than highly irregular matrices. These prove that our pruning rules significantly reduce search overhead by ignoring operators for the irregularity when the input matrix is regular.

Table III shows how pruning strategies affect search time and performances of AlphaSparse. We record test results before[10] and after pruning in 13 popular matrices evaluated from published researches. Pruning strategies reduce search time by 2.5× on average. Because pruning strategies include high-quality human experience, they eliminate unnecessary enumerations and make the Search Engine focus on areas of the design space that are highly likely to find high-performance formats in limited search time. Pruning strategies also improve performance by 1.2× on average. Compared with existing offline auto-tuners, such as PATUS (8 hours) [64], SDSL (≥33 hours) [65], Halide (2 hours-2 days) [66], PARTANS (2.5 hours-32 days) [67], the search time of AlphaSparse is competitive.

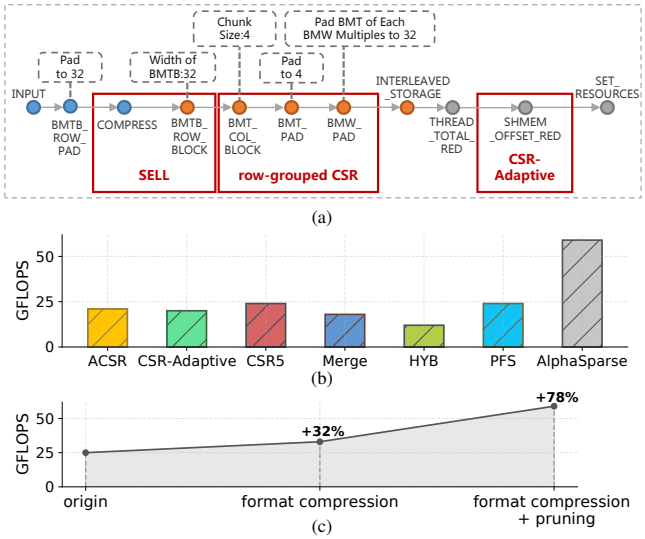

Fig. 14. An example of matrix scfxm1 − 2r on A100. (a) a snapshot of its Operator Graph, (b) the performance comparison with artificial formats and (c) performance improvements achieved by two key optimizations of AlphaSparse.

G. Creative Capability of AlphaSparse

Creating new machine-designed formats is the main driver of performance improvement. From our statistics, in 73.1% of test cases, AlphaSparse outperforms all counterparts by creating machine-designed formats which are not covered by its source formats (as referenced in Table II). In 16.5% of cases providing new formats, the branches appear in Operator Graphs, which means AlphaSparse designs different formats

[10]We remove the simulated annealing and other pre-defined pruning strategies (shown in Section VI-B) in the baseline of no pruning.

and corresponding kernel implementations for different parts of original matrices.

Figure 14a shows an example Operator Graph of an new format generated by AlphaSparse for matrix scfxm1 − 2r. It mainly includes the thread-block-level blocking strategy from SELL, the thread-level blocking strategy from row-grouped CSR, and the reduction strategy from CSR-Adaptive. Finally, as shown in Figure 14b, it achieves 2.7× speedup (which is the highest) over PFS and state-of-the-art artificial formats. Appropriate trade-offs between different design strategies achieve high performance. Compared with the source formats, for this matrix, the machine-designed format avoids high padding rate of SELL, inefficient global memory reduction of row-grouped CSR, ignorance of thread-level reduction in CSR-Adaptive, and benefits from regular row block indices of SELL, low padding rate of row-grouped CSR, efficient shared memory reduction of CSR-Adaptive. In terms of its state-of-the-art counterparts, expensive strategies, such as binning of ACSR and blocking for load balancing of CSR5 and Merge-based CSR, are unnecessary because the matrix is not too irregular. Moreover, HYB includes a large, inefficient COO component in this matrix, which makes it also worse than the machine-designed format. Figure 14c shows a 32% performance improvement is brought by Model-Driven Format Compression, and pruning strategies bring a further 78% performance improvement.

H. Limitation

In AlphaSparse, the lack of operators is the main reason causing slightly lower performance on specific matrices. A representative case is matrix GL7d19. Its best artificial format is HYB, which performs even better than machine-designed formats from AlphaSparse. In this matrix, the lengths of almost all rows are relatively balanced, except for a few rows that are several times longer. The matrix decomposition strategy of HYB is very suitable for this sparsity pattern, but the current AlphaSparse has not included this strategy.

In addition to HYB-like decomposition, two popular categories of operator have not been included: operators for local densities [68]–[70], diagonal patterns [2], [20]. They separate the regular parts of the matrix and handle them exclusively to achieve high performance. However, they only cover a small number of matrices. Our prototype implementation has not considered them, but they will be considered for more complete support in the future.

Currently, the proof-of-concept AlphaSparse only supports CUDA. However, it can be extended to other platforms by implementing new tailored operators. Users only need to define how the operator modifies metadata and occasionally need to define kernel fragments.

VIII. Related Works

Auto-tuners. Auto-tuners have proven to be a successful performance tuning approach, represented by ATLAS [71], FFTW [72], SPIRAL [73], and OSKI [74], for the increasingly diverse and complicated computer architecture designs. For sparse linear algebra, SMAT [15], clSpMV [19], TileSpMV [75] Naser Sedaghati et al. [76] and CSX [20] select the best artificial format and SpMV implementation for the given matrix; while IA-SpGEMM [77] selects the best formats for SpGEMM. TVM [29] and Ansor [78] are auto-tuner for dense tensor calculation by automatically generating code structure and selecting the best corresponding parameters. COGENT [79] provides high performance tensor contractions on GPU. CASpMV [80] include auto-tuner for matrix partition on the Sunway. Some general auto-tuners, ATF [81], OpenTuner [82], CLTune [83], Optuna [84], mNM [85], Muthu et al. [86], Tiwari et al. [87], Rigel [88] and SMAC3 [89], have been designed to ease the designing effort of an auto-tuner and target in a broader scope. AlphaSparse is not limited by selecting among artificial formats, kernel implementations, parameters, and it is able to create SpMV code and break through the limits of human design.

Artificial format and kernel design. To improve the performance of SpMV, a dozens of formats [13], [14] have been proposed. State-of-the-art formats are derived from several base formats. ALIGNED_COO [90], SCOO [91], BRO-COO [92], BCOO [25] are derived from COO. ICSR [48], CSR-Adaptive [34], ACSR [24], CSR5 [18], LightSpMV [49] are derived from CSR. ELL-R [93], AdELL [47], JAD [55] are derived from ELL. HYB [51], HDC [94] and HEC [95] are hybrid formats. These artificial formats are designed by human according to their observations, AlphaSparse can automatically creates formats without human intervene.

Code generation. TVM [29] is a template-based machine code generator for dense tensor calculation. TACO [30] can handle high-order sparse tensor calculation by compressing the index of each dimension. LL [96] is a DSL to define matrix format and its SpMV kernel. AlphaSparse provides a graph-based expression for generating of both format and kernel.

IX. Conclusion and Future Work

We present AlphaSparse, a fully automatic SpMV code designer, that generates outperforming format and kernel directly from an input sparse matrix. It unifies the modeling of format and kernel implementation and achieves an impressive speedup of up to 22.2× over state-of-the-art human-designed formats on the NVIDIA GPU. We will examine advanced search strategies from existing research [29], [37], [97], and add efficient format conversion routines in the future.

Acknowledgements

This project is supported by National Natural Science Foundation of China under Grant No. T2125013, 62032023, 61972377, 61702483, and international partnership program of Chinese Academy of Sciences, Grant No. 171111KYSB20180011, and China National Postdoctoral Program for Innovative Talents BX2021320.

REFERENCES

[1] M. Tillenius, E. Larsson, E. Lehto, and N. Flyer, "A task parallel implementation of a scattered node stencil-based solver for the shallow water equations," in *Proc. 6th Swedish Workshop on Multi-Core Computing, Halmstad University, Halmstad, Sweden*, 2013, pp. 33–36.

[2] D. Weber, J. Bender, M. Schnoes, A. Stork, and D. Fellner, "Efficient gpu data structures and methods to solve sparse linear systems in dynamics applications," in *Computer graphics forum*, vol. 32, no. 1. Wiley Online Library, 2013, pp. 16–26.

[3] M. Zheng, X. Li, and L. Guo, "Algorithms of gpu-enabled reactive force field (reaxff) molecular dynamics," *Journal of Molecular Graphics and Modelling*, vol. 41, pp. 1–11, 2013.

[4] S. B. Kylasa, H. M. Aktulga, and A. Y. Grama, "Puremd-gpu: A reactive molecular dynamics simulation package for gpus," *Journal of Computational Physics*, vol. 272, pp. 343–359, 2014.

[5] C. Chen, K. Li, A. Ouyang, Z. Zeng, and K. Li, "Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1275–1288, 2018.

[6] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a {PC}," in *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 31–46.

[7] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai *et al.*, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[8] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[9] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[10] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[11] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu, "Knights landing: Second-generation intel xeon phi product," *Ieee micro*, vol. 36, no. 2, pp. 34–46, 2016.

[12] I. Kuon, R. Tessier, and J. Rose, *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.

[13] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 2, pp. 428–440, 2015.

[14] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on gpgpus," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, pp. 1–49, 2017.

[15] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 117–126.

[16] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 94–105.

[17] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[18] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 339–350.

[19] B.-Y. Su and K. Keutzer, "clspmv: A cross-platform opencl spmv framework on gpus," in *Proceedings of the 26th ACM international conference on Supercomputing*, 2012, pp. 353–364.

[20] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: an extended compression format for spmv on shared memory systems," *ACM SIGPLAN Notices*, no. 8, pp. 247–256, 2011.

[21] 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.

[22] M. Daga and J. L. Greathouse, "Structural agnostic spmv: Adapting csr-adaptive for irregular matrices," in *2015 IEEE 22nd International conference on high performance computing (HiPC)*. IEEE, 2015, pp. 64–74.

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

[24] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 781–792.

[25] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: Yet another spmv framework on gpus," *Acm Sigplan Notices*, vol. 49, no. 8, pp. 107–118, 2014.

[26] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently running spmv on long vector architectures," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 292–303. [Online]. Available: https://doi.org/10.1145/3437801.3441592

[27] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 678–689.

[28] T. Oberhuber, A. Suzuki, and J. Vacata, "New row-grouped csr format for storing the sparse matrices on gpu with implementation in cuda," *arXiv preprint arXiv:1012.2270*, 2010.

[29] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[30] 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.

[31] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 123:1–123:30, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276493

[32] S. Chou, "Unified sparse formats for tensor algebra compilers," Cambridge, MA, Feb 2018. [Online]. Available: http://groups.csail.mit.edu/commit/papers/2018/chou-18-sm-thesis.pdf

[33] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[34] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 769–780.

[35] M. Heller and T. Oberhuber, "Adaptive row-grouped csr format for storing of sparse matrices on gpu," *arXiv preprint arXiv:1203.5737*, 2012.

[36] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A. R. Bishop, "Sparse matrix-vector multiplication on gpgpu clusters: A new storage format and a scalable implementation," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 2012, pp. 1696–1702.

[37] S. Chou, F. Kjolstad, and S. Amarasinghe, "Automatic generation of efficient sparse tensor format conversion routines," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 823–838.

[38] H. Anzt, S. Tomov, and J. Dongarra, "Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus," *University of Tennessee, Tech. Rep. ut-eecs-14-727*, 2014.

[39] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[40] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in

Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ser. ICS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 273–282. [Online]. Available: https://doi.org/10.1145/2464996.2465013

[41] Y. Liang, W. T. Tang, R. Zhao, M. Lu, H. P. Huynh, and R. S. M. Goh, "Scale-free sparse matrix-vector multiplication on many-core architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 12, pp. 2106–2119, 2017.

[42] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using cuda based on a hybrid sparse matrix format," in *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, vol. 11. IEEE, 2010, pp. V11–161.

[43] C. Zheng, S. Gu, T.-X. Gu, B. Yang, and X.-P. Liu, "Biell: A bisection ellpack-based storage format for optimizing spmv on gpus," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2639–2647, 2014.

[44] K. Hou, W.-c. Feng, and S. Che, "Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi-and many-core processors," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 713–722.

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

[46] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on gpus," in *Proceedings of the 28th ACM international conference on Supercomputing*, 2014, pp. 273–282.

[47] M. Maggioni and T. Berger-Wolf, "Adell: An adaptive warp-balancing ell format for efficient sparse matrix-vector multiplication on gpus," in *2013 42nd international conference on parallel processing*. IEEE, 2013, pp. 11–20.

[48] M. Yang, C. Sun, Z. Li, and D. Cao, "An improved sparse matrix-vector multiplication kernel for solving modified equation in large scale power flow calculation on cuda," in *Proceedings of The 7th International Power Electronics and Motion Control Conference*, vol. 3. IEEE, 2012, pp. 2028–2031.

[49] Y. Liu and B. Schmidt, "Lightspmv: Faster csr-based sparse matrix-vector multiplication on cuda-enabled gpus," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2015, pp. 82–89.

[50] X. Feng, H. Jin, R. Zheng, K. Hu, J. Zeng, and Z. Shao, "Optimization of sparse matrix-vector multiplication with variant csr on gpus," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*. IEEE, 2011, pp. 165–172.

[51] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Citeseer, Tech. Rep., 2008.

[52] G. E. Blelloch, M. A. Heroux, and M. Zagha, "Segmented operations for sparse matrix computation on vector multiprocessors," CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, Tech. Rep., 1993.

[53] Nvidia, "Cuda toolkit v11.6.1 programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions Accessed March 12, 2022.

[54] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, 2009, pp. 233–244.

[55] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[56] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: end-to-end optimization stack for deep learning," *arXiv preprint arXiv:1802.04799*, vol. 11, p. 20, 2018.

[57] T. Augustine, J. Sarma, L.-N. Pouchet, and G. Rodríguez, "Generating piecewise-regular code from irregular structures," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 625–639.

[58] 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.

[59] K. Rupp, F. Rudolf, and J. Weinbub, "Viennacl-a high level linear algebra library for gpus and multi-core cpus," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[60] K. Rupp, "viennacl," [EB/OL], http://viennacl.sourceforge.net/viennacl-about.html Accessed March 26, 2022.

[61] G. Tan, J. Liu, and J. Li, "Design and implementation of adaptive spmv library for multicore and many-core architecture," *ACM Trans. Math. Softw.*, vol. 44, no. 4, aug 2018. [Online]. Available: https://doi.org/10.1145/3218823

[62] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[63] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[64] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 676–687.

[65] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 13–24.

[66] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[67] T. Lutz, C. Fensch, and M. Cole, "Partans: An autotuning framework for stencil computation on multi-gpu systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–24, 2013.

[68] D. Langr, I. Šimeček, P. Tvrdík, T. Dytrych, and J. P. Draayer, "Adaptive-blocking hierarchical storage format for sparse matrices," in *2012 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2012, pp. 545–551.

[69] 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, 2010.

[70] V. Karakasis, G. Goumas, and N. Koziris, "A comparative study of blocking storage methods for sparse matrices on multicore architectures," in *2009 International Conference on Computational Science and Engineering*, vol. 1. IEEE, 2009, pp. 247–256.

[71] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[72] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[73] M. Puschel, J. M. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko *et al.*, "Spiral: Code generation for dsp transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

[74] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 071.

[75] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 68–78.

[76] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 99–108.

[77] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 94–105. [Online]. Available: https://doi.org/10.1145/3330345.3330354

[78] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, *Ansor: Generating High-Performance Tensor Programs for Deep Learning*, 2020.

[79] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "A code generator for high-performance tensor contractions on gpus," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 85–95.

[80] G. Xiao, K. Li, Y. Chen, W. He, A. Y. Zomaya, and T. Li, "Caspmv: a customized and accelerative spmv framework for the sunway taihulight," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 131–146, 2019.

[81] A. Rasch and S. Gorlatch, "Atf: A generic directive-based auto-tuning framework," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 5, p. e4423, 2019.

[82] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[83] C. Nugteren and V. Codreanu, "Cltune: A generic auto-tuner for opencl kernels," in *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*. IEEE, 2015, pp. 195–202.

[84] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.

[85] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" *Procedia Computer Science*, vol. 4, pp. 2136–2145, 2011.

[86] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 225–234.

[87] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[88] V. Sreenivasan, R. Javali, M. Hall, P. Balaprakash, T. R. Scogland, and B. R. de Supinski, "A framework for enabling openmp autotuning," in *International Workshop on OpenMP*. Springer, 2019, pp. 50–60.

[89] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International conference on learning and intelligent optimization*. Springer, 2011, pp. 507–523.

[90] M. Shah and V. Patel, "An efficient sparse matrix multiplication for skewed matrix on gpu," in *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, 2012, pp. 1301–1306.

[91] H.-V. Dang and B. Schmidt, "Cuda-enabled sparse matrix–vector multiplication on gpus using atomic operations," *Parallel Computing*, vol. 39, no. 11, pp. 737–750, 2013.

[92] W. T. Tang, W. J. Tan, R. S. M. Goh, S. J. Turner, and W.-F. Wong, "A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the gpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 9, pp. 2373–2385, 2014.

[93] F. Vázquez, J.-J. Fernández, and E. M. Garzón, "A new approach for sparse matrix vector product on nvidia gpus," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 8, pp. 815–826, 2011.

[94] W. Yang, K. Li, Y. Liu, L. Shi, and L. Wan, "Optimization of quasi-diagonal matrix–vector multiplication on gpu," *The international journal of high performance computing applications*, vol. 28, no. 2, pp. 183–195, 2014.

[95] H. Liu, S. Yu, Z. Chen, B. Hsieh, and L. Shao, "Sparse matrix-vector multiplication on nvidia gpu," *International Journal of Numerical Analysis & Modeling, Series B*, vol. 3, no. 2, pp. 185–191, 2012.

[96] G. Arnold, J. Hölzl, A. S. Köksal, R. Bodík, and M. Sagiv, "Specifying and verifying sparse matrix codes," *SIGPLAN Not.*, vol. 45, no. 9, p. 249–260, sep 2010. [Online]. Available: https://doi.org/10.1145/1932681.1863581

[97] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

AlphaSparse achieves up to 22.2 times (3.2 times on average) speedup over state-of-the-art formats and 2.8 times (1.5 times on average) over the up-to-date implementation of the traditional auto-tuning philosophy. Users need to input a Matrix Market file of a sparse matrix, and AlphaSparse will output the Operator Graph and kernel implementation. The experiments are conducted on NVIDIA A100 and RTX 2080. We use single-precision for floating-point values in experiments.

DOWNLOAD DATA OF TEST MATRICES

In our evaluation, all input matrices are from SuiteSparse Matrix Collection (https://sparse.tamu.edu). Two ways are provided to download data as follows.

Directly download from websites. Go to the websites of SuiteSparse Matrix Collection, click the link of specific matrix name, and click the download link named "Matrix Market". The downloaded file is zipped. By extracting the file, a ".mtx" file can be gotten.

Use the python interface. Install the *ssgetpy* Python module. Run "import ssgetpy" and type "help(ssgetpy)" to get a detailed help message on using ssgetpy to search and download sparse matrices. We have provided a python script named "get_data_set_from_UF.py" to download all the needed matrices. The variable "UF_DIR" is the destination of downloaded data.

REPRODUCE ALPHASPARSE

AlphaSparse is implemented by C++11 codes. The automatically output SpMV kernel is implemented by CUDA code. The steps to reproduce AlphaSparse is following:

(1) Clone the GitHub repository. And go to the root directory of AlphaSparse.
(2) Set the location of nvcc used by AlphaSparse in "cuda_code/make_template.sh". AlphaSparse will use this shell script to compile the kernel generated by it.
(3) Set the configuration of AlphaSparse. Fields named "ROOT_PATH_STR" and "spmv_header_file" in the file named "global_config.json" needs set according to the path of AlphaSparse.
(4) Create a directory for temporary data: "mkdir data_source"
(5) Compile. "make -j 16"
(6) Download Matrix Market File from SuiteSparse Matrix Collection (https://sparse.tamu.edu).
(7) Unpack the file. And use "data_prepare.py" to preprocess it: "python3 data_prepare.py {directory_matrix_market_file}/{matrix_name}.mtx {directory_matrix_market_file}/{matrix_name}.mtx.coo".
(8) Design SpMV program for specific matrix: "./main directory_matrix_market_file/matrix_name.mtx.coo > data_source/test.log"
(9) After several hours. The description of Operator Graph and its performance (GFLOPS) are shown in data_source/test.log. And corresponding SpMV kernel is shown in "cuda_code/template.cu".

We have also provided a Python script named "batch_test_spmv_builder.py" for batch tests. A more detailed tutorial is shown in https://github.com/AnonymousRepo123/AlphaSparse.

SYSTEM ENVIRONMENT

Except for the GPU, the environments of RTX 2080 and A100 platforms are the same. Our evaluation environment follows.

Linux: 5.4.0-99-generic Ubuntu 20.04 focal
CPU: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
Memory: 64GB
Device: NVIDIA TU104 [GeForce RTX 2080]
Driver Version: 495.29.05
CUDA Version: 11.5
Python: 3.8
GCC: 9.4

AUTHOR-CREATED OR MODIFIED ARTIFACTS:

Artifact 1

Persistent ID: `https://github.com/AnonymousRepo123/AlphaSparse`
Artifact name: AlphaSparse

Reproduction of the artifact without container: It is very easy to deploy AlphaSparse. No third-party library is needed to be installed by users. So it is not necessary to use a container.