IA-SpGEMM: An Input-aware Auto-tuning Framework for Parallel Sparse Matrix-Matrix Multiplication

Zhen Xie[◇†], Guangming Tan[◇†], Weifeng Liu[‡], Ninghui Sun[◇†*]
[◇]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
[†]University of Chinese Academy of Sciences
[‡]Department of Computer Science and Technology, China University of Petroleum, Beijing
xiezhen@ncic.ac.cn,tgm@ict.ac.cn,weifeng.liu@cup.edu.cn,snh@ict.ac.cn

ABSTRACT

Sparse matrix-matrix multiplication (SpGEMM) is a sparse kernel that is used in a number of scientific applications. Although several SpGEMM algorithms have been proposed, almost all of them are restricted to the compressed sparse row (CSR) format, and the possible performance gain from exploiting other formats has not been well studied. The particular format and algorithm that yield the best performance for SpGEMM also remain undetermined.

In this work, we conduct a prospective study on format-specific parallel SpGEMM algorithms, and analyze their pros and cons. We then propose IA-SpGEMM, an input-aware auto-tuning Framework for SpGEMM, that provides a unified programming interface in the CSR format and automatically determines the best format and algorithm for arbitrary sparse matrices. For this purpose, we set-up an algorithm set and design a deep learning model called MatNet that is trained by over 2,700 matrices from the SuiteSparse Matrix Collection to quickly and accurately predict the best solution by using sparse features and density representations. We evaluate our framework on CPUs and a GPU, and the results show that IA-SpGEMM is on average 3.27x and 13.17x faster than MKL on an Intel and an AMD platform, respectively, and is 2.23x faster than cuSPARSE on an NVIDIA GPU.

CCS CONCEPTS

• **Mathematics of computing** → **Mathematical software performance**; **Computations on matrices**; • **Theory of computation** → **Parallel algorithms**;

KEYWORDS

Auto-tuning, SpGEMM, Sparse Format, Neural Network

ACM Reference format:
Zhen Xie[◇†], Guangming Tan[◇†], Weifeng Liu[‡], Ninghui Sun[◇†*]. 2019. IA-SpGEMM: An Input-aware Auto-tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of 2019 International Conference on Supercomputing, Phoenix, AZ, USA, June 26–28, 2019 (ICS '19),* 12 pages.
https://doi.org/10.1145/3330345.3330354

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330354

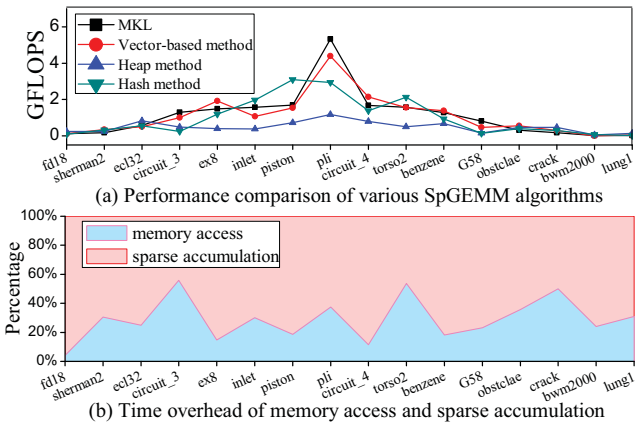

Figure 1: Comparison of performance of different algorithms and their overhead.

1 INTRODUCTION

Sparse matrix-matrix multiplication (SpGEMM) is a key kernels 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) [10]. Many graph processing operations, such as breadth-first search [21], Markov clustering [6], graph contraction [21], subgraph extraction [13], peer pressure clustering [49], and cycle detection [58], can be expressed as SpGEMM. GraphBLAS [7] also defines matrix-based graph algorithms. Efficient SpGEMM algorithms are thus crucial for these applications to achieve higher performance.

Several SpGEMM libraries are widely used, including the Intel MKL [26], vector-based sparse accumulator (SPA) [20], hash-based method [41], heap-based method [5], cuSPARSE [17], and CUSP [16] proposed for Nvidia and NSPARSE [42]. However, these libraries are sensitive to sparse input matrices and thus exhibit significant fluctuations in performance. In Figure 1(a), we compare the performance of various algorithms by calculating $A * A^T$ on an Intel CPU (as in Section 5.1). It is clear that different algorithms deliver their best performance on different matrices, and no single algorithm dominates on all datasets in terms of performance. 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, Figure 1(b) 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 [13, 18] 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 [31]. Much of the research on SpMV and SpTRSV has been dedicated to optimizing memory access by excavating classic storage formats [27, 35, 39, 55, 57] with promising results [51, 52]. Back to SpGEMM, such classic storage formats, as DIA, COO and ELL, can reduce memory requirements or accelerate memory access on vector architectures and change the order of the calculation process, or can even reduce the number of sparse accumulation operations. The other motivation of this work is to explore the influence on several classic storage formats on SpGEMM.

In this paper we design multiple SpGEMM algorithms based on a variety of widely used sparse storage formats and analyze the conditions conducive to better performance. Therefore, in order to integrate our implementations with the existing SpGEMM libraries and choose the optimal algorithm, we propose an input-aware auto-tuning framework for SpGEMM (IA-SpGEMM), which classifies two input matrices into the most appropriate category among the assembled SpGEMM algorithm set by employing a novel convolutional neural network called MatNet. 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 given architecture. Moreover, we extract the sparse features and density representation of the two input matrices as input to the training data. MatNet is then generated by using the collected performance data, extracted features, and density representations. Compared with traditional machine learning or empirical models, MatNet is 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, IA-SpGEMM provides a unified interface in the CSR format to quickly predict the best format and algorithm for two input matrices, and the matrices are finally executed by the corresponding implementation with possible format conversion. We evaluate the IA-SpGEMM 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 3.27x and 13.17x faster than the Intel MKL on dual Intel Xeon E5-2620 and dual AMD EPYC 7501, respectively, with an accuracy of 93%, and 2.23x faster than the NVIDIA cuSPARSE library on Tesla P100 with an accuracy of 91%. 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, a significant performance gaps naturally leads to the adoption of an auto-tuning model.
- We propose a convolutional neural network called MatNet to select the best format and algorithm from a large algorithm set. In benchmarking more than 8,000 matrix multiplication pairs, the predictive accuracy of MatNet was over 93%. It largely avoids the tedious work of manual choice and improves scalability to benefit from all algorithms with an acceptable overhead.
- We develop an input-aware auto-tuning Framework for SpGEMM (IA-SpGEMM) with a general interface based on the CSR format. Users can thus transparently obtain the best performance. We implement our framework on three processors and yield average speedups of 3.27x, 13.17x, and 2.23x.

2 BACKGROUND

2.1 Sparse Matrix Storage Format

The sparse storage format defines the structure used to storage 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 continuous 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 [45] for a more detailed illustration). Figure 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.

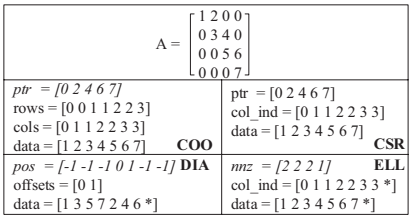

Figure 2: An example of the four sparse matrix formats, where the italics represent small changes. The COO format adds a row offset array, the DIA format adds a diagonal position array, and the ELL format adds an array for counting nnzs per row. All formats are sorted in row order.

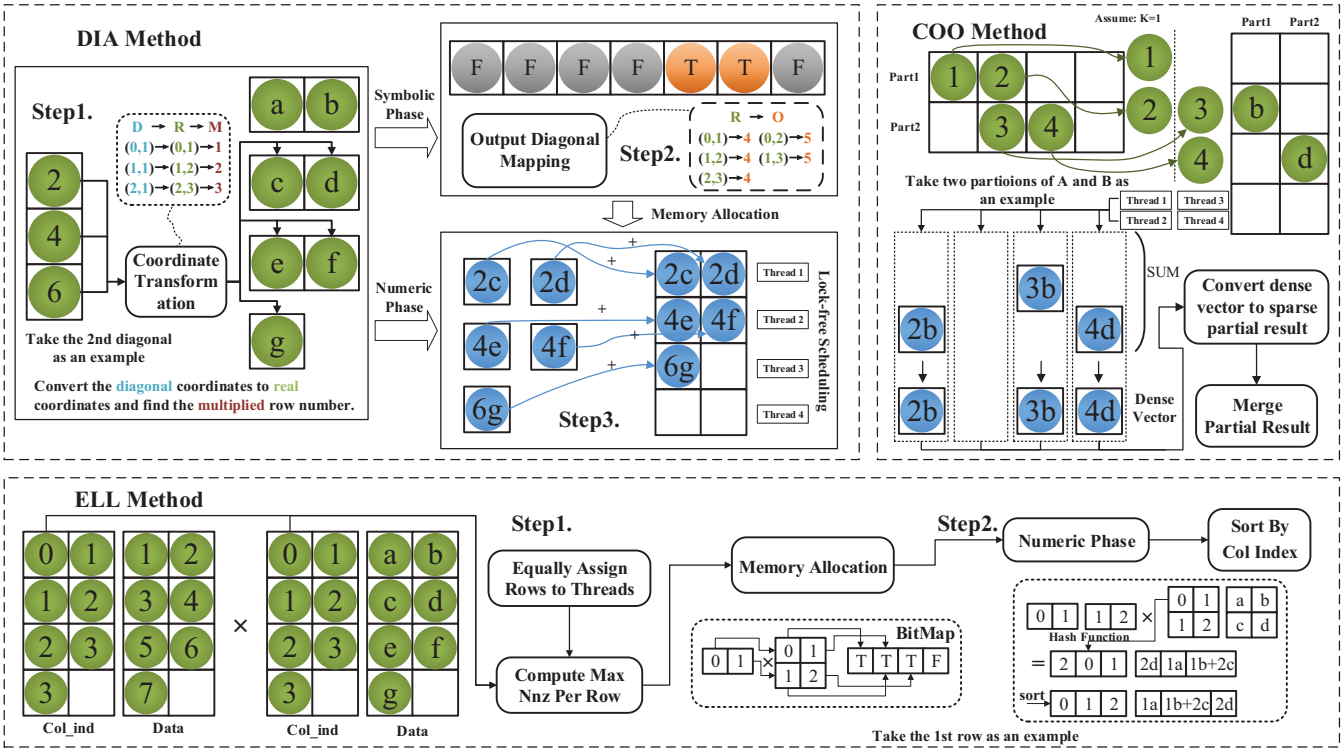

Figure 3: Flowchart of three format-specific algorithms and some examples of $A * A'$. The DIA method shows the processes of coordinate transformation and partial accumulation. The COO method divides matrices by k=1 and shows that the length of the dense vector is reduced to that of the previous quarter. 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.

2.2 Parallel SpGEMM Method

Let matrix A have size $m * n$ and B have size $n * k$. The matrix product is $C = AB$. The element in the i-th row and j-th 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 [25] and improved on MATLAB by Gilbert et al. [20]. 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 [13], a hash table [17, 42], or the "expansion, sorting, and compression" (ESC) method [16]. Some of these algorithms are included in our algorithm set.

Algorithm 1 Row-wise SpGEMM algorithm for $C = A * B$. 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: #parallel for
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] ← 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 by using $A * A'$ as an example [1]. We also compare six CPU algorithms and three 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 Figure 3, the multiplication proceeds generally as follows: Step 1: Each element of the diagonal line is first converted into real coordinates by coordinate transformation to obtain the multiplied row number (an example of the second diagonal of A is given in the figure). 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. Algorithm 2 significantly reduces the overhead due to memory access for the diagonal matrix and directly adds the intermediate results to the target address without extra memory consumption. Note

[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.

that row-based thread scheduling takes a row as the minimum unit such that it so it achieves load balancing and avoids 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."

Algorithm 2 DIA Method

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

The COO format separately stores non-zero elements of the same row, because of which the flexible format can more easily to 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 is successively computed for a part of C by the SPA method [20] and all the partial results are finally merged. As shown in Figure 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."

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). The memory of C is then determined. 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 hash table to store and accumulate the intermediate results. All partial

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
```

results are mapped to the corresponding positions by calculating the hash values of the column indices or keeping plus one whenever collision occurs. Finally, the disordered matrix C is sorted. Algorithm 4 has two main advantages: (1) Because the *Col_ind* is placed in continuous 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 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_function(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 Overview of algorithm set

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

Table 1: Performance statistics : "Dominance" and "Percentage" represent the number and proportion of the best and the better than baseline for various algorithms. "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.

	Method	Dominance		Percentage		Average Speedup	Speedup by "Ideal Tool"
		Best of all	Over BL.	Best of all	Over BL.		
Intel CPU	MKL (Baseline)	2874	-	35.07%	-	-	8.94x
	DIA method	491	1107	5.99%	13.51%	72.04x	
	COO method	283	506	3.45%	6.17%	7.63x	
	ELL method	1496	2879	18.26%	35.13%	9.92x	
	SPA vector-based	259	748	3.16%	9.13%	1.31x	
	Hash-based	2150	4307	26.24%	52.56%	6.37x	
	Heap-based	642	1951	7.83%	23.81%	6.21x	
AMD CPU	MKL (Baseline)	1708	-	20.96%	-	-	46.16x
	DIA method	745	1544	9.14%	18.94%	346.0x	
	COO method	342	586	4.20%	7.19%	8.20x	
	ELL method	1989	3044	24.40%	37.35%	32.96x	
	SPA vector-based	830	2529	10.18%	31.03%	1.58x	
	Hash-based	1757	2363	21.56%	28.99%	21.40x	
	Heap-based	779	1015	9.56%	12.45%	12.18x	
NVIDIA GPU	cuSPARSE(Baseline)	3827	-	51.07%	-	-	2.40x
	CUSP	208	769	2.78%	10.26%	6.27X	
	NSPARSE	3459	3525	46.16%	47.04%	3.71X	

build 8000+ matrix multiplication pairs by using all the matrices in the SuiteSparse Matrix Collection and compare the performance of these algorithms.

Table 2: Seven algorithms for CPU and three algorithms for GPU.

CPU	Intel MKL v19.0.0.117 mkl_sparse_sp2m (CSR) [26]
	DIA method (Algorithm 2)
	COO method (Algorithm 3)
	ELL method (Algorithm 4)
	SPA vector based method (CSR) [20]
	Hash based method (CSR) [41]
	Heap based method (CSR) [5]
GPU	CUSP v0.5.1 based on ESC method (COO) [16]
	cuSPARSE v8.0.61 (CSR) [17]
	NSPARSE (CSR) [42]

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 memory expansion to no more than five times due to format conversion. The execution times of all algorithms could be no longer than five times than of the MKL or cuSPARSE, which also means that these matrix pairs are not suitable for a specific format or algorithm.

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 on all matrix pairs. In the case of the Intel CPU, we mark MKL's performance as the baseline, which delivers the best performance on only 35% of the matrix pairs. The DIA method outperforms the baseline on 1,107 matrix pairs better than baseline and yields the best performance on 491, e.g., dw256A*dwa512 and qpband*Trefethen_20000. These matrices are almost composed of one or multiple diagonal lines. The COO method outperforms the baseline on 506 matrix pairs and delivers the best results on approximately half of them, e.g., human_gene2*appu and msc10848*crystk02. The non-zero rate (≈8%) 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 1,496, e.g., G48*G49 and ch7-9-b3*ch7-9-b2. The vector-, hash-, and heap-based methods perform better than the baseline on 7,006 matrices, and on 3,051 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 seven algorithms perform best on 20.96%, 9.14%, 4.20%, 24.40%, 10.18%, 21.56%, and 9.56% of the cases, respectively. In comparison, AMD benefits more from the diversity of formats and algorithms. For the GPU, the two algorithms (cuSPARSE and NSPARSE) are suitable for almost half of the matrix pairs. NSPARSE shows advantages in performance on large matrices, whereas the ESC algorithm is effective on only on a few matrix pairs.

3.4 Performance analysis

The popular SpGEMM libraries do not yield the best performance on all matrix pairs. On the Intel architecture, MKL delivers the best performance on approximately 35% 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% of the dataset. The "COO method" is suitable for specific cases and some matrix pairs still stand out. Furthermore, the vector-, hash-, and heap-based methods run on their "best of

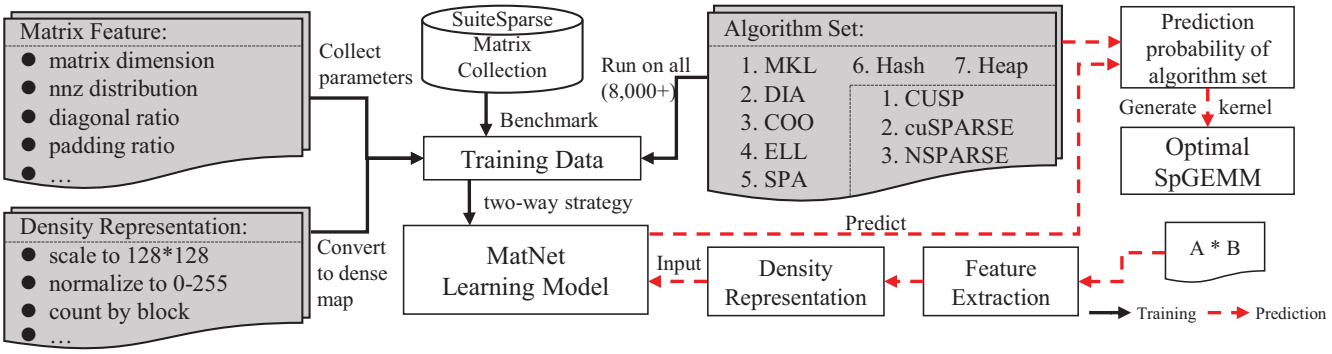

Figure 4: IA-SpGEMM overview: The solid line indicates the collection and training phase, and the dotted line is the execution flow of the user interface. The collection phase includes extracting two patterns of input and the execution times of all algorithms. The training phase generates the MatNet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

all" matrix pairs to obtain average speedups of 1.31x, 6.37x, and 6.21x, 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" method and the NSPARSE algorithm obtain average speedups of 6.27x and 3.71x, respectively on their "best of all" cases.

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 8.94x, 46.16x, and 2.40x for all matrix pairs on the three platforms. Such performance improvements urgently drive us to design an auto-tuning framework.

4 OVERVIEW OF IA-SPGEMM

In the previous section, our experimental results demonstrate the enormous potential for leveraging various formats and algorithms. We thus develop an Input-aware Auto-tuning Framework for SpGEMM (IA-SpGEMM) to select the best format and algorithm on multiple architectures. The framework is shown in Figure 4. It considers the impact on 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 problem, and select an outstanding convolutional neural network for classification to achieve this goal.

Recognizing the best format and algorithm 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 and algorithms as the output of the training data. Thus, this method incorporates matrix features and algorithms together to automatically generate a highly accurate classifier. As shown in Figure 4, the IA-SpGEMM system is divided into two parts: training and prediction. It first trains the neural network MatNet (Section 4.3) by using the collected training data, and the prediction part indicates the probability of each algorithm to generate the best SpGEMM kernel.

Conveniently, the IA-SpGEMM provides a unified interface based on the CSR format, which lends usability and portability to it. It can quickly replace libraries in the IA-SpGEMM framework. It also supports two usage methods to fit unique needs. The first is one where the framework automatically selects the optimal algorithm, whereas the other supports the inspector-executor approach. This difference brings two benefits. First, the developer can transparently benefit from multiple formats and algorithms. Second, the framework can save the best choice by automatic tuning and reuse the known best algorithm on the same matrix to significantly reduce the overhead due to feature extraction and forward propagation of the neural network. Extensibility is also an advantage of the IA-SpGEMM. 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 the IA-SpGEMM: feature extraction, density representation, and the design of the neural network.

4.1 Feature Extraction

As an automatic input-tuning system, the IA-SpGEMM first considers 13 fine-grained features related to the distribution and characteristics of four formats for CPU, and eight features of two 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 3 summarizes all the sparse features used in the IA-SpGEMM. The first eight features represent the most common structure, including the number of rows and columns, and non-zero elements, which suit for all four formats. The ninth to 11th 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 ELL format for aligned memory access. The 13th feature represents the coefficient

Table 3: 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	the ratio of padding data in DIA and ELL formats
CV	the coefficient of variation of non-zero elements

of variation (CV) of the COO format used in [1] 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 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 a 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 resize an 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.

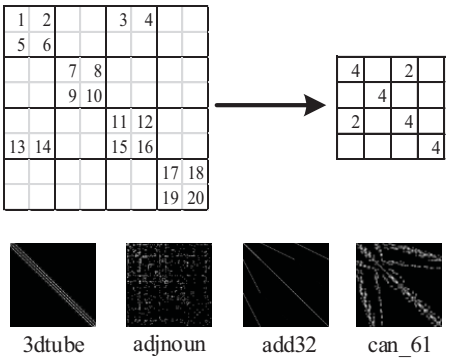

Figure 5: An example to convert 8×8 matrix to 4×4 density representation.

As shown in Figure 5, 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 ~255).

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. Note that any sampling method (such as distance histogram representation [61]) may also 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).

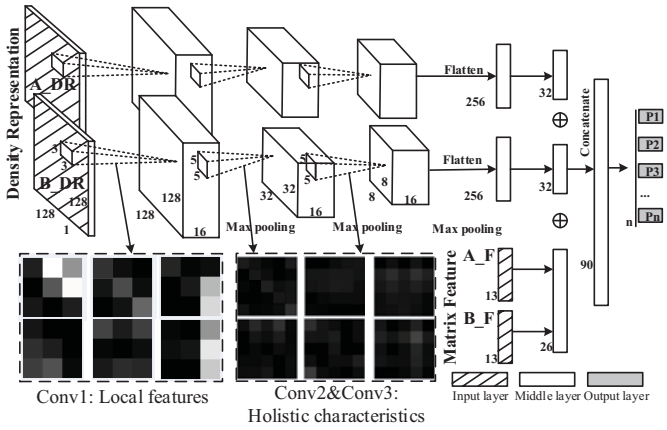

Figure 6: Details of the parameters of the neural network, and visualization of some kernels of MatNet.

4.3 MatNet Design

The traditional CNN has delivered impressive results in image classification [22, 28, 47]. Several convolutional layers and pooling layers are used in it to extract high-level features [46, 59]. The feed-forward neural network (FFNN) is applied to classify multidimensional data [56]. The standard FFNN is a multi-layer feedforward network with an input, a hidden, and an output layer. It can be used to learn and store a large number of mappings between the input and output layers. With regard to our questions, we found that these two inputs, in Section 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. We thus explore a learning model to combine the two patterns, and this is described below.

Based on the powerful classification capability of neural networks, we design the MatNet model, which combines the CNN and the FFNN to enhance the ability to classify images and features simultaneously. As show in Figure 6, this structure consists of four separate inputs, 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 features of matrices A and B, and are marked as A_F and B_F, respectively. This is so 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). The extended FFNN can aggregate sparse features.

We then define the training data, which include features (13 for CPU and nine for GPU), density representations (128*128), and the probability of each algorithm. For example, if the execution times of

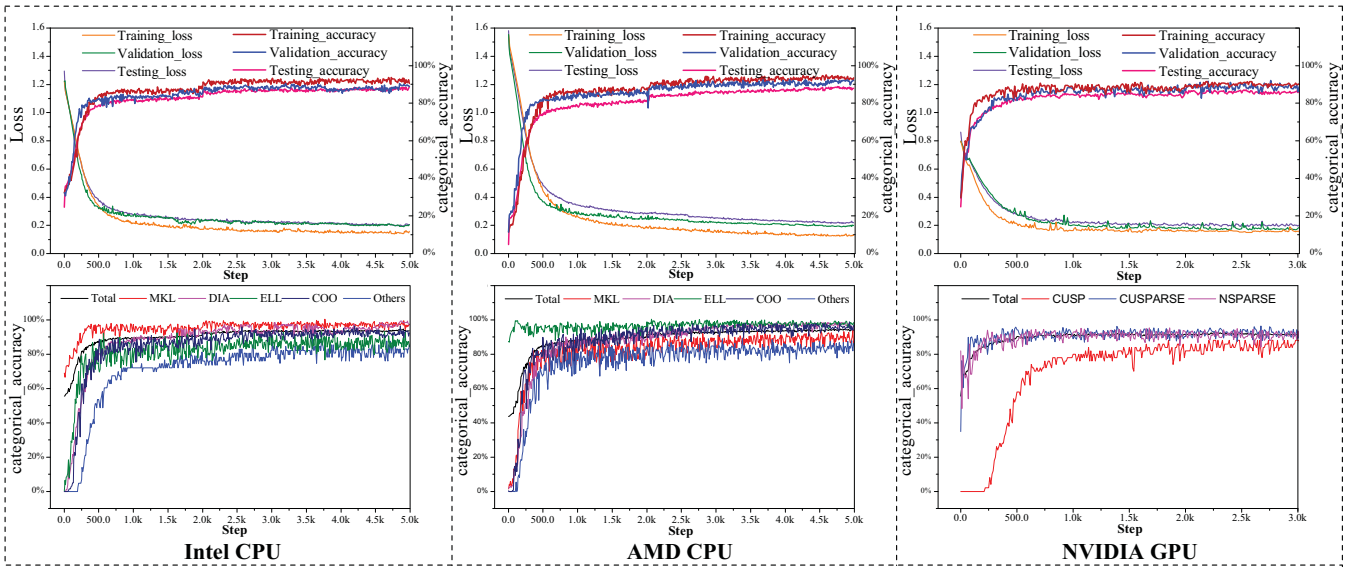

Figure 7: Loss and accuracy of the MatNet, and details of various formats and algorithms during the training phase.

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. Unlike past work on training learning models absolutely, which can cause confusion between algorithms delivering similar performance, the probability of the output fairly preserves differences that are critical for selection.

From 2,726 matrices, we randomly select 5,000 records as the training data (fewer records for GPU with limited memory). These data have two characteristics: 1) The training data are from two matrices that may be completely unrelated. 2) Similar density representations may lead to completely different results. Moreover, the relationship between the two types of input (feature and density representation) is not obvious. These unrelated data thus affect each other during the training phase, which affect the accuracy of the network, so we adopt a *two-way* strategy to eliminate interference. Therefore, training is divided into two separate stages. The first phase trains two kinds of neural networks (CNN and FFNN) 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. With the gradual addition of more training information, the accuracy of prediction can be improved step by step.

Another major advantage of this model is scalability. With the same training method, the IA-SpGEMM 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 IA-SpGEMM 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 IA-SpGEMM on three architectures, as shown in Table 4. Two of them are x86 multicore processors and the other is a manycore processor.

Baseline: The IA-SpGEMM is compared with several state-of-the-art SpGEMM libraries, such as Intel MKL v19.0.0.117 and the hash-based method [41] for CPU, and NVIDIA cuSPARSE v8.0.61 and NSPARSE [42] 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% is used for training, 20% for validation, and 20% for testing. The matrices range in size from 56KB to 33GB, and the number of non-zero elements

Table 4: 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

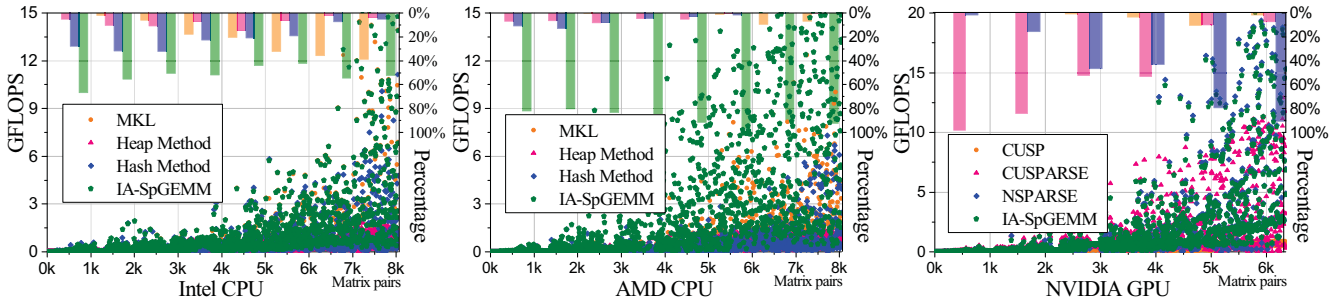

Figure 8: Performance and proportion of different formats and algorithms on the three architectures. MKL and hash-based method for CPU, and cuSPARSE and NSPARSE for GPU are state-of-the-art libraries.

ranges from 4,000 to 2 billion. The dataset includes large and actively growing sets of sparse matrices that arise in practice.

5.2 Results of Training

Figure 7 gives an overview of the loss and accuracy of MatNet discussed in Section 4.2 for classifying matrix pairs into the best format and algorithm on the three platforms. Several aspects are compared below.

5.2.1 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 Figure 7 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% accuracy. We then merge the two independent trainings and adjust the learning rate (second phase). Loss continues to decline and accuracy increases slowly, and finally the training process stabilizes at 4,000 iterations. An accuracy of 93% is achieved for the training set, and 91% and 90% 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%. For the GPU platform, MatNet incurs 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%.

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 significantly improves accuracy. With the combination of the two types of training data by using the two-way strategy, MatNet is thus further upgraded.

5.2.2 Best Format and Algorithm. We now provide details of the convergence of various formats and algorithms by MatNet. 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%.

Table 5: Comparison of results of prediction of the two machine learning classification methods.

Platform	Method	Decision Tree		MatNet	
		pre.(%)	recall(%)	pre.(%)	recall(%)
Intel CPU	MKL	79.669	86.294	88.137	96.160
	DIA	81.037	66.384	94.284	76.741
	ELL	54.731	63.157	96.039	86.607
	COO	67.105	70.223	92.531	78.636
	Others	71.875	45.098	89.201	85.201
AMD CPU	MKL	75.021	73.972	93.277	87.763
	DIA	80.174	53.336	89.512	90.319
	ELL	71.875	88.461	95.424	85.887
	COO	76.288	86.064	87.570	93.392
	Others	79.613	58.461	86.841	89.129
NVIDIA GPU	CUSP	66.327	28.571	92.5	74.326
	CUSPARSE	76.428	82.294	95.215	93.675
	NSPARSE	82.667	78.981	91.961	87.620

In addition, we used a widely used traditional decision tree algorithm (CART approach [12]) to compare with MatNet. The decision tree is constructed by features of the two matrices. Table 5 shows the performance of the two classifiers on two indicators, where pre. represents precision and recall measures the number of correct results returned. The result shows that MatNet outperforms the decision tree on the two indicators with an average precision of 91.50% and recall of 86.57%, whereas the decision tree has a precision of 74.19% and recall of 67.79%. The results in terms of accuracy on the three platforms also show that our MatNet is an effective cross-platform model.

5.3 Speedups and Overhead

The results of speedup of the IA-SpGEMM are presented in Figure 8. The predicted formats and algorithms are generated by the MatNet model, and each execution of the IA-SpGEMM includes the

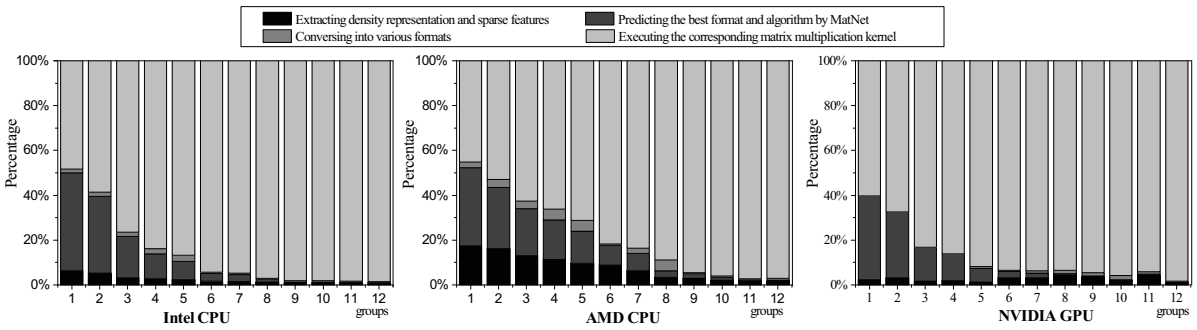

Figure 9: Performance breakdown of several groups of matrices of different sizes.

complete overhead incurred by feature extraction, prediction, and format conversion (if needed). The x-axis represents the sequence of matrix pairs 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 3.27x, 13.17x, and 2.23x. Furthermore, we test the speedups of IA-SPGEMM in comparison with state-of-the-art methods ([41] and [42]), 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 "ideal tool" mentioned in Section 3.4, the IA-SpGEMM achieves an accuracy of 94% without overhead and 37% with overhead as its best performance 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 27 minutes to train the complete MatNet for 4,900 records on two NVIDIA P100 GPUs. In addition, SpGEMM using the IA-SpGEMM 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 Figure 9, 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 the IA-SpGEMM is below 20%. 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 Usage

In this section, we open-source IA-SpGEMM with a unified interface for the SPGEMM kernel and provide a test case to compute $A * B$ for validating the results of 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 applications [14, 23, 33]. Various approaches have been proposed to optimize data dependence and unbalanced sparse computations. Venkat et al. [2, 53, 54, 60] developed several techniques for dependence analysis and data transformation optimization for sparse computations during the compilation phase, Arash et al. [3, 4] used a performance model and a blocking mechanism to resolve the problem of load imbalance. This paper focuses on the format, algorithm, and auto-tunner 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 [20, 43] is a general solution, whereas other methods involve sorting a heap [5] or merging rows [44]. Moreover, a few GPU algorithms have been proposed, CUSP [16] uses an expand-sort-compress (ESC) algorithm that pre-allocates and collects all intermediate results, and accumulates them through sort and compression operations. cuSPARSE [17], NSPARSE [42] and Kokkos [19] uses a hash table to combine the intermediate results in global memory. bhPARSE [34] first assigns rows into bins by the size of the intermediate result and output, and launches various kernels. The hybrid method [36, 38], multiple-levels algorithm [5], and row merge algorithm [24] can also show good performance on partial matrices. These algorithms can be added to our IA-SpGEMM to yield better performance. Moreover, our current system selects four main formats on the CPU and two formats (COO and CSR) on the GPU. But there are 10 popular formats [50], including Compressed Sparse Column (CSC), Sliced ELL (SELL) [40], Block CSR (BCSR) [45], Hybrid (HYB) [11] and CSR5[37]. However, this work

focuses on building an IA-SpGEMM framework compatible with various formats and algorithms. Using this framework, we can still design new algorithms for these formats to speed-up this kernel, and the SpGEMM algorithms can advance the IA-SpGEMM system. In addition, the IA-SpGEMM framework proposed in this paper resolves the problem on a single node and, in many cases, dominate the whole overhead. We would like to see that the following work could integrate our framework into distributed SpGEMM implementations[8, 9, 13].

Selecting the best format and algorithm has received considerable attention in recent years. The work closest to this study is that by Zhao et al.[61], which for the first time used a CNN to select the matrix format for SpMV and yielded an accuracy of 93%. Several studies [15, 32, 48] have been devoted to the best storage formats through auto-tuning methods, 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 handwritten digits, which was one of the first applications of the CNN. The LeNET-5 [29] was developed for this task. The FFNN [30] 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 an Input-aware Auto-tuning Framework for SpGEMM (IA-SpGEMM) that can automatically determine the best format and algorithm 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 the IA-SpGEMM 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 is supported by the National Key Research and Development Program of China (2017YFB0202105, 2016YFB0201305, 2016YFB0200803, 2016YFB0200300) and National Natural Science Foundation of China under grant no. (61521092, 91430218, 31327901, 61472395, 61432018, 61671151).

REFERENCES

[1] Walid Abu-Sufah and Asma Abdel Karim. 2013. Auto-tuning of sparse matrix-vector multiplication on graphics processors. In *International Supercomputing Conference*. Springer, 151–164.
[2] Khalid Ahmad, Anand Venkat, and Mary Hall. 2016. Optimizing LOBPCG: Sparse Matrix Loop and Data Transformations in Action. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 218–232.
[3] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P Sadayappan. 2014. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 273–282.
[4] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P Sadayappan. 2015. A model-driven blocking strategy for load balanced sparse matrix–vector multiplication on GPUs. *J. Parallel and Distrib. Comput.* 76 (2015), 3–15.
[5] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.
[6] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. 2018. HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks. *Nucleic acids research* 46, 6 (2018), e33–e33.
[7] David Bader. [n. d.]. Graph BLAS Forum. https://graphblas.org. ([n. d.]).
[8] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. 2015. Brief announcement: Hypergraph partitioning for parallel sparse matrix-matrix multiplication. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. ACM, 86–88.
[9] Grey Ballard, Christopher Siefert, and Jonathan Hu. 2016. Reducing communication costs for sparse matrix multiplication within algebraic multigrid. *SIAM Journal on Scientific Computing* 38, 3 (2016), C203–C231.
[10] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.
[11] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 18.
[12] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen. 1984. *Classification and Regression Trees*. Taylor & Francis. https://books.google.com/books?id=JwQx-WOmSyQC
[13] Aydin Buluç and John R Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.
[14] Hong Cheng, Zicheng Liu, Lu Yang, and Xuewen Chen. 2013. Sparse representation and learning in visual recognition: Theory and applications. *Signal Processing* 93, 6 (2013), 1408–1425.
[15] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM sigplan notices*, Vol. 45. ACM, 115–126.
[16] Steven Dalton, Luke Olson, and Nathan Bell. 2015. Optimizing sparse matrixâĂŤmatrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)* 41, 4 (2015), 25.
[17] Julien Demouth. 2012. Sparse matrix-matrix multiplication on the gpu. In *Proceedings of the GPU Technology Conference*, Vol. 3.
[18] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2017. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE, 693–702.
[19] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. 2018. Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures. *Parallel Comput.* 78 (2018), 33–46.
[20] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.
[21] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2008. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering* 10, 2 (2008).
[22] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 580–587.
[23] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 315–323.
[24] Felix Gremse, Andreas Hofter, Lars Ole Schwen, Fabian Kiessling, and Uwe Naumann. 2015. GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71.
[25] Fred G Gustavson. 1978. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.
[26] R Intel. 2019. *Intel math kernel library reference manual*. Technical Report. Tech. Rep.[Online]. Available: https://software.intel.com/sites/default/files/mkl-2019-developer-reference-c_0.pdf.
[27] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM*

Journal on Scientific Computing 36, 5 (2014), C401–C423.
[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[30] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. 1988. A theoretical framework for back-propagation. In *Proceedings of the 1988 connectionist models summer school*, Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann, 21–28.
[31] Ang Li, Weifeng Liu, Mads R. B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and Analyzing the Real Impact of Modern On-package Memory on HPC Scientific Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. 26:1–26:14.
[32] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 117–126.
[33] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. 2015. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 806–814.
[34] Weifeng Liu. 2015. *Parallel and Scalable Sparse Basic Linear Algebra Subprograms*. Ph.D. Dissertation. University of Copenhagen.
[35] Weifeng Liu, Ang Li, Jonathan D. Hogg, Iain S. Duff, and Brian Vinter. 2017. Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4244–n/a.
[36] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS '14)*. 370–381.
[37] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS '15)*. 339–350.
[38] Weifeng Liu and Brian Vinter. 2015. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *J. Parallel and Distrib. Comput.* 85, C (2015), 47–61.
[39] Weifeng Liu and Brian Vinter. 2015. Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors. *Parallel Comput.* 49, C (2015), 179–193.
[40] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 111–125.
[41] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. 2018. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. *arXiv preprint arXiv:1804.01698* (2018).
[42] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 101–110.
[43] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. 2015. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *International Conference on High Performance Computing*. Springer, 48–57.
[44] Karl Rupp, Florian Rudolf, and Josef Weinbub. 2010. ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. In *Intl. Workshop on GPUs and Scientific Applications*. 51–56.
[45] Youcef Saad. 1990. SPARSKIT: A basic tool kit for sparse matrix computations. (1990).
[46] Dominik Scherer, Andreas Müller, and Sven Behnke. 2010. Evaluation of pooling operations in convolutional architectures for object recognition. In *Artificial Neural Networks–ICANN 2010*. Springer, 92–101.
[47] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.
[48] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 99–108.
[49] Viral B Shah. 2007. *An interactive system for combinatorial scientific computing with an emphasis on programmer productivity*. University of California, Santa Barbara.
[50] FS Smailbegovic, Georgi N Gaydadjiev, and Stamatis Vassiliadis. 2005. Sparse matrix storage format. In *Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc*, Vol. 2005. 445–448.
[51] Bor-Yiing Su and Kurt Keutzer. 2012. clSpMV: A cross-platform OpenCL SpMV framework on GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 353–364.
[52] Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and Implementation of Adaptive SpMV Library for Multicore and Many-Core Architecture. *ACM Trans. Math. Softw.* 44, 4 (2018), 46:1–46:25.
[53] Anand Venkat, Mary Hall, and Michelle Strout. 2015. Loop and data transformations for sparse matrix code. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 521–532.
[54] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 41.
[55] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 338–353.
[56] Liyang Wei, Yongyi Yang, Robert M Nishikawa, and Yulei Jiang. 2005. A study on several machine-learning methods for classification of malignant and benign clustered microcalcifications. *IEEE transactions on medical imaging* 24, 3 (2005), 371–380.
[57] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: yet another SpMV framework on GPUs. In *PPOPP*.
[58] Raphael Yuster and Uri Zwick. 2004. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 254–260.
[59] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.
[60] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the gpu microarchitecture to achieve bare-metal performance tuning. *ACM SIGPLAN Notices* 52, 8 (2017), 31–43.
[61] Yue Zhao, Chunhua Liao, Jiajia Li, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 94–108.