A Pattern-Based SpGEMM Library for Multi-Core and Many-Core Architectures

Zhen Xie, Guangming Tan, *Member, IEEE*,
Weifeng Liu, *Senior Member, IEEE*, and Ninghui Sun, *Member, IEEE*

Abstract—General sparse matrix-matrix multiplication (SpGEMM) is one of the most important mathematical library routines in a number of applications. In recent years, several efficient SpGEMM algorithms have been proposed, however, most of them are based on the compressed sparse row (CSR) format, and the possible performance gain from exploiting other formats has not been well studied. And some specific algorithms are restricted to parameter tuning that has a significant impact on performance. So the particular format, algorithm, and parameter that yield the best performance for SpGEMM remain undetermined. In this article, we conduct a prospective study on format-specific parallel SpGEMM algorithms and analyze their pros and cons. We then propose a pattern-based SpGEMM library, that provides a unified programming interface in the CSR format, analyses the pattern of two input matrices, and automatically determines the best format, algorithm, and parameter for arbitrary matrix pairs. For this purpose, we build an algorithm set that integrates three new designed algorithms with existing popular libraries, and design a hybrid deep learning model called MatNet to quickly identify patterns of input matrices and accurately predict the best solution by using sparse features and density representations. The evaluation shows that this library consistently outperforms the state-of-the-art library. We also demonstrate its adaptability in an AMG solver and a BFS algorithm with 30 percent performance improvement.

Index Terms—SpGEMM, spare BLAS, sparse format, auto-tuning, neural network

1 Introduction

Sparse matrix-matrix multiplication (SpGEMM) is an essential sparse kernel in a number of applications. For example, it often accounts for more than half of the cost of the setup phase for restricting and interpolating matrices in algebraic multigrid methods (AMG) [2]. Many graph processing operations, such as breadth-first search [3], Markov clustering [4], graph contraction [3], subgraph extraction [5], peer pressure clustering [6], and cycle detection [7], can be expressed as SpGEMM. GraphBLAS [8] also defines matrix-based graph algorithms. Efficient SpGEMM algorithms are thus crucial for these applications to achieve higher performance.

Currently, more than a dozen SpGEMM libraries have been proposed and widely used, such as the Intel MKL [9], vector-based sparse accumulator (SPA) [10], hash-based method [11], heap-based method [12], cuSPARSE [13], bhSPARSE [14], InCSR_SpGEMM [15], Fastspmm [16], and CUSP [17] and NSPARSE [18] proposed by NVIDIA. We define these libraries as *libraries* in the following expression.

- *Zhen Xie, Guangming Tan, and Ninghui Sun are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100864, China, and also with the University of Chinese Academy of Sciences, Beijing 100864, China. E-mail: xiezhen@ncic.ac.cn, {tgm, snh}@ict.ac.cn.*
- *Weifeng Liu is with the Department of Computer Science and Technology, China University of Petroleum, Beijing 102249, China. E-mail: weifeng.liu@cup.edu.cn.*

Manuscript received 12 Apr. 2020; revised 27 Apr. 2021; accepted 1 June 2021. Date of publication 17 June 2021; date of current version 8 July 2021.
(Corresponding author: Zhen Xie.)
Recommended for acceptance by C. Ding.
Digital Object Identifier no. 10.1109/TPDS.2021.3090328

However, these libraries are sensitive to sparse input matrices and thus exhibit significant fluctuations in performance. In Fig. 1a, we compare the performance of selected four libraries by calculating $A \times A^T$ on an Intel CPU (as in Section 5.1). X-axis represents the matrix, and Y-axis represents the performance of SpGEMM. It is clear that different algorithms deliver their best performance on different matrices, and no single algorithm dominates on all data sets in terms of performance. Such performance variance is also reflected on a large set of inputs (Section 3.3). The fundamental reason accounting for the uncertain performance of the existing solution is the difference of matrix pattern of inputs, which determine the behavior of computation and memory access. This problem transfers the burden of identifying the optimal library onto application programmers and poses special challenges for the automatic library selector.

On the contrary, Fig. 1b shows two sources of overhead: the proportion of time spent on sparse accumulation and memory access for the SPA SpGEMM algorithm. It is clear that memory access takes up a significant amount of execution time. However, research in the area has largely ignored the potential for improving performance by optimizing memory access and has preferred instead to continue to develop new sparse accumulation algorithms [5], [19] for the compute part. To some extent, SpGEMM is similar to sparse matrix-vector multiplication (SpMV) and sparse triangular solve (SpTRSV) for irregular and indirect memory access patterns [20]. Much of the research on SpMV and SpTRSV has been dedicated to optimizing memory access by excavating classic storage formats [21], [22], [23], [24], [25] with promising results [26], [27]. Back to SpGEMM, such classic storage formats, as DIA, COO, and ELL, can reduce memory requirements or accelerate memory access

1045-9219 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

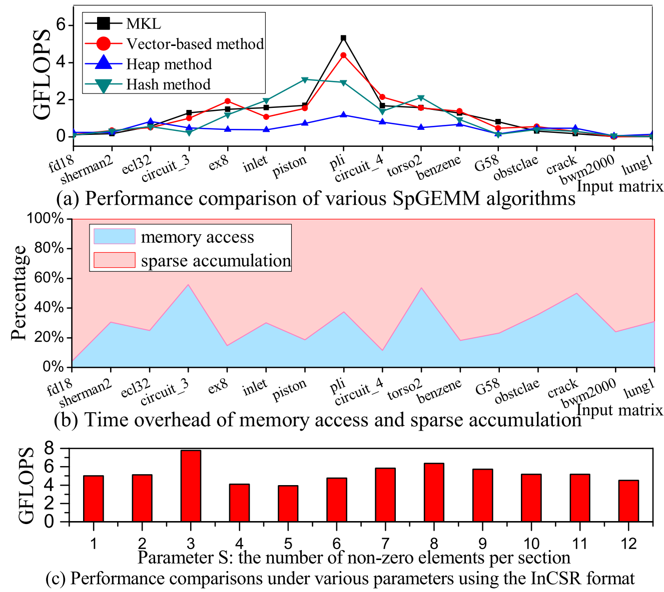

Fig. 1. Comparison of performance of different algorithms and their overhead.

on vector architectures and change the order of the calculation process, or can even reduce the number of sparse accumulation operations. We define these storage formats as *formats* in the following expression. The other motivation of this work is to explore the influence of several classic storage formats on SpGEMM.

The current SpGEMM libraries mainly rely on the compressed sparse row (CSR) format and some variants. For example, Indexed CSR (InCSR) [15] divides each row of the sparse matrix into multiple sections of S elements, which could reduce the size of each multiplication and thus increase data locality. Fig. 1c shows the performance of $A \times A^T$ for "Bar" matrix when the parameter S changes from 1 to 12. It is obvious that the performance of matrix multiplication varies with the change of parameter S. The performance corresponding to the optimal parameter ($S = 3$) is 29.3 percent higher than that of the default parameter ($S = 8$). In addition, another tunable parameter in block CSR (BCSR) format for bmSPARSE algorithm [28] divides the entire matrix into multiple blocks and can also affect the performance greatly on GPU. Such parameters and the mutual influence increase the complexity of manual tuning and became a challenge.

In this paper, we design multiple SpGEMM algorithms based on a variety of widely used sparse storage formats and analyze the conditions leading to better performance. Then we motivate this work by conducting a comprehensive experiment to verify the performance improvement of the diversification of the SpGEMM algorithm. Therefore, in order to integrate three new algorithms with the existing SpGEMM libraries and choose the optimal implementation (algorithm and parameter), we propose a pattern-based auto-tuning SpGEMM library, which classifies two input matrices into the most appropriate category among the assembled SpGEMM algorithm set and chooses the optimal performance parameters by employing a novel deep learning-based tuner. We call the precision with which the model can select the correct format with the best performance as accuracy. To train this tuner, we thus build a large number of matrix multiplication pairs by using all matrices from the current version of SuiteSparse Matrix Collection and collect the performance data of the SpGEMM algorithm set as the output of the training data on a variety of architectures.

Moreover, in order to excavate the pattern of the input matrix and make more accurate predictions, we explore some matrix features as input of the training data, such as the sparse features and density representation. And based on the comparison of the effectiveness of different features, we construct a hybrid neural network called MatNet, which combines the Multilayer Perceptron Neural Network (MLP) and Convolutional Neural Network (CNN) together. The output of MatNet consists of two aspects: the most appropriate algorithm and the optimal parameter if needed, so we use multitask-learning (MTL) to train this model. The model topology allows information sharing when predicting two dependent variables: the optimal algorithm and parameter. Compared with traditional machine learning and single-task neural network models, the hybrid neural network MatNet improves the model accuracy. We train these models on three platforms and found that MatNet is the most suitable for solving this problem and can be easily migrated to other architectures with nearly equivalent prediction accuracy.

In addition, as a lightweight SpGEMM library, this library provides a unified interface in the CSR format to quickly predict the best implementation for two input matrices, and the matrices are finally executed with possible format conversion. We evaluate the library on three processors (an Intel CPU, an AMD CPU, and an Nvidia GPU), and show that it achieves significantly better performance that is on average 3.89x and 21.14x faster than the Intel MKL on dual Intel Xeon E5-2620 and dual AMD EPYC 7501, respectively, with an accuracy of 93 percent, and 2.57x faster than the NVIDIA cuSPARSE library on Tesla P100 with an accuracy of 91 percent. We also evaluate the adaptability of this library in an Algebraic Multigrid (AMG) solver of sparse solver Hypre [29] and a breadth-first search (BFS) of graph analysis [30], and the results show about 30 percent performance improvement.

The main contributions of this paper are as follows:

- We propose multiple SpGEMM algorithms based on a variety of widely used sparse storage formats, and redesign the sparse accumulation and memory access methods that represent two main overheads in the SpGEMM. We also analyze the advantages and disadvantages of various format-specific algorithms. By comparing it with current libraries by running all matrices from the SuitSparse Matrix Collection, significant performance gaps naturally lead to the adoption of an auto-tuning model.
- We compare several existing auto-tuners based on different abstractions of matrix patterns and merge two highly advantageous patterns (sparse features and density representation) as training input.
- We propose a hybrid neural network called MatNet to select the best format, parameter, and algorithm from a large algorithm set. In order to predict the most appropriate implementation, we leverage

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

A = 1 2 0 0 / 0 3 4 0 / 0 0 5 6 / 0 0 0 7	
ptr = [0 2 4 6 7] rows = [0 0 1 1 2 2 3] cols = [0 1 1 2 2 3 3] data = [1 2 3 4 5 6 7] **COO**	ptr = [0 2 4 6 7] col_ind = [0 1 1 2 2 3 3] data = [1 2 3 4 5 6 7] **CSR**
pos = [-1 -1 -1 0 1 -1 -1] **DIA** offsets = [0 1] data = [1 3 5 7 2 4 6 *]	*nnz = [2 2 2 1]* **ELL** col_ind = [0 1 1 2 2 3 3 *] data = [1 2 3 4 5 6 7 *]
rows = [0 0 1 1 2 2 3] cols = [0 1 1 2 2 3 3] data = [1 2 3 4 5 6 7] counter-vector = [2, 2, 0, 2, 1, 1, 2, 0, 2, 2, 0, 1] **InCRS**	
col_ind = [0 1 1 2 2 3 3 *] data = [1 2 3 4 5 6 7 *] rl = [2 2 2 1] **ELLPACK-R**	
rows = [0 0 1] cols = [0 1 1] data = [1 0 3 2 0 4 0 0 5 0 7 6] **BCSR**	

Fig. 2. An example of the seven sparse matrix formats, where the italics represent small changes. The COO format adds a row offset array, the DIA format adds a diagonal position array, and the ELL format adds an array for counting NNZs per row. As an example, the parameter S and b in InCRS format are 4 and 2, the block size in BCSR format are 2×2. All formats are sorted in row order.

multi-task learning to train this neural network. In benchmarking more than 8,000 matrix multiplication pairs, the predictive accuracy of MatNet is over 93 percent.

- We develop a pattern-based SpGEMM library with a general interface based on the CSR format. Users can thus transparently obtain the best performance. We implement our library on three processors and yield average speedups of 3.89x, 21.14x, and 2.57x. We also deploy our library to two real-world applications and achieve about 30 percent performance improvement.

The rest of this paper is organized as follows. Section 2 is a brief introduction to multiple sparse storage formats and the SpGEMM algorithm. In Section 3, based on DIA, COO, and ELL formats, three SpGEMM algorithms are designed for optimizing sparse accumulation and memory access simultaneously for the first time, and the advantages and disadvantages of various algorithms are analyzed. In Section 4, we propose the overview of pattern-based library and describe several components in detail, including pattern selection and model design. We report and analyze the experiment results in Section 5. Section 6 summarizes the related work and Section 7 concludes this paper.

2 Background

2.1 Sparse Matrix Storage Format

The sparse storage format defines the structure used to store the distributions and values of a sparse matrix, with the goal of balancing the reduction in storage space by storing only non-zero elements and implementing efficient memory access by placing the accessed data into a contiguous memory space. To achieve higher efficiency in sparse routines, at least tens of formats have been developed since the 1970s. In particular, most have been derived from the four classic formats which are described below (refer to [31] for a more detailed illustration). Fig. 2 shows an example of multiple formats on matrix A.

- Coordinate (COO) Format: The coordinate format is the most flexible and simplest format. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly.
- Compressed Sparse Row (CSR) Format: The most popular representation contains three arrays: the beginning position of each row is stored in ”ptr”, and the column indices and values of each non-zero element are stored in ”col_ind” and ”data”, respectively.
- Diagonal (DIA) Format: Values of diagonals are stored as columns in a dense matrix. Another ”offsets” array saves offsets from the main diagonal.
- ELLPACK (ELL) Format: It uses two matrices to pack all non-zeros to the left with the same number of rows. The first ”col_ind” matrix stores the column indices and the second ”data” matrix stores the values.
- Indexed CRS (InCRS) Format: It is a variant of CSR to improve the data locality. It stores information on the number of non-zeros inside the sections and blocks. InCRS divides each row into sections of S elements, which are sub-divided into blocks of b elements.
- ELLPACK-R Format: It is a variant of ELL to further improve the performance on GPUs. ELLPACK-R consists of an additional integer array called rl, which stores the actual length of each row, regardless of the number of the zero elements padded.
- Block Compressed Sparse Row (BCSR) Format: BCSR stores fixed-size blocks contiguously, row by row. The block size is a parameter for performance tuning.

2.2 Parallel SpGEMM Method

Let matrix A have size $m \times n$ and B have size $n \times k$. The matrix product is $C = AB$. The element in the ith row and jth column in matrix C can be expressed as: $c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}$. The parallel SpGEMM method was proposed by Gustavson [32] and improved on MATLAB by Gilbert *et al.* [10]. This algorithm (Algorithm 1) in parallel multiplies rows of A by the entire B matrix to calculate rows of C by summing the product of all non-zero elements as the sparse accumulation operation. Similarly, many GPU SpGEMM algorithms improve the sparse accumulation operation for accumulating partial results by using distributed memory [5], a hash table [13], [18], or the ”expansion, sorting, and compression” (ESC) method [17]. Some of these algorithms are included in our algorithm set.

Algorithm 1. Row-Wise SpGEMM Algorithm for $C = A \times B$. We Use C/C++ Notation, i.e., C[i,j] Refers to the $(i+1)$th Row and the $(j+1)$th Column Element in the Matrix C

1: **#parallelfor**
2: **for** $i = 0$ to $C.row$ **do**
3: **for** $j = A.row_ind[i]$ to $A.row_ind[i+1]$ **do**
4: $//accumulate\ partial\ results\ in\ row$
5: $C[i,j] \leftarrow C[i,j] + A[i,j] * B[j,:]$

3 SpGEMM Algorithm

In this section, three format-special SpGEMM algorithms, as well as their advantages and disadvantages, are introduced

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

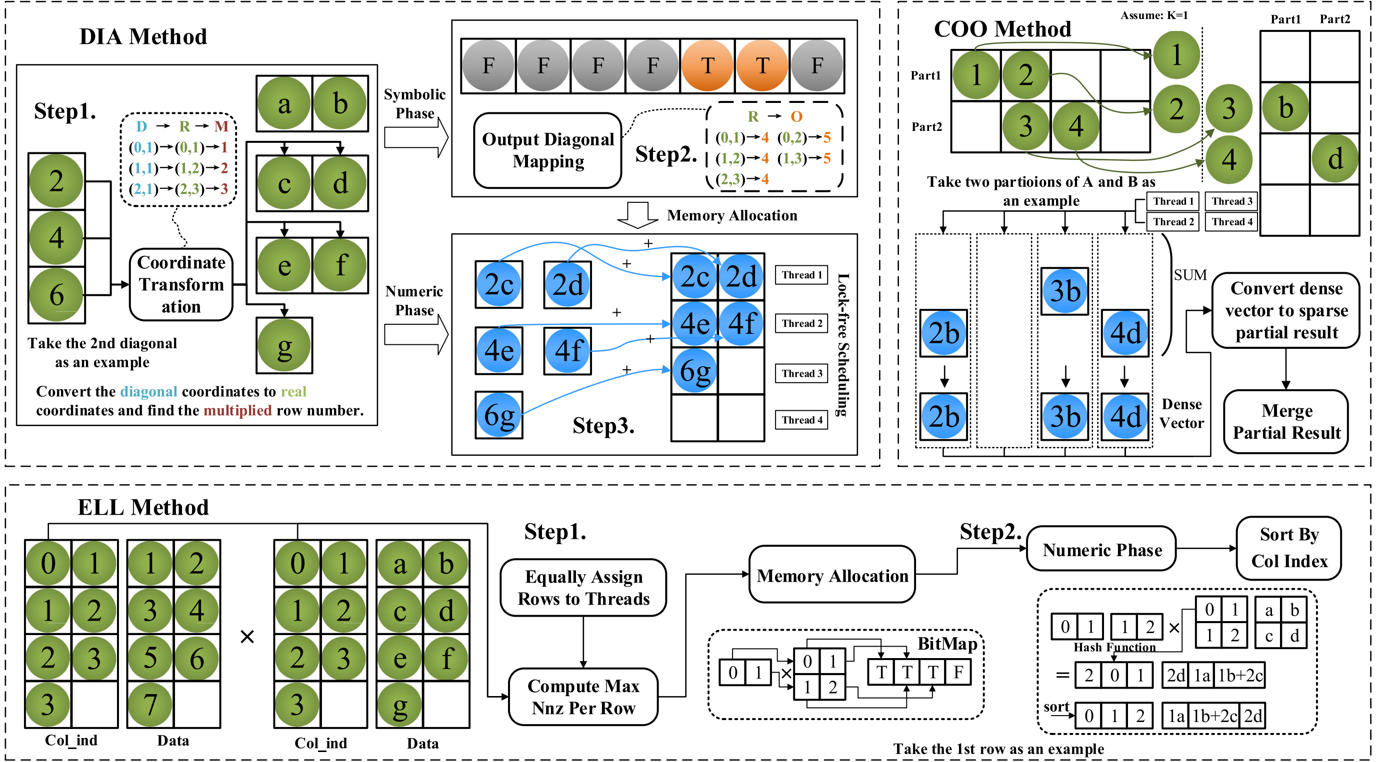

Fig. 3. Flowchart of three format-specific algorithms and some examples of $A \times A'$. The DIA method shows the processes of coordinate transformation and partial accumulation. The COO method divides matrices by k=1 and shows that the length of the dense vector is reduced to that of the previous quarter. The ELL method also uses a line as an example to demonstrate the fast symbol phase by BitMap and hash functions used in the numeric phase.

by using $A \times A'$ as an example.[1] We also analyze and compare nine CPU algorithms and five GPU algorithms by running all matrices from the SuiteSparse Matrix Collection such that the motivation for auto-tuning naturally emerges.

3.1 SpGEMM in DIA, COO, and ELL Formats

Because the DIA format continuously stores diagonal elements, it appears impossible to multiply two diagonals directly. To connect these diagonals, we first append a "pos" array to the original DIA format to record the order of each diagonal line, which can be used to quickly and easily convert diagonal coordinates into real coordinates. As shown in Fig. 3, the multiplication proceeds generally as follows: Step 1: Each element of the diagonal line is first converted into real coordinates to obtain the multiplied row number. For example in Fig. 3, three diagonal coordinates (0,1, 1,1, and 2,1) are converted to row coordinates (0,1, 1,2, and 2,3) and then look for the multiplied columns (1, 2, and 3). We call this process "coordinate transformation". Step 2: The real coordinates of the outputs are mapped to the corresponding diagonal numbers, and the bitmap where the diagonal numbers are located is marked as "T". Step 3: The memory of C is allocated according to the number of "T"s in the bitmap, and the partial results are added to the corresponding positions in the same manner as in the first step.

1. A and A' have the same structure and different values. The values of A range from 1 to 7 and those of A' are from a to g.

Algorithm 2. DIA Method

1: **function** DIA_MUL_DIA(A, B, C)
2: *Malloc Dense BitMap*$[A.row + B.col - 1]$ *and Init by false*
3: **for** $i \in A.row$ **do**
4: **for** $j \in A.num_diagonals$ **do**
5: $A_j \leftarrow i + A.offsets[j]$ *//Convert DIA to REAL*
6: **for** $k \in B.num_diagonals$ **do**
7: $B_j \leftarrow B_i + B.offsets[k]$ *//Convert DIA to REAL*
8: $out_dia \leftarrow A.row - A_i + B_j - 1$ *//Mapping*
9: **if** $BitMask[out_dia]$ $==$ *false* **then**
10: $C.output_dia \leftarrow C.output_dia + 1$
11: $BitMask[out_dia] \leftarrow true$
12: $Malloc_DIA(C.output_dia)$
13: #**parallel for**
14: **for** $i \in A.row$ **do**
15: **for** $j \in A.num_diagonals$ **do**
16: $A_j \leftarrow i + A.offsets[j]$
17: **for** $k \in B.num_diagonals$ **do**
18: $B_j \leftarrow B_i + B.offsets[k]$
19: $out_dia \leftarrow A.row - A_i + B_j - 1$
20: $[out_i, out_j] \leftarrow [A_i, C.pos[out_j - out_i + C.row - 1]]$
21: $C.data[out_i, out_j] \leftarrow C.data[out_i, out_j] + A.data[i, j] \times B.data[B_i, k]$

Compared with the need to store a large amount of row information for CSR-based SpGEMM, Algorithm 2 significantly reduces the overhead due to memory access for the diagonal matrix and directly adds the intermediate results

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

to the target address without extra memory consumption. Note that row-based thread scheduling takes a row as the minimum unit such that it can achieve load balancing and avoid write-write conflicts among threads. This "lock-free scheduling" method can avoid altogether the use of the lock. We call this method the "DIA method."

The COO format separately stores non-zero elements of the same row, because of which the flexible format can more easily be split and merged. Algorithm 3 first divides matrix A into k parts by row and matrix B into k parts by column (k is two or four). Each partition of A and B are successively computed for a part of C by the SPA method [10] and all the partial results are finally merged. As shown in Fig. 3, matrix A is first divided into four row matrices and matrix A' is divided into four column matrices. Taking two partitions as an example, four threads perform the multiplication calculation of each part respectively. Given that the number of columns of matrix A' is divided into a quarter of those of the SPA algorithm, the memory consumption of each thread is a quarter of that of the SPA algorithm. Finally, the partial results between threads are merged into the remelting result matrix. The most significant advantage of this algorithm is that it greatly reduces the length of the dense vector $\frac{B_col}{k}$ times over that of the SPA method and improves the efficiency of the cache, but incurs additional overhead in partitioning and merging the matrices. We call this the "COO method."

Algorithm 3. COO Method

```
1:  function COO_MUL_COO(A, B, C)
2:    Divide A to A_1, ..., A_k by row
3:    Divide B to B_1, ..., B_k by column
4:    for m ∈ k do
5:      for n ∈ k do
6:        Malloc Dense Vector[B_n.col] and Init by −1
7:        #parallel for
8:        for i ∈ A_m.row do
9:          for j ∈ A_m.ptr[i] to A_m.ptr[i+1] do
10: for k ∈ B_n.ptr[A_m.cols[j] : A_m.cols[j]+1] do
11: if mask[B_n.cols[k]] ≠ i then
12: mask[B_n.cols[k]] ← i
13: num_nnz ← num_nnz + 1
14:       Malloc_CSR(C_mn)
15:       #parallel for
16:       for i ∈ A_m.row do
17:         for j ∈ A_m.ptr[i] to A_m.ptr[i+1] do
18: for k ∈ B_n.ptr[A_m.cols[j] : A_m.cols[j]+1] do
19: output ← B_n.cols[k]
20: Sums[output] += A_m.data[j] × B_n.data[output]
21:         Sparsify Sums to C_mn
22:   Merge C_11, ..., C_kk to C
```

The ELL format packages the original matrix into two rectangular matrices of the same size by shifting all non-zero elements to left for more efficient memory access. Because each line of the ELL format contains the same non-zero number, this format makes it possible to reduce the overhead of the symbol phase of the SpGEMM algorithm. In Step 1, we first equally assign matrix rows to threads and use the Col_ind of two matrices to compute the maximum non-zero elements per row of C by a bitmap (an example of the first row of A is given). For example in Fig. 3, one matrix row and two matrix columns can determine the memory consumption of the accumulated matrix rows (three non-zero elements in figure) by BitMap multiplication [28]. In Step 2, the memory of C is allocated by the maximum number of non-zero elements per row, and the newly allocated memory is used as a hash table to store and accumulate the intermediate results. All partial results are mapped to the corresponding positions by calculating the hash values of the column indices or keeping plus one whenever a collision occurs. Finally, the disordered matrix C is sorted. Algorithm 4 has two main advantages: (1) Because the Col_ind is placed in contiguous memory space, the symbolic phase can make full use of the SIMD instructions to speed-up the efficiency of loading and assigning data. (2) In the numeric phase, the memory space pre-allocated to C is used as a hash table, which not only benefits the advantage of the hash table as the sparse accumulator but also avoids memory consumption. We call this the "ELL method."

Algorithm 4. ELL Method

```
1:  function ELL_MUL_ELL(A, B, C)
2:    Malloc Dense BitMap[B.col] and Init by False
3:    #parallel for
4:    for k ∈ A.row do
5:      for i ∈ A.nnz[k] do
6:        for j ∈ B.nnz[A_col[k × B_max + i]] do
7:          if Mask[B.cols[i × B_max + j]] ≠ k then
8:  Mask[B.cols[i × B_max + j]] ← k
9:  nnz_row ← nnz_row + 1
10:       C.nnz_row[i] ← nnz_row
11:   C_max ← MAX(C.nnz_row)
12:   Malloc_ELL(C.row × C.max_nnz_per_row)
13:   #parallel for
14:   for k ∈ C.row do
15:     for i ∈ A.nnz[k] do
16:       for j ∈ B.nnz[A_col[k × A_max + i]] do
17:         Output_hash ← hash(B.cols[A.col[k, i], j])
18:         C.cols[i, Output_hash] ← B.cols[A.col[k, i], j]
19:         C.data[i, Output_hash] += A.data[k, i] × B.data[A.col
      [k, i], j]
20:   Sort C.cols and C.data
```

3.2 Algorithm Set and Static Analysis

Thus far, we have constructed three format-specific algorithms. By integrating them with currently available popular algorithm libraries, as shown in Table 1, nine SpGEMM algorithms are developed for the CPU and six for the GPU.

For SpGEMM algorithms on CPU, except for the three methods we designed, the MKL, SPA vector-based method, hash-based method, heap-based method, and bhSPARSE are designed based on the CSR format on CPU, and the InCSR_SpGEMM is proposed for the InCSR format. Regarding the types of matrices that various algorithms are good at handling, *MKL* is a closed-source library, so its advantages and disadvantages are difficult to analyze.

For the *SPA vector-based method*, it accumulates multiple intermediate results into a dense accumulator. It may not be suitable for high parallelism and large accumulator size due to high memory requirements, but it does not require extra

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Nine Algorithms for CPU and Six Algorithms for GPU

Platform	Algorithm	Parameter
CPU	Intel MKL mkl_sparse_sp2m (CSR) [9]	N
	DIA method (DIA)	N
	COO method (COO)	N
	ELL method (ELL)	N
	SPA vector based method (CSR) [10]	N
	Hash based method (CSR) [11]	N
	Heap based method (CSR) [12]	N
	bhSPARSE (CSR) [14]	N
	InCSR_SpGEMM (InCSR) [15]	**Y**
GPU	CUSP v0.5.1 ESC method (COO) [17]	N
	cuSPARSE v8.0.61 (CSR) [13]	N
	NSPARSE (CSR) [18]	N
	FastSpMM (ELLPACK-R) [16]	N
	bhSPARSE (CSR) [14]	N
	bmSPARSE (BCSR) [28]	**Y**

overhead for hashing, sorting, or merging operations, thus this method is especially suitable for matrices with low sparsity and a small number of columns.

For the *hash-based method*, it allocates a memory space based on the upper bound estimation as the hash table and uses the column indexes of the intermediate results as the key. Hash table is used to reduce the memory consumption of dense accumulator in the SPA vetcore-based method by using hash function. Based on the characteristics of the hash function, this algorithm performs well on these matrices with randomly distributed non-zero elements.

For the *heap-based method*, the most time-consuming part is sort-based merging, which sorts the intermediate results according to the column indexes and then sums the values with the same column indexes. Therefore, the algorithm performs well on the matrix with a large number of rows, mainly because the sorting overhead is evenly amortized.

For the *bhSPARSE* algorithm, it is a hybrid method for result matrix pre-allocation. It divides all rows into multiple groups according to the number of non-zeros and progressively allocates space for the long rows, and therefore suitable for matrices where non-zero elements of each row are not uniform.

For the *InCSR_SpGEMM* algorithm, it accesses memory elements directly and reuses sharing data among a mesh by grouping the non-zero elements of each row, thus leads a huge speedup when the non-zero elements of the second matrix are the block distribution. In particular, the InCSR_SpGEMM algorithm requires two parameters to determine the size of the section (S) and the number of blocks within each section (b). Both two parameters affect the efficiency of the memory access and become the main factors that affect the performance of this algorithm.

For SpGEMM algorithms on GPU, the *CUSP* is based on the COO format and uses the same algorithm as the SPA vector-based method, therefore performs well on these matrices with low sparsity and a small number of columns.

For the *cuSPARSE* and *NSPARSE* algorithm, they are both based on the CSR format and the hash-based sparse accumulator. The difference between these two algorithms is the use of a different hash function. In particular, the NSPARSE improves the BalancedHash [33] and reduces the consumption of shared memory by partitioning the rows of the input and output, respectively. Both of these two algorithms can get decent performance on most matrices.

For the *FastSpMM* and *bmSPARSE* algorithm, the FastSpMM uses ELLPACK-R to enhance performance by storing the sparse matrix in a regular data structure. However, this algorithm may suffer when processing very irregular sparse matrices. The bmSPARSE is based on the BCSR format and converts the original matrix to regular small dense matrices for efficient data locality. Thus, the block size is a decisive parameter for the bmSPARSE algorithm in terms of performance.

Through theoretical analysis, we can see that these algorithms have various advantages and disadvantages, and there are overlaps between the advantages among algorithm sets. For example, the matrix "3dtube" can not only satisfy the distribution of ELLPACK-R format for FastSpMM algorithm but can also be arranged in blocks so that the InCSR_SpGEMM and bmSPARSE have good performance. In addition, the parameters of InCSR_SpGEMM and bmSPARSE algorithm determine the way of non-zero element grouping and the optimal choices vary greatly for different inputs.

To understand the performance potential of this algorithm set and the best parameter settings, we build 8000+ matrix multiplication pairs by using all the matrices in the SuiteSparse Matrix Collection and compare the performance of these algorithms.

3.3 Performance Comparison

We compare the performance of various algorithms on three architectures (as in Section 5.1). To achieve accurate results and complete the task in a controllable time, the run time is the average of 10 trials, and we restrict the memory consumption of the new matrix format to no more than five times the CSR format. Based on this memory restriction, approximately 92.54, 18.37, 57.14, 43.25, 56.55, and 19.62 percent of the matrix pairs will work in COO, DIA, ELL, InCSR, EllPACK-R, and BCSR format. The execution times of all algorithms could be no longer than five times that of the MKL or cuSPARSE, which also means that these matrix pairs are not suitable for a specific format or algorithm. In addition, for the InCSR_SpGEMM and bmSPARSE algorithm, the choice of parameters for the parameter-adjustable algorithm is also an important factor in terms of performance. Therefore, we construct a variety of parameter combinations to cover the entire search space. For the InCSR_SpGEMM algorithm, the parameter S and b can range from 2 to 32, and the combinations are limited to the criterion that S is greater than b. For the bmSPARSE algorithm, we build various block size from 1×1 size to 8×8. Finally, the performance data for various algorithms and parameter combinations are collected.

As shown in Table 2, a general view of the experiments clearly shows significant differences in performance with varying inputs, formats, algorithms, and platforms. In addition, no single format and algorithm can constantly deliver the best performance[2] on all matrix pairs. Each format and

2. The best performance is the best out of the consider options.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Performance Statistics: "Dominance" and "Percentage" Represent the Number and Proportion of the Best and the Better than Baseline for Various Algorithms

	Method	Dominance		Percentage		Average Speedup	Speedup by "Ideal Tool"
		Best of all	Over BL.	Best of all	Over BL.		
Intel CPU	MKL (Baseline)	1329	-	16.21%	-	-	10.52x
	DIA method	474	1107	5.78%	13.51%	72.04x	
	COO method	255	255	3.11%	3.11%	7.63x	
	ELL method	1443	2879	17.62%	35.13%	9.92x	
	SPA vector-based	215	748	2.61%	9.13%	1.31x	
	Hash-based	1381	4307	16.85%	52.56%	6.37x	
	Heap-based	475	1951	5.79%	23.81%	6.21x	
	bhSPARSE	1874	2749	22.87%	33.54%	3.73x	
	InCSR_SpGEMM	749	1353	9.14%	16.51%	9.75x	
AMD CPU	MKL (Baseline)	708	-	8.64%	-	-	52.21x
	DIA method	745	1544	9.09%	18.94%	346.0x	
	COO method	265	586	3.23%	7.19%	8.20x	
	ELL method	1876	3044	22.89%	37.35%	32.96x	
	SPA vector-based	463	2529	5.65%	31.03%	1.58x	
	Hash-based	758	2363	9.24%	28.99%	21.40x	
	Heap-based	779	1015	9.50%	12.45%	12.18x	
	bhSPARSE	1341	2546	16.36%	31.07%	21.40x	
	InCSR_SpGEMM	1260	1315	15.37%	16.05%	12.18x	
NVIDIA GPU	cuSPARSE(Baseline)	1585	-	19.64%	-	-	3.11x
	CUSP	208	769	2.78%	10.26%	6.27X	
	NSPARSE	2657	3525	32.9%	47.04%	3.71X	
	FastSpMM	569	769	4.05%	7.47%	4.56X	
	bhSPARSE	2631	3418	32.60%	42.72%	3.24X	
	bmSPARSE	422	874	5.21%	10.79%	2.85X	

"Average Speedup" calculates the average speedup in cases where the best perform is attained on a specific algorithm, and "Ideal Tool" uses the best performance to obtain the global speedup on three platforms.

algorithm has its own advantages, and some new algorithms can not be executed in a reasonable amount of time or memory footprint on all data sets.

In the case of the Intel CPU, we mark MKL's performance as the baseline, which delivers the best performance on only 16.21 percent of the matrix pairs. The DIA method outperforms the baseline on 1,107 matrix pairs better than baseline and yields the best performance on 474, e.g., dw256A $\times$ dwa512 and qpband $\times$ Trefethen_20000. These matrices are almost composed of one or multiple diagonal lines. The COO method outperforms the baseline on 255 matrix pairs and delivers the best results on all of them, e.g., human_gene2 $\times$ appu and msc10848 $\times$ crystk02. The nonzero rate ($\approx$8 percent) and the number of columns of these matrices are large. The ELL method exceeds the baseline on 2,879 matrix pairs and performs optimally on 1,443, e.g., G48 $\times$ G49 and ch7-9-b3 $\times$ ch7-9-b2. The vector-, hash-, heap-based, bhSPARSE, and InCSR_SpGEMM methods perform better than the baseline on 7,006 matrices, and on 4,694 matrix pairs yield the best performance among all methods. For the AMD platform, using the same method to sort the performance of the algorithms, the nine algorithms perform best on 9.09, 3.23, 22.89, 5.65, 9.24, 9.50, 16.36, and 15.37 percent of the cases, respectively. In comparison, AMD benefits more from the diversity of formats and algorithms. For the GPU, the three algorithms (cuSPARSE, NSPARSE, and bhSPARSE) are suitable for almost 85 percent of the matrix pairs. NSPARSE shows advantages in performance on large matrices, whereas the ESC algorithm is effective only on a few matrix pairs.

For parameter selection, as shown in Table 3, the optimal performance parameters are considerably different for various input matrices. For kim1 $\times$ mario001, the best parameter on Intel CPU is $S = 16$ and $b = 2$, but it is $S = 8$ and $b = 2$ on AMD CPU. For the BCSR format, pwtk and mip1 are composed of block units of different sizes, so the optimal block size is not the same.

3.4 Performance Analysis

The existing SpGEMM libraries do not yield the best performance on all matrix pairs. On the Intel architecture, MKL delivers the best performance on approximately 16.21 percent of the dataset, and almost all matrix pairs can be executed within a reasonable time. The improvement in performance is highly correlated with data size, but the overhead of the MKL framework is not expected, especially for small matrix pairs. Our "DIA method" modifies the order of memory access, and reduces the number of sparse accumulation operations and memory consumption when the input matrix pairs satisfy a diagonal distribution. It thus exhibits impressive performance with an average speedup of 72.04x. The "ELL method" is based on the most efficient sparse format for memory access. It significantly improves the efficiency of memory access and saves time in the symbol phase with an average speedup of 9.92x. But this format will still introduce overhead due to padding data for unbalanced row distribution in the matrix pairs. Thus, this method works well for about 35 percent of the dataset. The "COO method" is suitable for specific cases and some matrix pairs still stand out. Furthermore, the vector-, hash-,

TABLE 3
Optimal Parameters of Randomly Selected Ten Matrix Pairs for InCSR_SpGEMM and bmSPARSE Algorithm

Matrix Pair	Optimal Parameter for InCSR format		Matrix Pair	Optimal Parameter for BCSR format
	Intel CPU	AMD CPU		NVIDIA GPU
sstmodel $\times$ model7	S: 8 b: 2	S: 8 b: 2	**pwtk $\times$ hood**	3×1
ex6 $\times$ filter2D	S: 16 b: 4	S: 16 b: 4	**mip1 $\times$ H2O**	2×2
kim1 $\times$ mario001	S: 16 b: 2	S: 8 b: 2	**ldoor $\times$ GL7d16**	7×7
troll $\times$ m14b	S: 32 b: 8	S: 32 b: 8	**circuit5M $\times$ rel9**	2×1
ins2 $\times$ ESOC	S: 8 b: 2	S: 8 b: 2	**rail4284 $\times$ dgreen**	1×2

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

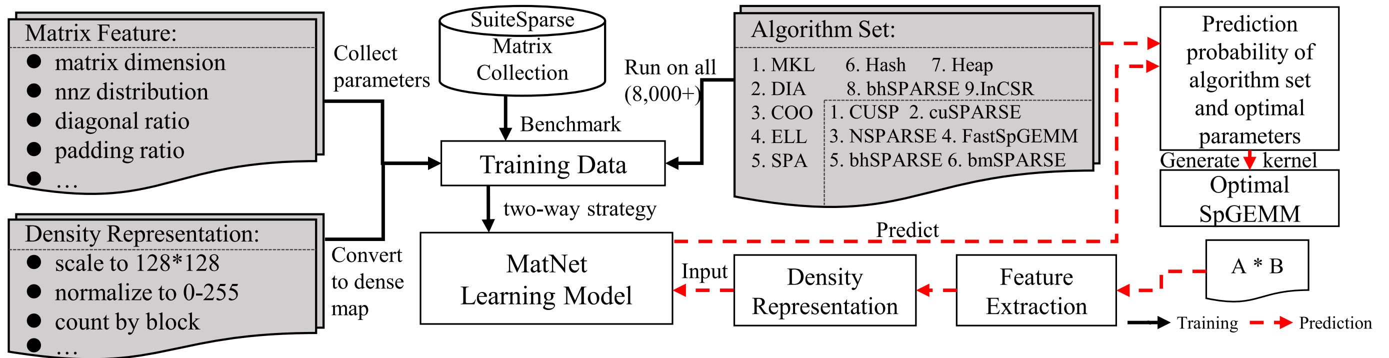

Fig. 4. Overview of the pattern-based SpGEMM library: The solid line indicates the collection and training phase, and the dotted line is the execution flow of the user interface. The collection phase includes extracting two patterns of input and the execution times of all algorithms. The training phase generates the MatNet model by the two-way strategy.

heap-based, bhSPARSE, and InCSR_SpGEMM methods run on their "best of all" matrix pairs to obtaining average speedups of 1.31x, 6.37x, 6.21x, 3.73x, and 9.75x, respectively. In contrast to Intel, AMD's performance has the same proportions, but its absolute performance is slightly lower than Intel's. In addition, because of the higher memory bandwidth needed for the AMD architecture, the "DIA" and "ELL" algorithms deliver better performance. On the GPU platform, compared with the cuSPARSE library, the "ESC", NSPARSE, and bhSPARSE algorithm obtain average speedups of 6.27x, 3.71x, and 3.24x, respectively on their "best of all" cases.

Based on the optimal parameter that leads to the best performance, the InCSR_SpGEMM algorithm performs well for about 9.14 and 15.37 percent of the dataset and can achieve average speedups of 9.75x and 12.18x on Intel CPU and AMD CPU, respectively. The bmSPARSE algorithm can get the best performance on 5.21 percent of the dataset with an average speedup of 2.85x.

We ideally assume that there is an "absolutely perfect" tool that can accurately predict the best choice without any overhead. It would achieve average speedups of 10.52x, 52.21x, and 3.11x for all matrix pairs on the three platforms. Such performance improvements motivate us to design an auto-tuning library.

4 Overview of SpGEMM Library

In the previous section, our experimental results demonstrate the enormous potential for leveraging various formats, algorithms, and parameters on different platforms. We thus develop a pattern-based SpGEMM library to select the best solution on multiple architectures. The overview of this SpGEMM library is shown in Fig. 4. This SpGEMM library consists of two parts: training and prediction. The training part can also be regarded as an offline part. It collects the patterns and performance data of various matrix pairs and trains the autotuning model. The prediction part can also be regarded as an online part to extract the pattern of the input matrix pair, execute the forward propagation of the model to obtain the best algorithm and parameter, convert the matrix pair to other formats if necessary, and finally execute the corresponding SpGEMM calculation. It considers the impact on the performance of matrix patterns and machine configurations for the SpGEMM kernel and is evaluated by thousands of matrix pairs. To achieve this goal, we need a learning model to combine a large number of matrix patterns, algorithms, and machine configurations to find the optimal matching solution. However, it is challenging for a general algorithm to find the most suitable solution in a large search space. Therefore, we first convert the auto-tuning problem into a feature and image classification and a regression problem, compare one machine learning model and two neural network models and select an outstanding hybrid neural network to achieve this goal.

Recognizing the best format, algorithm and parameter is a complex task that requires a large amount of data for training. We use all 2,726 matrices from the SuitSparse Matrix Collection to build 8000+ matrix multiplication pairs, and extract the matrix features and density representations (Sections 4.1 and 4.2) as the input to the training data. We then collect the execution times for various formats, algorithms, and parameters as the output of the training data. Thus, this method incorporates matrix patterns and algorithms together to automatically generate a highly accurate tuner. As shown in Fig. 5, this library is divided into two parts: training and prediction. It first trains the outstanding neural network model called MatNet (Section 4.3) by using the collected training data, then the prediction part indicates the probability of each algorithm and some required parameters to generate the best SpGEMM kernel. In addition, we use a novel multitask-learning strategy that allows information sharing when predicting multiple dependent variables (such as, the optimal algorithm and performance parameters) while including customized layers for each variable. Compared with the traditional single-task training, this multi-task learning strategy improves the model accuracy while simplifying the training process.

Conveniently, the pattern-based SpGEMM library provides a unified interface based on the CSR format, which leads to usability and portability. It can quickly replace existing libraries with the new SpGEMM library. It also supports two usage methods to fit unique needs. The first is one where the library automatically selects the optimal algorithm and required parameters, whereas the other supports the inspector-executor approach. This difference brings two benefits. First, the developer can transparently benefit from multiple formats and algorithms with adjustable parameters. Second, the framework can save the best choice by automatic tuner and reuse the known best solution on the

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

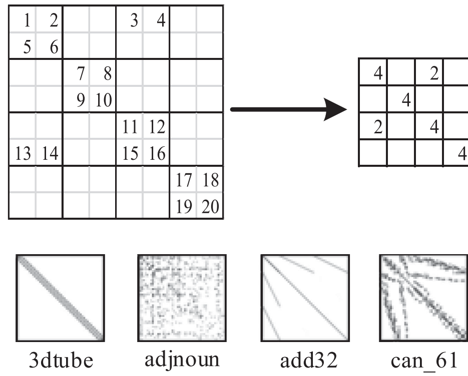

Fig. 5. An example to convert 8×8 matrix to 4×4 density representation.

TABLE 4
Sparse Features and Description

Feature	Description
row, col, nnz	the number of rows, columns and non-zero elements
nnz_ratio	the ratio of non-zero elements in CSR format
max, min, average	the maximun, minimun, and average numbers of non-zero elements
VAR	the variance of non-zero elements
dia_num	the number of diagonals in DIA format
dia_ratio	the number of diagonals divided by all diagonals
dia_pad, ell_pad, InCSR_pad, BCSR_pad	the ratio of padding data in DIA, ELL, InCSR, and BCSR formats
CV	the coefficient of variation of non-zero elements

same matrix to significantly reduce the overhead due to feature extraction and forward propagation of the neural network. Extensibility is also an advantage of this library. Given the inherent characteristics of the neural network, it is open to the addition of new algorithms and training data to improve performance and robustness.

We now introduce the three components of this library: feature extraction, density representation, and the design of the neural network for auto-tuner.

4.1 Feature Extraction

As an automatic input-tuning system, the library first considers 13 fine-grained features related to the distribution and characteristics of five formats for CPU and nine features of four formats for GPU. Some of them intuitively affect the performance of SpGEMM, e.g., the number and ratio of non-zero elements. Other features reflect memory consumption and algorithm performance resulting from the storage structure. Table 4 summarizes all the sparse features used for predicting the optimal SpGEMM algorithm. The first eight features represent the most common structures, including the number of rows and columns, and non-zero elements, which suit all four formats. The ninth and 10th describe diagonal features of the DIA format, including the number of diagonals and the fill ratio of the added zero elements. The 12th expresses the fill ratio of the DIA, ELL, InCSR, and BCSR format. The 13th feature represents the coefficient of variation (CV) of the COO format used in [34] to evaluate the diversity of the number of non-zero elements per row.

4.2 Density Representation

Sparse matrices usually have high sparsity and different sizes while the deployment of convolutional neural network (CNN) generally requires fixed-size input data. This difference leads to two problems. The first is that sparse matrices are usually very large, which causes a large inference overhead for the neural network if complete matrices are used as input. The second problem is that matrix pairs contain large and different numbers of rows and columns, which need to be transformed to the same size. For the image field, the general approach is to shrink large pixels or enlarge small ones to a fixed-size image. This method can also be used to convert the sparse matrix into a small density representation that can represent the coarse-grained patterns of the original matrix with an acceptable size. The density representation as the primary image input to the CNN represents a snapshot matrix that abstracts most of the sparse patterns.

As shown in Fig. 6, we apply this method to map an 8×8 matrix to a 4×4 matrix as an example. The original matrix is divided into 4×4 blocks, and each block is counted by non-zero elements that fill into the corresponding new matrix. Then, the original 8×8 matrix and the 4×4 density representation both contain several diagonals with some irregular non-zero elements. Block count is related to non-zero elements on the matrix, and normalization restricts their number to within a reasonable range ($0 \sim 255$).

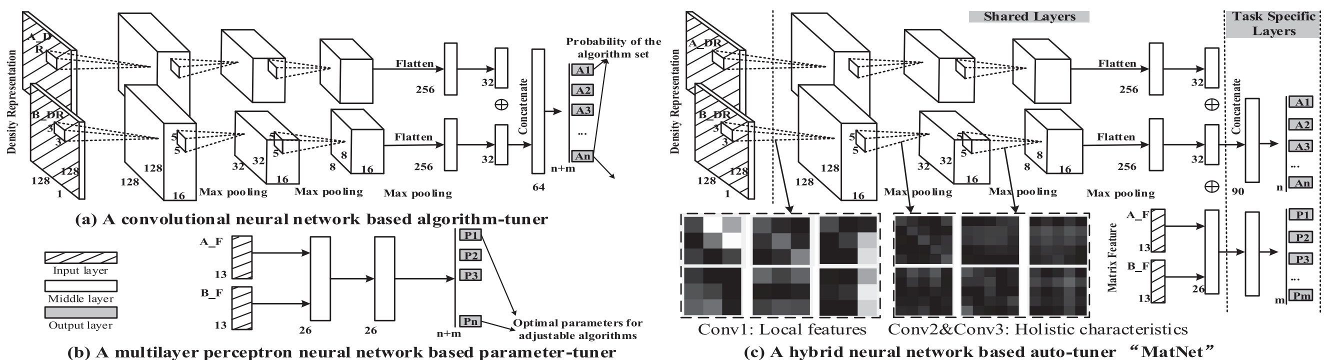

Fig. 6. Details of the two neural network methods, and visualization of some kernels of MatNet. 1) the first method consists of two types of neural networks, in which one is a CNN model (a) for predicting the optimal algorithm, and the other one is an MLP model (b) for predicting the necessary parameter. 2) The second method combines the CNN model and the MLP model by multi-task learning and shares inter-layers for higher accuracy.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

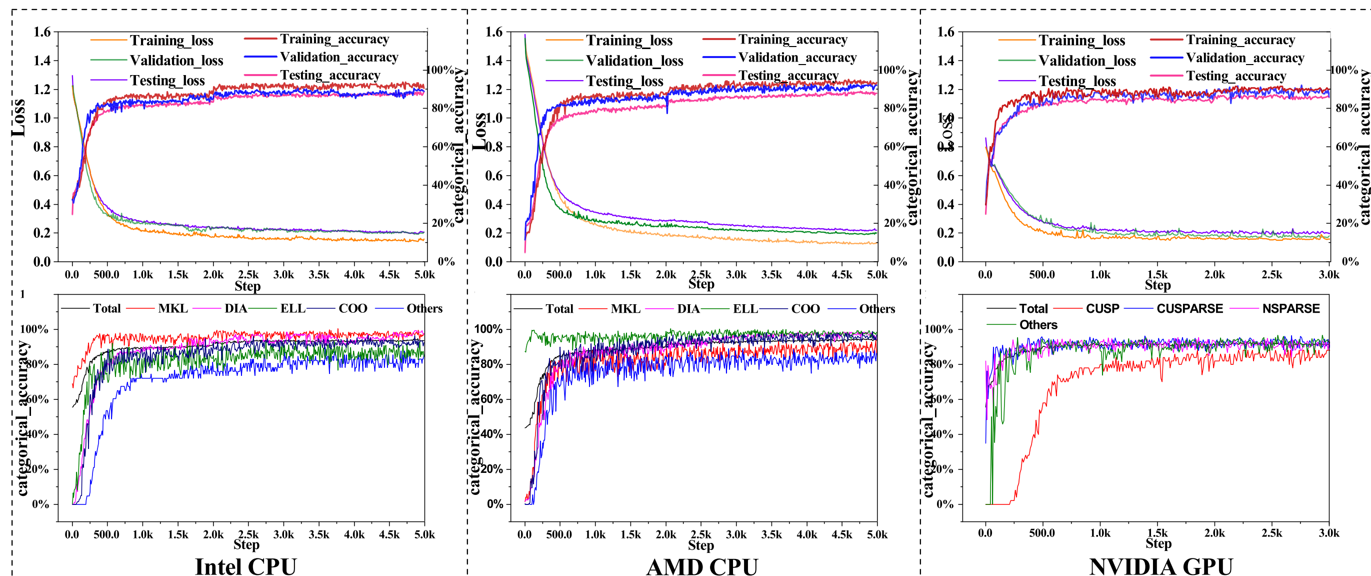

Fig. 7. Loss and accuracy of the MatNet, and details of various formats and algorithms during the training phase.

To ensure sufficient accuracy and acceptable overhead for the neural network, we define 128 × 128 (by comparing with the size of 64 × 64, 128 × 128, and 256 × 256, and choosing the best one) as the size of the density representation and apply the scaling method to map the sparse matrix to the density representation. Four matrices being sampled in Fig. 6 also show the pattern of matrices. Note that any sampling method (such as distance histogram representation [35]) may still lose potential features, which can affect the choice of format and algorithm. An approach is thus is needed to make full or systematic use of these data from different dimensions (fine- and coarse-grained) and complement the loss of accuracy caused by data abstraction (feature extraction and scaling method).

4.3 The Design and Comparison of Neural Network Models

The traditional neural network has delivered impressive results in image classification [36], [37], [38]. For convolutional neural network (CNN), several convolutional layers and pooling layers are used in it to extract high-level features [39], [40] to classify multidimensional data [41]. The standard multilayer perceptron neural network (MLP) is a multi-layer feed-forward network with input, multiple hidden, and output layers. It can be used to learn and store a large number of mappings between the input and output layers for regression tasks.

With regard to our problems, we found two completely different patterns that could be used as inputs from Sections 4.1 and 4.2. And based on the powerful capability of neural networks, we construct two kinds of methods. The first method consists of two types of neural networks, in which one (sub-graph (a) in Fig. 7) is a CNN using the density representation as input for predicting the optimal algorithm, and the other one (sub-graph (b) in Fig. 7) is an MLP using the sparse features as input for predicting the necessary parameter in the parameter-adjustable algorithm (such as, InCSR_SpGEMM and bmSPARSE). The second method (sub-graph (c) in Fig. 7) combines the CNN and the MLP to enhance the ability to classify algorithm and fit parameter simultaneously and leverages the multi-task learning to predict two independent output (the optimal algorithm and parameter) using one neural network model called MatNet. As shown in Fig. 7, these models consist of two types of input patterns, two of which are the density representations of matrices A and B, and are marked as A_DR and B_DR, respectively, and the others are sparse features of matrices A and B and are marked as A_F and B_F, respectively. The first method uses two inputs independently, and the hybrid model MatNet leverages all the inputs simultaneously.

We then define the training data, which include features (13 for CPU and nine for GPU), density representations (128 × 128), the probability of each algorithm, and the optimal parameter for InCSR on CPU and BCSR on GPU. For example, if the execution times of the five algorithms are $T1, T2, T3, T4$, and $T5$, the probability of each algorithm can be calculated as: $Pi = \frac{\frac{1}{Ti}}{\frac{1}{T1}+\frac{1}{T2}+\ldots+\frac{1}{T5}}$, respectively (if a specific algorithm cannot be executed in a reasonable time, then $\frac{1}{Ti}$ is set to zero). Then, the "best choice" corresponds to the algorithm with the highest probability. And the optimal parameter corresponds to the one with the shortest execution time, and the parameter is also set to zero if the optimal algorithm does not require any parameters. Past work trains learning models absolutely and can cause confusion between algorithms that deliver similar performance. Thus, the probability of the output fairly preserves differences that are critical for selection.

By comparing the two types of output, we have two observations: 1) The two outputs are classification and regression problems respectively, which have interference in training. 2) Because the parameters support the choice of format, the two outputs may have some potential relationships.

For the MatNet model, these unrelated input data thus affect each other during the training phase, which affects the accuracy of the network, so we adopt a *two-way* strategy to eliminate interference. Therefore, the training of MatNet

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 5
Two CPUs and One GPU Used for Evaluation

	Intel CPU	AMD CPU	NVIDIA GPU
Core	Xeon E5- 2690 v4 2 processors, 28 cores @2.60 GHz	EPYC 7501 2 processors, 64 cores @2.00 GHz	Tesla P100 56 SMs @1328 MHz
Caches	L1: 32 KB × 14 L2: 256 KB × 14 L3: 35 MB	L1: 32 KB × 32 L2: 512 KB × 32 L3: 64 MB	L2: 4096 KB
Memory	128 GB DDR4-2133 2 × 4 channels	256 GB DDR4-2666 2 × 8 channels	16 GB 1.4 Gbps HBM2
Bandwidth	136.6 GB/s	341 GB/s	732 GB/s

is divided into two separate stages. The first phase trains two kinds of neural networks (CNN and MLP) independently. The second phase maintains all parameters of the previous training and merges all components to update the parameters at the last level. In this way, the mutual influence of features and density representations can be significantly reduced in the first phase, and all outputs can share trained features to improve accuracy in the second phase. With the gradual addition of more training information, the accuracy of prediction can be improved step by step. Moreover, the CNN can produce a number of filters to discriminate local features by using the conv1 layer and holistic characteristics by using the conv2/conv3 layer (some kernels are visualized), and the MLP can aggregate sparse features.

Then we compare the accuracy of these three neural networks in Section 5.2.2. We found that these two inputs, in Sections 4.1 and 4.2, respectively, are not perfect and have shortcomings. Features only capture the fine-grained patterns of the matrix, whereas density representation abstracts from coarse-grained patterns but ignores details. MatNet explores a hybrid model and a separate learning strategy to combine the two patterns, and then achieves the most effective prediction.

Another major advantage of the MatNet model is scalability. With the same training method, the model can easily be deployed on new platforms and new algorithms can be added to it to improve diversity. Because the configurations of the chosen platforms are completely different, the collected training data and the "best result" can vary significantly from one platform to the other. With retraining, MatNet can also achieve high accuracy on these platforms.

5 Evaluation

In this section, we evaluate the speedup of the SpGEMM library by running all matrices in the SuiteSparse Matrix Collection on the three architectures and analyze the accuracy and overhead of MatNet.

5.1 Setup

Platform. We compare the performance of the proposed SpGEMM library on three architectures, as shown in Table 4. Two of them are x86 multicore processors and the other is a manycore processor.

Baseline. The proposed SpGEMM library is compared with several state-of-the-art SpGEMM libraries, such as Intel MKL v19.0.0.117 and the hash-based method [11] for CPU, and NVIDIA cuSPARSE v8.0.61 and NSPARSE [18] for GPU. We enable the OpenMP threading model on both CPU platforms with 28 threads on the dual Intel E5-2690 v4 and 64 threads on the dual AMD EPYC 7501 with the "-O3" option.

Dataset. A total of 2,726 matrices from the SuiteSparse matrix collection are used to randomly construct 8,195 matrix pairs for evaluation, for a total of 220 GB in total. Of this, 60 percent is used for training, 20 percent for validation, and 20 percent for testing. The matrices range in size from 56 KB to 33 GB, and the number of non-zero elements ranges from 4,000 to 2 billion. The dataset includes large and actively growing sets of sparse matrices that arise in practice.

Usage. All experiments in this paper are based on automatic mode, in which each SpGEMM calculation contains the overhead of the prediction of algorithm and parameter as well as the overhead of necessary format conversion. The execution time includes these overheads and one SpGEMM execution time. Moreover, symmetry in the sparse matrices is not used in our SpGEMM algorithm, although some matrices in the benchmark suite are symmetric. The reason is that when the matrix is relatively small, all memory accesses can be placed in the cache, reducing memory consumption is meaningless. When the matrix is particularly large, the memory consumption is much larger than the capacity of LLC or shared memory, so the performance cannot be greatly improved. In addition, the introduction of the symmetry of the matrix requires the conversion of column numbers during SpGEMM calculation, which introduces additional overhead. Therefore, this paper still only focuses on the general sparse matrix-matrix multiplication.

5.2 Results of Training

Fig. 8 gives an overview of the loss and accuracy of MatNet discussed in Section 4.3 for classifying matrix pairs into the best format, algorithm, and parameter on three platforms. Several aspects are compared below.

5.2.1 Comparison of Traditional Machine Learning Model and Two Neural Networks

We used a widely used machine learning algorithm decision tree (CART approach [42]) to compare with two neural network methods. The decision tree is constructed by features of the two matrices. Table 6 shows the performance of the four classifiers on two indicators, where pre. represents precision and recall measures the number of correct results returned. The result shows that MatNet greatly outperforms the decision tree model and is always better than the CNN and MLP model on the two indicators with an average precision of 91.50 percent and recall of 86.57 percent, whereas the decision tree has a precision of 74.19 percent and recall of 67.79 percent, the CNN and MLP model has a precision of 86.24 percent and recall of 81.69 percent. So we choose MatNet as the default classifier in this paper. In addition,

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

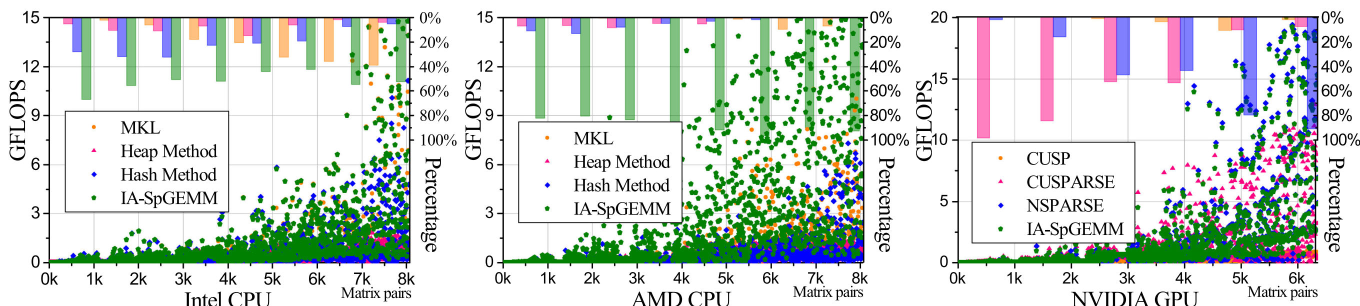

Fig. 8. Performance and proportion of different formats and algorithms on the three architectures. MKL and hash-based method for CPU, and cuSPARSE and NSPARSE for GPU are state-of-the-art libraries.

the results in terms of accuracy on the three platforms also show that our MatNet is an effective cross-platform model.

5.2.2 Loss and Accuracy

The loss function (categorical_crossentropy) is used to indicate how far prediction deviates from the target value. During the training phase, weights are updated based on this quantity. Another indicator is accuracy, which monitors how many cases are correctly predicted. While the network is being trained, loss decreases, and accuracy increases.

The top half of Fig. 8 compares the losses and accuracies for the training, validation, and testing data. For the Intel platform, as the number of steps of iteration increases for independent training (first phase), the loss of MatNet decreases gradually. After 2,000 iterations, the network converges to an almost 85 percent accuracy. We then merge the two independent pieces of training and adjust the learning rate (second phase). The loss continues to decline and accuracy increases slowly, and finally the training process stabilizes at 4,000 iterations. An accuracy of 93 percent is achieved for the training set, and 91 and 90 percent for the validation and testing sets, respectively. On the AMD platform, the network demonstrates similar learning proficiency, and the final accuracy values are 92, 90, and 89 percent. For the GPU platform, MatNet incurs a slightly higher loss but has higher accuracy. The network converges at close to 2,000 iterations. At this time, the intervention is interpolated and training continues. After 3,000 steps, the accuracy values are stable at 92, 90, and 87 percent.

The results show that MatNet quickly learned the characteristics of the matrices and maintained continuous convergence on the three platforms, which also indicates that the density representation and features contain potential connections to the best format and algorithm. In addition, the two-stage training method combines the patterns learned by the CNN and MLP model, and the accuracy of MatNet is significantly improved. With the combination of the two types of training data by using the two-way strategy, MatNet is thus further upgraded.

TABLE 6
Comparison of Results of Prediction of the Two Machine Learning Classification Methods

Platform	Method	Decision Tree		MLP model	
		pre.(%)	recall(%)	pre.(%)	recall(%)
Intel CPU	MKL	79.669	86.294	67.581	74.067
	DIA	81.037	66.384	63.967	61.776
	ELL	54.731	63.157	72.568	82.581
	COO	67.105	70.223	58.835	76.854
	Others	71.875	45.098	64.854	59.956
AMD CPU	MKL	75.021	73.972	69.845	77.267
	DIA	80.174	53.336	78.356	55.295
	ELL	71.875	88.461	62.590	57.376
	COO	76.288	86.064	59.175	73.835
	Others	79.613	58.461	66.732	69.361
NVIDIA GPU	CUSP	66.327	28.571	72.275	77.216
	CUSPARSE	76.428	82.294	65.348	60.825
	NSPARSE	82.667	78.981	80.285	81.475
Platform	**Method**	**CNN model**		**MatNet**	
		pre.(%)	recall(%)	pre.(%)	recall(%)
Intel CPU	MKL	81.275	92.512	88.137	96.160
	DIA	88.536	74.134	94.284	76.741
	ELL	94.135	91.133	96.039	86.607
	COO	84.476	70.762	92.531	78.636
	Others	83.731	81.397	89.201	85.201
AMD CPU	MKL	90.873	84.686	93.277	87.763
	DIA	83.483	85.764	89.512	90.319
	ELL	91.416	79.478	95.424	85.887
	COO	82.764	87.923	87.570	93.392
	Others	84.747	83.689	86.841	89.129
NVIDIA GPU	CUSP	86.843	69.856	92.5	74.326
	CUSPARSE	91.847	89.413	95.215	93.675
	NSPARSE	84.563	78.797	91.961	87.620

5.2.3 Best Algorithm and Parameter

We provide details of the convergence for various formats and algorithms in Fig. 8. It is clear that as the overall accuracy of the network gradually increases, various formats and algorithms exhibit different tendencies of convergence. The main reason for this is that these formats and algorithms account for an unbalanced proportion of records for the training data. For example, the Intel platform's MKL algorithm and the AMD platform's ELL method first converges to high precision by occupying the largest proportion of the training data, and the GPU's cuSPARSE and NSPARSE algorithms exhibit similar rates of convergence using similar numbers of records. Finally, all formats and algorithms achieve accuracy higher than 90 percent.

To evaluate the predicted parameter, we define three indicators to evaluate the difference between the predicted value and the optimal value. Mean Squared Error(MSE)[3] is the average of the squares of the residuals, R-squared(R^2)[4] is used to measure of how far prediction deviates from regression line, and Relative Mean Squared Error(RMSE)[5] represents prediction accuracy. As shown in Table 7, the

3. $MSE = \frac{1}{m}\sum_m^0 (Actual_value - Predicted_value)^2$

4. $R^2 = 1 - \frac{SS_{Regression}}{SS_{Total}}$ (SS_Regression:Sum Squared Regression Error; SS_Total:Sum Squared Total Error;)

5. $RMSE = \frac{1}{m}\sum_m^0 \frac{(Actual_value - Predicted_value)^2}{row\ number\ of\ matrix}$

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 7
Accuracy of MLP and MatNet on Three Statistical Indicators

Adjustable algorithms	MLP model		
	RSS	R^2	RMSE
InCSR_SpGEMM on Intel CPU	0.09	0.31	0.47%
InCSR_SpGEMM on AMD CPU	0.14	0.58	0.70%
bmSPARSE on NVIDIA GPU	0.29	0.82	0.93%
Adjustable algorithms	**MatNet model**		
	RSS	**R^2**	**RMSE**
InCSR_SpGEMM on Intel CPU	0.04	0.22	0.29%
InCSR_SpGEMM on AMD CPU	0.07	0.36	0.46%
bmSPARSE on NVIDIA GPU	0.12	0.56	0.51%

lower three parameters represents a higher accuracy, we compare the indicators for a MLP neural network model.

By comparing two predicted block sizes from the MLP and MatNet models, we found that the MatNet model achieves better accuracy. The MatNet model leverages the multi-task learning method, which uses not only the sparse feature of the input matrices but also the filters trained by the CNN model. So the MatNet method can accurately predict optimal parameters. Note that the accuracy of the predicted parameter on CPU is better than that on GPU. The main reason is that there is more training data on CPU.

5.3 Speedups and Overhead

The results of the speedup of the proposed library are presented in Fig. 9. The predicted formats and algorithms are generated by the MatNet model, and each execution of SpGEMM includes the complete overhead incurred by feature extraction, prediction, and format conversion (if needed). The x-axis represents the sequence of matrix pairs we constructed with incremental non-zeros, and the y-axis on the left side represents the GFLOPS of SpGEMM and the y-axis on the right side calculates the proportion of various optimization methods that deliver the best performance. Compared with the MKL and cuSPARSE method, our algorithm achieves average speedups of 3.89x, 21.14x, and 2.57x. Furthermore, we test the speedups of our SpGEMM library in comparison with state-of-the-art methods ([11] and [18]), and they are 2.45x, 8.22x, and 1.94x on average. The performance gain consists of several parts: (1) A variety of formats significantly reduce the time needed for memory access for the corresponding matrix pairs. (2) The three proposed algorithms change the number of sparse accumulations or reduce memory consumption. (3) Our framework also makes full use of currently available algorithms.

Compared with the best speedup obtained with the "ideal tool" mentioned in Section 3.4, our library can achieve 94 percent of the performance improvement without overhead and 37 percent with overhead on the same dataset. The main reason for the reduction in speedups is that the fixed time for predicting the best format and algorithm is expensive, especially for small matrix pairs.

Overhead is still in our discussion. Note that after collecting the training data, it takes approximately 34 minutes to train the complete MatNet for 4,900 records on two NVIDIA P100 GPUs. In addition, SpGEMM using the pattern-based framework features multiple stages: 1) extracting the density representation and sparse features of the two matrices; 2) predicting the best format and algorithm by MatNet; 3) conversion into various formats (if necessary); and 4) executing the corresponding matrix multiplication kernel. In Fig. 10, a proportion chart shows the average performance breakdown of 12 groups of matrices with increasing sizes. The overhead of the first and the third parts is proportional to the size of the matrix pair, and the second part takes about 0.18 milliseconds per matrix pair. It is clear that the first three performance overheads account for a smaller proportion of the total time as the matrix pairs become larger. Most of the extra overhead incurred by our library is below 20 percent. We thus recommend not using our framework on very small matrix pairs so that the overhead incurred by the auto-tuner does not become another system bottleneck. In addition, the inspector-executor method divides SpGEMM into two stages: analysis and execution. The inspector inspects the matrix patterns and applies format changes, and the executor calls the routine by reusing the predicted results. As the number of computations increases, these overheads are almost completely diluted and the proportion of overhead is significantly reduced.

5.4 Performance Optimization for Real Applications

We deploy our library to two real-world applications and the test results are as follows.

Algebraic Multigrid (AMG). Considering the problem of correcting the equation $Au = f$ with a certain accuracy,

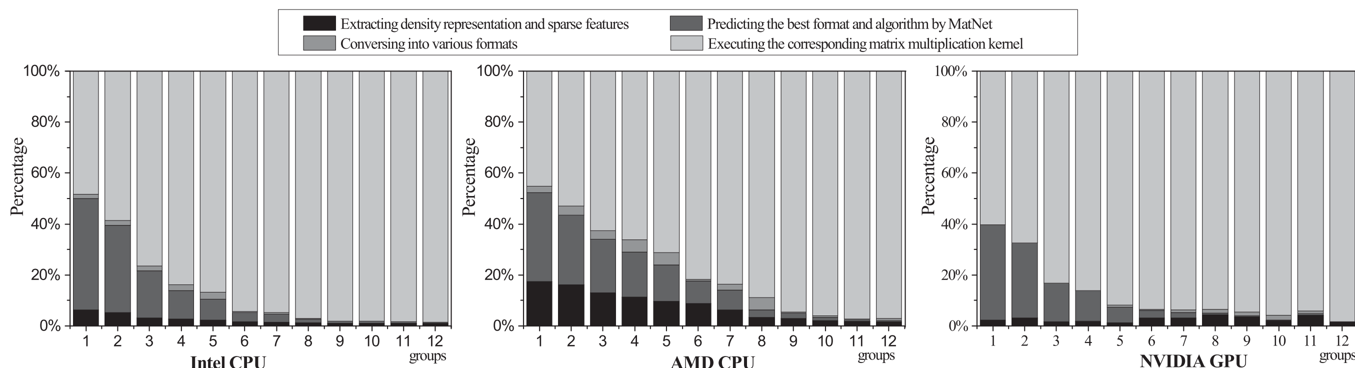

Fig. 9. Performance breakdown of several groups of matrices of different sizes.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

TABLE 8
Performance Improvement of Applying the SpGEMM Library to AMG Application

Platform	Cases	Best algorithm	Best parameter	Speedup for SpGEMM	Speedup for AMG
Intel CPU	2D 5-point	ELL method	-	3.62x	1.29x
	2D 9-point	ELL method	-	3.75x	1.33x
	3D 7-point	ELL method	-	3.93x	1.36x
AMD CPU	2D 5-point	ELL method	-	5.07x	1.31x
	2D 9-point	ELL method	-	5.12x	1.36x
	3D 7-point	ELL method	-	5.74x	1.38x
NVIDIA GPU	2D 5-point	FastSpMM	-	2.41x	1.21x
	2D 9-point	FastSpMM	-	3.51x	1.27x
	3D 7-point	FastSpMM	-	3.75x	1.28x

(The best parameter being null means that the selected algorithm does not need any parameters).

where A is a large sparse matrix, u and f are dense vectors. In particular, Galerkin products multiply three sparse matrices in each level of an AMG hierarchy, where rectangular matrix P^T is a restriction operator, square matrix A is initially the system matrix, and rectangular matrix P is a prolongation operator.

The input problem is from the 2D 5-point, 2D 9-point, 3D 7-point Poisson problem. The two 2D problems have dimensions 1024×1024 and generate system matrices of size 1048576×1048576. The two 3D problems have dimensions $101 \times 101 \times 101$ and generate system matrices of size 1030301×1030301. Table 8 show the best algorithm, parameter, and speedup of Galerkin products P^TAP in constructing an AMG hierarchy for a smoothed aggregation preconditioner. Our library selects the ELL format and ELL method for the best performance because the matrix built by AMG has the same non-zero elements in each row. Thus we deliver average speedups of 3.77x, 5.31x, and 3.22x on the SpGEMM kernel and speedups of 1.32x, 1.35x, and 1.25x on the entire AMG application by using automatic mode. The overhead of format conversion and model inference account for 2 percent of the total SpGEMM execution time.

Breadth First Search (BFS). Many graph processing algorithms perform multiple BFS to discover the potential of internal connections, For example, Betweenness Centrality measures the centrality and the shortest paths of an unweighted graph. In linear algebraic terms, this algorithm corresponds to multiplying a square sparse matrix with a tall-skinny matrix. The square one represents the graph and the tall-skinny one represents the stack of frontiers, each column representing one BFS frontier. In the memory-efficient implementation of the Markov clustering algorithm [6], a matrix is multiplied with a subset of its column, representing another use case of multiplying a square matrix with a tall-skinny matrix.

In our evaluations, we choose three representative graphs for SpGEMM BFS from the network repository and generate the tall-skinny matrix by randomly selecting columns from the graph itself. Table 9 shows the best algorithm, parameter, and speedup for the BFS application. Our library selects different algorithms for three graphs on three platforms because the graphs have completely different patterns and therefore require specific algorithms. Thus we deliver average speedups of 3.58x, 4.49x, and 2.57x on the entire BFS application by using automatic mode. The overhead of format conversion and model inference account for 3 percent of the total execution time.

TABLE 9
Performance Improvement of Applying the SpGEMM Library to BFS Application

Platform	Cases	Best algorithm	Best parameter	Speedup for SpGEMM
Intel CPU	sc-msdoor	bhSPARSE	-	1.67x
	roadNet-TX	DIA method	-	6.28x
	soc-pokec	InCSRSpGEMM	S: 32 b:8	2.79x
AMD CPU	sc-msdoor	bhSPARSE	-	2.62x
	roadNet-TX	DIA method	-	8.12x
	soc-pokec	InCSRSpGEMM	S: 32 b:16	2.74x
NVIDIA GPU	sc-msdoor	NSPARSE	-	1.41x
	roadNet-TX	bhSPARSE	-	2.51x
	soc-pokec	bmSPARSE	-	3.82x

5.5 Usage

In this section, we open-source this pattern-based SpGEMM library with a unified interface for the SPGEMM kernel and provide a test case to compute $A * B$ for validating the results of the prediction of MatNet. The source code is available at *https://github.com/zhen-xie/IA-SpGEMM.git*. In addition, our model can be easily extended to more platforms and algorithms by collecting more training records and fine-tuning the MatNet model.

6 Related Work

Sparse kernels have been widely used to improve higher efficiency in a number of HPC applications [1], [43], [44], [45], [45], [46], [47], [48], [49]. Various approaches have been proposed to optimize data dependence and unbalanced sparse computations. Venkat *et al.* [50], [51], [52], [53] developed several techniques for dependence analysis and data transformation optimization for sparse computations during the compilation phase, Arash *et al.* [54] used a performance model and a blocking mechanism to resolve the problem of load imbalance. This paper focuses on the format, algorithm, and auto-tuner for the SpGEMM kernel.

SpGEMM was parallelized and optimized on CPUs. The most significant difference between these algorithms is the method used for nonzero accumulation. As in the COO algorithms used in this paper, the dense accumulator [10], [55] is a general solution, whereas other methods involve sorting a heap [12] or merging rows [56]. Moreover, a few GPU algorithms have been proposed, CUSP [17] uses an expand-sort-compress (ESC) algorithm that pre-allocates and collects all intermediate results, and accumulates them through sort and compression operations. cuSPARSE [13], NSPARSE [18] and Kokkos [57] uses a hash table to combine the intermediate results in global memory. bhPARSE [58] first assigns rows into bins by the size of the intermediate result and output, and launches various kernels. The hybrid method [14], [59], multiple-levels algorithm [12], and row merge algorithm [60] can also show good performance on partial matrices. These algorithms can be added to our framework to yield better performance. In addition, our framework proposed in this paper resolves the problem on a single node and, in many cases, dominates the whole overhead. We would like to see that the following work could integrate our framework into distributed SpGEMM implementations [5], [61], [62].

Selecting the best format and algorithm has received considerable attention in recent years. The work closest to this

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

study is that by Zhao *et al*.[35], which for the first time used a CNN to select the matrix format for SpMV and yielded an accuracy of 93 percent. Several studies [63], [64], [65], [66] have been devoted to the best storage formats through auto-tuning methods or automatically disable the auto-tuning when it is not likely to be beneficial, but some methods may be limited owing to the learning ability of the models applied. Moreover, choosing the best format can be seen as a classification problem, and is similar to recognizing hand-written digits, which was one of the first applications of CNN. The LeNET-5 [67] was developed for this task. The FFNN [68] is also widely used for the classification model. Unlike SpMV auto-tuners, our algorithm needs to consider the patterns of the two arbitrary matrices at the same time and classify them into appropriate directories. We thus introduced these two neural networks to automatic tuning and designed a new convolutional neural network (MatNet) to connect them for the SpGEMM. We found that sparse kernels can benefit from the neural network method. Extending neural networks to more sparse kernels can also help reveal connections between optimization methods and specific parameters.

7 Conclusion

In this work, we proposed a variety of SpGEMM algorithms for DIA, COO, and ELL formats, and presented a pattern-based SpGEMM library that can automatically determine the best format, algorithm, and parameter for any sparse matrix pairs. It gathers a set of SpGEMM algorithms that naturally allow for the use of a deep learning model (MatNet) to predict the best choice by using features and density representation. The results show that our library yields better performance than four other state-of-the-art libraries. We also expect more sparse and input-sensitive algorithms can be inspired by our method.

Acknowledgments

We would like to express our gratitude to all reviewers for their constructive comments. This work was supported by the National Key Research and Development Program of China under Grants 2017YFB0202105, 2016YFB0201305, 2016YFB0200803 and 2016YFB0200300 and National Natural Science Foundation of China under Grants 61521092, 91430218, 31327901, 61472395, 61432018 and 61671151. Article IA-SpGEMM: An Input-aware Auto-tuning Framework for Parallel Sparse Matrix-Matrix Multiplication [1] published at IC19.

References

[1] Z. Xie, G. Tan, W. Liu, and N. Sun, "IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proc. ACM Int. Conf. Supercomputing*, 2019, pp. 94–105.

[2] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM J. Sci. Comput*., vol. 34, no. 4, pp. C123–C152, 2012.

[3] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Comput. Sci. Eng.*, vol. 10, no. 2, pp. 20–25, Mar./Apr. 2008.

[4] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "HipMCL: A high-performance parallel implementation of the Markov clustering algorithm for large-scale networks," *Nucleic Acids Res.*, vol. 46, no. 6, pp. e33–e33, 2018.

[5] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM J. Sci. Comput*., vol. 34, no. 4, pp. C170–C191, 2012.

[6] V. B. Shah, *An Interactive System for Combinatorial Scientific Computing With an Emphasis on Programmer Productivity*. Santa Barbara, CA, USA: Univ. California Press, 2007.

[7] R. Yuster and U. Zwick, "Detecting short directed cycles using rectangular matrix multiplication and dynamic programming," in *Proc. 15th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2004, pp. 254–260.

[8] D. Bader, "Graph BLAS forum," [Online]. Available: https://graphblas.org

[9] R. Intel, "Intel math kernel library reference manual," 2019. [Online]. Available: https://software.intel.com/sites/default/files/mkl-2019-developer-reference-c_0.pdf

[10] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse matrices in matlab: Design and implementation," *SIAM J. Matrix Anal. Appl*., vol. 13, no. 1, pp. 333–356, 1992.

[11] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "High-performance sparse matrix-matrix products on Intel KNL and multicore architectures," 2018, *arXiv: 1804.01698*.

[12] A. Azad *et al*., "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM J. Sci. Comput*., vol. 38, no. 6, pp. C624–C651, 2016.

[13] J. Demouth, "Sparse matrix-matrix multiplication on the GPU," in *Proc. GPU Technol. Conf.*, 2012.

[14] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 370–381.

[15] P. A. Golnari and S. Malik, "Sparse matrix to matrix multiplication: A representation and architecture for acceleration (long version)," 2019, *arXiv: 1906.00327*.

[16] G. Ortega, F. Vázquez, I. García, and E. M. Garzón, "FastSpMM: An efficient library for sparse matrix matrix product on GPUs," *Comput. J.*, vol. 57, no. 7, pp. 968–979, 2014.

[17] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix–matrix multiplication for the GPU," *ACM Trans. Math. Softw*., vol. 41, no. 4, 2015, Art. no. 25.

[18] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU," in *Proc. 46th Int. Conf. Parallel Process.*, 2017, pp. 101–110.

[19] M. Deveci, C. Trott, and S. Rajamanickam, "Performance-portable sparse matrix-matrix multiplication for many-core architectures," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2017, pp. 693–702.

[20] A. Li *et al*., "Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2017, pp. 26:1–26:14.

[21] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaSpMV: Yet another SpMV framework on GPUs," in *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 107–118.

[22] 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 J. Sci. Comput*., vol. 36, no. 5, pp. C401–C423, 2014.

[23] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Comput*., vol. 49, no. C, pp. 179–193, 2015.

[24] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurrency Comput. Pract. Experience*, vol. 29, no. 21, 2017, Art. no. e4244.

[25] X. Wang, W. Liu, W. Xue, and L. Wu, "swSpTRSV: A fast sparse triangular solve with sparse level tile layout on sunway architectures," in *Proc 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 338–353.

[26] B.-Y. Su and K. Keutzer, "clSpMV: A cross-platform OpenCL SpMV framework on GPUs," in *Proc. 26th ACM Int. Conf. Supercomputing*, 2012, pp. 353–364.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

[27] 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, pp. 46:1–46:25, 2018.
[28] J. Zhang and L. Gruenwald, "Regularizing irregularity: Bitmap-based and portable sparse matrix multiplication for graph data on GPUs," in *Proc. 1st ACM SIGMOD Joint Int. Workshop Graph Data Manage. Experiences Syst. Netw. Data Analytics*, 2018, pp. 1–8.
[29] R. D. Falgout and U. M. Yang, "Hypre: A library of high performance preconditioners," in *Proc. Int. Conf. Comput. Sci.*, 2002, pp. 632–641.
[30] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 804–811.
[31] Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*. Scotts Valley, CA, USA: CreateSpace Independent Publishing Platform, 1990.
[32] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Trans. Math. Softw.*, vol. 4, no. 3, pp. 250–269, 1978.
[33] P. N. Q. Anh, R. Fan, and Y. Wen, "Balanced hashing and efficient GPU sparse general matrix-matrix multiplication," in *Proc. Int. Conf. Supercomputing*, 2016, pp. 1–12.
[34] W. Abu-Sufah and A. A. Karim, "Auto-tuning of sparse matrix-vector multiplication on graphics processors," in *Proc. Int. Supercomputing Conf.*, 2013, pp. 151–164.
[35] Y. Zhao, C. Liao, J. Li, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proc. 23rd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2018, pp. 94–108.
[36] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, 2015.
[37] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2014, pp. 580–587.
[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Advances Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[39] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 818–833.
[40] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition," in *Proc. Int. Conf. Artif. Neural Netw.*, 2010, pp. 92–101.
[41] L. Wei, Y. Yang, R. M. Nishikawa, and Y. Jiang, "A study on several machine-learning methods for classification of malignant and benign clustered microcalcifications," *IEEE Trans. Med. Imag.*, vol. 24, no. 3, pp. 371–380, Mar. 2005.
[42] L. Breiman, J. Friedman, C. Stone, and R. Olshen, *Classification and Regression Trees*. New York, NY, USA: Taylor & Francis, 1984. [Online]. Available: https://books.google.com/books?id=JwQx-WOmSyQC
[43] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 806–814.
[44] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 315–323.
[45] Z. Xie, W. Dong, J. Liu, H. Liu, and D. Li, "Tahoe: Tree structure-aware high performance inference engine for decision tree ensemble on GPU," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 426–440.
[46] Z. Xie, Z. Cao, Z. Wang, D. Zang, E. Shao, and N. Sun, "Modeling traffic of big data platform for large scale datacenter networks," in *Proc. IEEE 22nd Int. Conf. Parallel Distrib. Syst.*, 2016, pp. 224–231.
[47] W. Dong, J. Liu, Z. Xie, and D. Li, "Adaptive neural network-based approximation to accelerate eulerian fluid simulation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2019, pp. 1–22.
[48] W. Dong, Z. Xie, G. Kestor, and D. Li, "Smart-PGSim: Using neural network to accelerate AC-OPF power grid simulation," 2020, *arXiv: 2008.11827*.
[49] Z. Xie, W. Dong, J. Liu, I. Peng, Y. Ma, and D. Li, "MD-HM: Memoization-based molecular dynamics simulations on big memory system," in *Proc. ACM Int. Conf. Supercomputing*, 2021, pp. 215–226.
[50] A. Venkat *et al.*, "Automating wavefront parallelization for sparse matrix computations," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2016, Art. no. 41.
[51] K. Ahmad, A. Venkat, and M. Hall, "Optimizing LOBPCG: Sparse matrix loop and data transformations in action," in *Proc. Int. Workshop Lang. Compilers Parallel Comput.*, 2016, pp. 218–232.
[52] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 521–532, 2015.
[53] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the GPU microarchitecture to achieve bare-metal performance tuning," *ACM SIGPLAN Notices*, vol. 52, no. 8, pp. 31–43, 2017.
[54] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan, "A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs," *J. Parallel Distrib. Comput.*, vol. 76, pp. 3–15, 2015.
[55] M. M. A. Patwary *et al.*, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *Proc. Int. Conf. High Perform. Comput.*, 2015, pp. 48–57.
[56] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL-A high level linear algebra library for GPUs and multi-core CPUs," in *Proc. Int. Workshop GPUs Sci. Appl.*, 2010, pp. 51–56.
[57] M. Deveci, C. Trott, and S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures," *Parallel Comput.*, vol. 78, pp. 33–46, 2018.
[58] W. Liu, "Parallel and scalable sparse basic linear algebra subprograms," Ph.D. dissertation, Dept. Fac. Sci., Copenhagen Univ. Copenhagen, Denmark, 2015.
[59] W. Liu and B. Vinter, "A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors," *J. Parallel Distrib. Comput.*, vol. 85, no. C, pp. 47–61, 2015.
[60] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann, "GPU-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM J. Sci. Comput.*, vol. 37, no. 1, pp. C54–C71, 2015.
[61] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication," in *Proc. 27th ACM Symp. Parallelism Algorithms Architectures*, 2015, pp. 86–88.
[62] G. Ballard, C. Siefert, and J. Hu, "Reducing communication costs for sparse matrix multiplication within algebraic multigrid," *SIAM J. Sci. Comput.*, vol. 38, no. 3, pp. C203–C231, 2016.
[63] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 99–108.
[64] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: An input adaptive auto-tuner for sparse matrix-vector multiplication," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 117–126, 2013.
[65] 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.
[66] Y. Zhao, W. Zhou, X. Shen, and G. Yiu, "Overhead-conscious format selection for SpMV-based applications," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 950–959.
[67] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[68] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, "A theoretical framework for back-propagation," in *Proc. Connectionist Models Summer School*, 1988, pp. 21–28.

Zhen Xie received the BS degree in computer science from the Wuhan University of Technology, China, in 2013, and the PhD degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2019. He is currently a postdoctoral fellow at the University of California, Merced, California. His research interests include parallel algorithms and performance optimization, with a recent focus on high performance computing for machine learning.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.

Guangming Tan (Member, IEEE) received the BS degree in mathematics from Xiangtan University, China, in 2002, and the PhD degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2008. He is a professor with the Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Science, China. His research interests include focus on parallel computing, domain-specific architecture, and big data.

Weifeng Liu (Senior Member, IEEE) received the BE and ME degrees in computer science, both from the China University of Petroleum, Beijing, China, in 2002 and 2006, respectively, and the PhD degree from Niels Bohr Institute, University of Copenhagen, Denmark, in 2016. He is currently a full professor with the Department of Computer Science and Technology, China University of Petroleum, Beijing, China. His research interests include numerical linear algebra and parallel computing, particularly in designing parallel and scalable algorithms and data structures for sparse matrix computations on throughput-oriented architectures. He is a member of the ACM and the SIAM.

Ninghui Sun (Member, IEEE) received the BS degree from Peking University, China, in 1989, and the MS and PhD degrees both in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 1992 and 1999, respectively. He is a professor with the Institute of Computing Technology, Chinese Academy of Sciences, China. He is an academician of the Chinese Academy of Engineering. His research interests include computer architecture, operating system, and parallel algorithm.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: China University of Petroleum. Downloaded on July 19,2021 at 08:30:28 UTC from IEEE Xplore. Restrictions apply.