Journal of Parallel and Distributed Computing 185 (2024) 104799

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

Revisiting thread configuration of SpMV kernels on GPU: A machine learning based approach

Jianhua Gao, Weixing Ji *, Jie Liu, Yizhuo Wang, Feng Shi

School of Computer Science and Technology, Beijing Institute of Technology, Beijing, 100081, China

ARTICLE INFO

Keywords:
SpMV
Machine learning
GPU
Thread configuration

ABSTRACT

Sparse matrix-vector multiplication (SpMV) optimization on GPUs has been challenging due to irregular memory accesses and unbalanced workloads. The majority of existing solutions assign a fixed number of threads to one or more rows of sparse matrices according to empirical formulas. However, this method does not give the optimal thread configuration and results in a significant performance loss. This paper proposes a new machine learning-based thread assignment strategy for SpMV on GPU, predicting the near-optimal thread configuration for matrices. Further, we partition irregular sparse matrices into blocks according to the distribution of non-zero elements and predict the optimal thread configuration for each block. A new SpMV kernel is designed to accelerate the execution of different blocks. Experimental results show that our machine learning-based approach can select the near-optimal thread configuration for most matrices. The efficiency of SpMV for irregular matrices is also improved by matrix partitioning and blockwise prediction. Finally, we dive into the trained model to find out the connection between the features of a sparse matrix and its optimal thread configuration.

1. Introduction

Analysis and design problems of practical nature in many fields, such as structural analysis [25], social science [32], and electric circuit design [18], can be transformed into sparse linear systems. Iterative solvers, such as conjugate gradient (CG) and generalized minimal residual method (GMRES), are the most common and popular methods for solving these systems. They heavily rely on SpMV, which accounts for 80% of the total time in solving linear systems [10]. Hence, it is necessary to develop highly efficient SpMV algorithms to reduce the execution time of existing solvers and improve the overall performance.

Graphics Processing Units (GPUs) have emerged as promising computing devices due to their massive parallelism and high memory bandwidth. However, the matrices generated in practice tend to be very sparse. A cell in the netlist of circuits is usually connected to a few others. Thus the number of non-zero elements (NNZ) in each row is usually negligible. There are a few nodes that have hundreds of fan-outs, such as power source and clock input [18]. Irregular memory accesses and unbalanced work among rows pose substantial challenges for SpMV acceleration on GPUs.

The current researches show that the performance of SpMV algorithms is susceptible to the distribution of non-zero elements in input matrices [12,15,27]. No sparse compression format can give the highest performance across all sparse matrices and devices. Therefore, several format selection methods have been developed in recent years [4,12,29,34,38,41,42]. However, compressed sparse row (CSR) is still the most popular compression format and is widely used by many applications. Converting CSR to a new format introduces significant overhead, which is non-negligible in practice [42]. To address the issue of unbalanced workloads, researchers have developed several new algorithms based on the CSR for GPU to improve the performance of SpMV [11][8][36][23][15]. Since the NNZ in different rows may vary substantially, it is difficult to find an efficient parallel strategy to fully utilize the underlying hardware's computing capacity. Initially, one thread is assigned to a matrix row in parallel computing of SpMV [11]. However, this is inefficient for processing long rows, as threads in a GPU warp are running in single instruction multiple threads (SIMT) paradigms, and other threads in the same warp assigned to short rows have to wait for the threads processing long rows. The CSR-vector algorithm, assigning several threads to process each row, was proposed so that long rows can be processed much faster than a single thread [11]. One of the key issues in this design is determining the number of threads assigned to one row in SpMV implementation. Existing implementations of CSR-vector either set a fixed number of threads [8][11], or calculate

* Corresponding author.
E-mail addresses: gjh@bit.edu.cn (J. Gao), jwx@bit.edu.cn (W. Ji), jieliu@bit.edu.cn (J. Liu), frankwyz@bit.edu.cn (Y. Wang), bitsf@bit.edu.cn (F. Shi).

https://doi.org/10.1016/j.jpdc.2023.104799
Received 30 November 2022; Received in revised form 13 October 2023; Accepted 1 November 2023
Available online 10 November 2023

the number of threads based on empirical formulas [16][36]. In the latest implementation of the CSR-vector from CUSP [16], the number of threads is based on the average NNZ per row of the input matrix. Besides, a similar rule was proposed by Reguly et al. [36]. However, our experiments with more than 2,000 sparse matrices show that there is still a significant performance gap between existing thread configurations and the optimal one.

Based on the above observations, in this paper, we propose a machine learning (ML) based approach to predict the optimal thread configuration (the number of threads assigned to rows). We use simple and easy-to-calculate matrix features as the input of the classification model and train and test the model using more than 3,000 matrices. Analysis shows that our model has over 80% test accuracy and achieves near-optimal results for most matrices. Furthermore, irregular matrices are virtually partitioned into several sub-matrix blocks according to their distribution of non-zero elements in different parts, and the corresponding optimal thread configuration for each block is predicted individually and independently. The partition of an irregular matrix generates multiple locally-regular blocks, making it possible to choose the optimal thread configuration for each block.

Our main contributions are as follows:

- Present a machine learning-based method to predict the optimal thread configuration for each sparse matrix. The experimental results show that the predicting model achieves an accuracy of higher than 80% on two different GPU devices. Moreover, the predicted thread configuration is exceptionally close to optimal performance, with an arithmetic mean speedup of 1.21x and 1.24x on the two machines against the CSR-vector using 16 threads for each row.
- Provide a non-zeros distribution-aware matrix partitioning method and design a new SpMV kernel, which uses a blockwise ML-based thread configuration approach. The experimental results show that the performance for irregular matrices is significantly improved by matrix partition and blockwise prediction. It achieves an arithmetic mean speedup of 3.00x and 2.67x on two GPUs.
- Dive deep into the trained ML models to find out the connections between the features of a sparse matrix and its optimal thread configuration. Moreover, we derived a lightweight inference model that costs only a few microseconds for predicting in practical use.

The rest of this paper is organized as follows. Section 2 lays out the background of the research and the importance of thread configuration prediction in SpMV calculation. Section 3 presents a coarse-grained global prediction strategy for thread configuration of SpMV on GPU according to the features of sparse input matrices. Section 4 proposes a blockwise partition and thread configuration prediction method. A new SpMV kernel is also presented based on our partition and prediction system. The evaluation results are given in Section 5. Section 6 gives a deep discussion on the model and overhead analysis. In Section 7, we discuss related work on SpMV optimization. Finally, the conclusion is given in Section 8.

2. Problem statement

2.1. Revisiting the CSR-vector algorithm

CSR-vector assigns a group of cooperative threads *TpR* (Threads per Row) to process each matrix row. It limits *TpR* to 2, 4, 8, 16, or 32 to take advantage of the synchronous execution of threads in a warp. These cooperative threads first access the non-zero elements in the same row and the corresponding elements of the multiplied vector to perform multiplications. Then, the logarithmic parallel reductions are used to generate the output. Finally, the first of every TpR threads writes the result back to global memory. CSR-vector is more efficient for memory access and thread parallelism than CSR-scalar, which assigns one

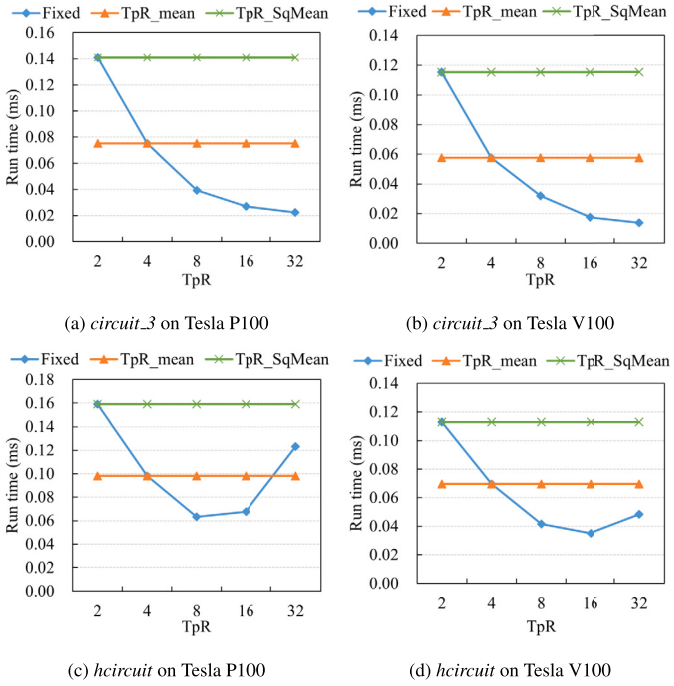


Fig. 1. Runtime comparison of CSR-vector with different *TpR* for two matrices *circuit_3* and *hcircuit* on two GPUs.

thread to each row. However, too small *TpR* in CSR-vector is of little benefit to the access locality and may incur a significant performance loss similar to CSR-scalar for some irregular matrices. Similarly, too large *TpR* may cause some threads to be idle, resulting in a waste of computing resources. Therefore, the setting of *TpR* deserves careful consideration.

Bell et al. [11] first proposed CSR-vector, with a *TpR* value of 32. The latest CUSP library [16] provides an implementation of CSR-vector where *TpR* is computed as,

$$TpR\_mean = \min(2^{\lceil \log_2^{\lfloor \frac{nnz}{m} \rfloor} \rceil}, 32), \tag{1}$$

where m and nnz represent the number of rows and non-zero elements of a sparse matrix. Besides, Reguly et al. [36] suggested an empirical formula for *TpR* as

$$TpR\_SqMean = \min(2^{\lceil \log_2^{\lfloor \sqrt{\frac{nnz}{m}} \rfloor} \rceil}, 32). \tag{2}$$

We tested the performance of CSR-vector with different *TpR* on two GPU platforms (Tesla V100 and P100) for two sparse matrices (*circuit_3* and *hcircuit*) from the SuiteSparse Matrix Collection [17]. Figs. 1(a) and (b) show the experimental results for *circuit_3*, where the fastest *TpR* is 32 on both machines. The TpR_mean and TpR_SqMean pick up the worst and the second worst *TpR* on two machines, respectively. For matrix *hcircuit*, as shown in Figs. 1(c) and (d), the fastest *TpR* is 16 and 8 on Tesla V100 and P100, respectively.

The above observation indicates that, on the one hand, there is still a significant performance gap between existing *TpR* settings and the optimal one. On the other hand, the optimal *TpR* of a matrix may vary with GPUs, but TpR_mean or TpR_SqMean pick up the same TpR. Therefore, given an input matrix, finding a strategy to derive the optimal or near-optimal *TpR* on a given GPU platform, has the potential to further improve the performance of CSR-vector.

2.2. Revisiting the CSR-adaptive algorithm

Although CSR-vector is usually superior to CSR-scalar, it does not consider the uneven distribution of non-zero elements in different rows,

Table 1
Different thread configuration examples of sparse matrices.

Matrix	Min	Max	TpR_mean	TpR_SqMean	Optimal
mk12-b2	3	3	4	2	2
rdb450l	4	6	8	2	8
n2c6-b7	8	8	8	2	4
hcircuit	1	1,399	4	2	16

thus giving a poor performance for some irregular matrices. CSR-adaptive [15] was designed to handle the unbalanced load in SpMV. First, a matrix is partitioned into sub-matrix blocks containing one or more rows, and the NNZ in each sub-matrix block is roughly the same. Like CSR-vector, CSR-adaptive assigns a fixed number of threads (512, for instance) to each sub-matrix block. CSR-adaptive significantly improves the computational efficiency of the sparse matrices with long rows. However, since the non-zero elements in a sub-matrix block are processed element-by-element by all threads in one thread block, and each thread performs partial computations of one row, all threads in the thread block use shared memory instead of registers to cache the intermediate results. Once all the threads have finished the element-wise multiplication and thread block synchronization, one thread is assigned to reduce the partial results of each row and write the final result back to global memory. Hence, several shared memory reads and writes are performed to compute both long and short rows. Moreover, a moderate number of threads are idle in the reduction stage when the number of rows in a sub-matrix block is much less than the number of threads in a thread block. For example, the sparse matrix *iprob* is partitioned into 25 blocks by CSR-adaptive. Except for the block containing one long row, each block has 128 rows. When the thread block size is fixed to 256, half of the threads in each thread block are idle in the reduction phase.

In summary, CSR-adaptive achieves a balanced load among thread blocks at the cost of additional memory access and synchronization overhead. Besides, a moderate number of threads may be idle in the row-by-row reduction stage. Therefore, a strategy that combines CSR-vector and CSR-adaptive is more promising in improving the overall performance.

3. Global thread configuration prediction

3.1. Methodology overview

Finding the optimal *TpR* setting for a sparse matrix is challenging. The main reason for this is the complex and diverse distribution of non-zero elements in different rows. Table 1 presents four examples of sparse matrices. The NNZ in each row of the matrix *mk12-b2* is 3, the configuration TpR_SqMean picks up the optimal *TpR* setting of 2, and the strategy TpR_mean derives the near-optimal *TpR* setting of 4. The NNZ in each row of the matrix *rdb450l* falls within the interval [4,6], and these two thread configurations reach different results. For the matrix *n2c6-b7* with 8 non-zero elements in each row, both configurations fail to pick up the optimal *TpR* setting. For matrices with a large difference in NNZ per row, the *TpR* given by these configurations may cause significant performance loss. For example, the shortest row of the matrix *hcircuit* has only one non-zero element, and its longest row has 1,399 non-zero elements, but its average NNZ per row is only about 4. TpR_mean and TpR_SqMean give a thread configuration where the calculation of long rows is a performance bottleneck, far inferior to the optimal one.

The above observations indicate that the optimal thread configuration is related to not only the NNZ per row but also some other features. Moreover, the relationship between matrix features and the optimal thread configuration is not straightforward, and it is difficult to model the relationship using formulas. Therefore, this paper proposes a prediction strategy for thread configuration based on machine learning techniques. Fig. 2 presents the overview of our approach. We first

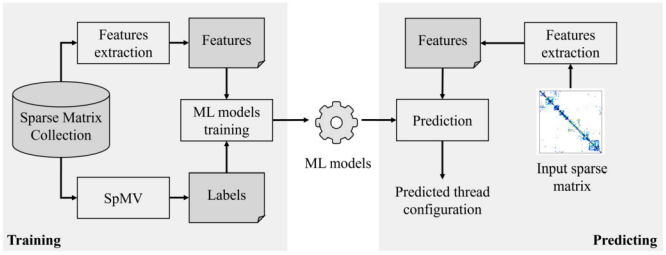
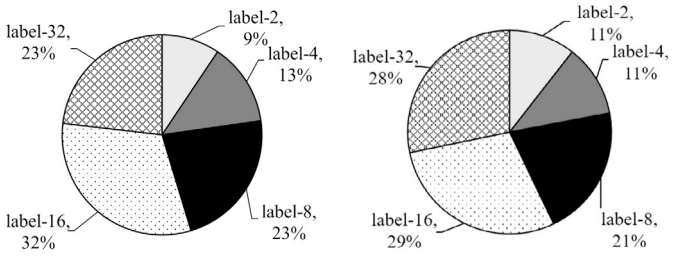


Fig. 2. Training and predicting the optimal *TpR* setting using an AutoML framework.



(a) Tesla P100 (b) Tesla V100

Fig. 3. The distribution of matrices with different labels.

selected some features of sparse matrices, and each matrix is labeled with the optimal *TpR*. Then, these labeled matrices are fed into different ML models for training. Next, the model achieving the highest accuracy in the testing set is selected. Finally, given an input sparse matrix, the trained ML model predicts the optimal TpR setting based on its features.

3.2. Dataset and feature set

The training and testing sets are from the SuiteSparse Matrix Collection [17], which is a prevalent sparse matrix dataset and widely used in existing works on the optimization of sparse matrix computation. The collection has 2,893 sparse matrices, with 1,020 asymmetric sparse matrices. Except for some very large or complex-type sparse matrices, 2,833 sparse matrices and the transpositions of 978 asymmetric sparse matrices constitute the final dataset. Adding the transpositions of the asymmetric matrices increases the size of the dataset, thus improving the model accuracy.

To find the optimal *TpR* setting for each matrix, we run CSR-vector for the entire dataset with all five thread configurations, including the *TpR* of 2, 4, 8, 16, and 32. We record the average run time of 100 SpMV iterations for each matrix. Each matrix is labeled with the *TpR* corresponding to the shortest run time. Fig. 3 summarizes the distribution of matrices with different labels.

Table 2 summarizes our feature set used in model training and testing. The rightmost column shows that all features are easy to compute for a given input matrix. As part of the pre-processing overhead, the calculation of matrix features must be as simple as possible so as not to offset the benefits from the optimal *TpR* setting.

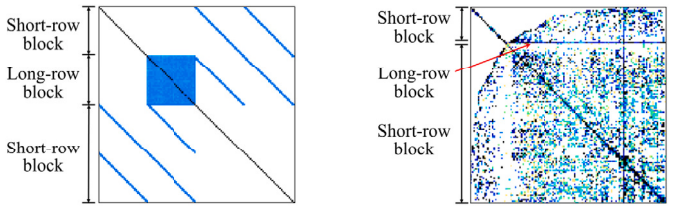
3.3. Model training

The popularization of ML techniques in different domains has encouraged us to explore their application to our problem. The biggest challenge in using ML is building a model and tuning numerous parameters suitable for a specific problem. AutoML [24] can automatically build ML models and tune parameters, thus simplifying the workflow of AI research.

AutoGluon [21] is an open-source implementation of AutoML from Amazon, focusing on automated stack assembling, deep learning, and

Table 2
The matrix features used for model training.

Feature	Description	Complexity
m	The number of rows	$O(1)$
n	The number of columns	$O(1)$
nnz	The number of non-zero elements	$O(1)$
d	The density of non-zero elements	$O(1)$
min	The minimum number of non-zero elements per row	$O(n)$
max	The maximum number of non-zero elements per row	$O(n)$
$mean$	The average of non-zero elements per row	$O(1)$
var	The variance on non-zero elements per row	$O(2n)$
max_mu	The difference between max and $mean$	$O(1)$
$SqMean$	The square root of $mean$	$O(1)$
cov	The covariance on non-zero elements per row	$O(1)$



(a) *exdata_1* (b) *shermanACa*

Fig. 4. Thumbnails of two example matrices.

一整个震惊到我了

real-world applications. AutoGluon assembles multiple models and combines them to form a new hybrid model with multiple layers. We use AutoGluon to find an optimal model that can fit our dataset.

Training: We prepare datasets for each GPU, in which all selected matrices are represented as data points with eleven features. Each point is labeled with the *TpR* that achieves the best performance on a GPU platform. Then, we shuffle all data points in the dataset and randomly select 75% as the training set and the remaining 25% as the testing set.

Testing: The testing set is used to evaluate the performance of the trained model. The same testing set is used for both machines to observe performance improvement.

Various ML algorithms can be used to solve multi-class classification problems, but finding the best algorithm is time-consuming and cost-intensive. Therefore, we use an AutoML tool to simultaneously run our dataset against combinations of many different algorithms to find the optimal one.

4. Blockwise thread configuration prediction

4.1. Overview

Fig. 4 shows thumbnails of two sparse matrices. Both matrices have dozens of long rows and many short rows. Specifically, for the matrix *exdata_1*, the NNZ of each long row is about 1,500, and that of each short row is around 3. The matrix *shermanACa* has around 2,500 non-zero elements in each long row and 6 non-zero elements in each short row. If the number of threads assigned to each row is too small, the long row will inevitably become the bottleneck of SpMV calculation. However, when many threads are assigned to each row, there will be several idle threads during the calculation of short rows. CSR-vector takes a compromise between long and short rows, but this is not optimal for both long and short rows. CSR-adaptive assigns more threads to process long rows than CSR-vector, but it introduces more shared memory accesses and synchronization operations in short rows multiplication. Therefore, the algorithm may still be improved to overcome this inefficiency.

Given this challenge, we propose partitioning the input sparse matrix according to the distribution of non-zero elements in different rows and then predicting the optimal thread configuration for different blocks. We use the aforementioned ML-based method to predict the optimal thread configuration for each short-row block.

Algorithm 1: Partition algorithm.

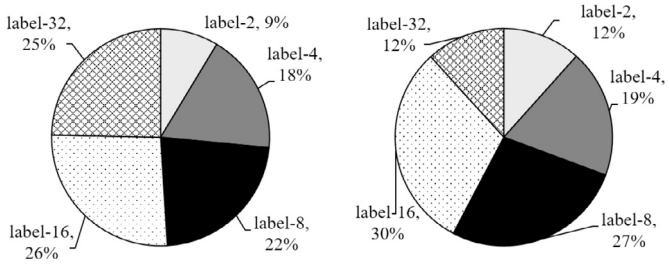
```
Input: M, nnz, csrRow
Output: nBlk, rowBlk, nnzBlk, isLong.
1  nBlk ← 0; sumNnz ← 0; rowBlk[0] ← 0;
2  if csrRow[1] − csrRow[0] ≥ T_LONG then
3  |   lastIsLong ← 1;
4  else
5  |   lastIsLong ← 0;
6  end
7  for r ← 0 to M do
8  |   curNnz ← csrRow[r + 1] − csrRow[r];
9  |   sumNnz ← sumNnz + curNnz;
10 |   if (curNnz ≥ T_LONG)&&(!lastIsLong) then
11 |   |   rowBlk[nBlk + 1] ← r;
12 |   |   nnzBlk[nBlk] ← sumNnz − curNnz;
13 |   |   sumNnz ← curNnz;
14 |   |   nBlk ← nBlk + 1; lastIsLong ← 1;
15 |   end
16 |   if (curNnz < T_LONG)&&(lastIsLong) then
17 |   |   rowBlk[nBlk + 1] ← r;
18 |   |   nnzBlk[nBlk] ← sumNnz − curNnz;
19 |   |   isLong ← isLong + pow(2, nBlk);
20 |   |   sumNnz ← curNnz;
21 |   |   nBlk ← nBlk + 1; lastIsLong ← 0;
22 |   end
23 end
24 rowBlk[nBlk + 1] ← M; nnzBlk[nBlk] ← sumNnz;
25 if lastIsLong then
26 |   isLong ← isLong + pow(2, nBlk);
27 end
28 nBlk ← nBlk + 1;
```

4.2. Matrix partition

Since we use the CSR format for all blocks, our partition is performed virtually and just needs to scan the row pointer array of CSR once. Hence, there is no format conversion and data re-organization in our design. The detailed partition algorithm is presented in Algorithm 1. We use the scalar $nBlk$ to denote the number of partitioned blocks. $rowBlk$ and $nnzBlk$ are two arrays that store the row index and the NNZ of each block, respectively. We use a bit vector $isLong$ to distinguish between long-row and short-row blocks. Considering the matrix *shermanACa* presented in Fig. 4 as an example, it is partitioned into three blocks. $isLong = (2)_{10} = (010)_2$ denotes that the first and third blocks are two short-row blocks, and the second one is a long-row block. If the current row is long, and the last row is short, then it is time to start a new block (lines 10-15), and vice versa (lines 16-22). In this way, consecutive rows with similar non-zero element distributions are packaged into one block.

4.3. Thread configuration prediction

Our training dataset is generated by slicing regular sparse matrices. The reason is that the partitioned blocks from Algorithm 1 are more regular than the original ones with long rows. To find the optimal thread configuration for the sliced blocks, we run the SpMV multiple times with different TpR. Finally, we generated a training set including 4,772 data samples, sliced from 2,386 regular sparse matrices. Fig. 5 summarizes the distribution of the optimal thread configurations in the training dataset. Our testing dataset contains irregular sparse matrices and has 326 short-row blocks partitioned from 221 sparse matrices. In addition, the same matrix feature set for global thread configuration prediction is used. We also use AutoML to find the best-fit model.



(a) Tesla V100 (b) Tesla P100

Fig. 5. The distribution of the optimal thread configurations.

```
__global__ void spmv_kernel(nBlk, isLong, rowBlk[], TpRBlk[],
      nThdBlk[], ...)
{
  __shared__ double sdata[T_LONG];
  extern __shared__ double ptrs[][2];
  for (i = 0; i < nBlk; i++) {
    thdBlkPtr_0 = nThdBlk[i];
    thdBlkPtr_1 = nThdBlk[i+1];
    nThdBlks = thdBlkPtr_1 - thdBlkPtr_0;
    rStart = rowBlk[i];
    rStop = rowBlk[i+1];
    TpR = TpRBlk[i];
    if (blockIdx.x >= thdBlkPtr_0 && blockIdx.x < thdBlkPtr_1
        ) {
      if ((isLong && 1) == 0) { /* short-row block */
        thdId = blockDim.x * nThdBlks + threadIdx.x;
        thdLane = threadIdx.x & (TpR - 1);
        vecId = thdId / TpR;
        vecLane = threadIdx.x / TpR;
        nVec = (blockDim.x / TpR) * nThdBlks;
        for (r = rStart + vecLane; r < rStop; r += nVec)
        // ...
      } else { /* long-row block */
        vecLane = (blockIdx.x - thdBlkPtr_0);
        for (r = rStart + vecLane; r < rStop; r += nThdBlks)
        // ...
      }
    }
    isLong = isLong >> 1;
  }
}
```

Listing 1: The pseudocode of the new SpMV kernel.

4.4. SpMV kernel optimization

We also designed a new SpMV kernel based on CSR-vector and CSR-adaptive to improve performance. After the partition, the distribution of non-zero elements in each block is regular. Therefore, our main idea is to calculate short-row blocks using CSR-vector, which assigns a fixed number of threads to each row and caches intermediate results in registers. Moreover, the long-row blocks are calculated using CSR-adaptive, which assigns one thread block to a row and caches intermediate results in the shared memory. In this way, the workloads in each block are balanced, and the long rows will not become a performance bottleneck.

Our new SpMV kernel is given in Listing 1. The array $\boldsymbol{nThdBlk}$ stores the starting and stopping thread block indices assigned to each matrix block. It is calculated before launching the kernel according to the NNZ in each matrix block and the total number of thread blocks. Specifically, if the block i is a long-row block, then $\boldsymbol{nThdBlk}[i]$ equals $\boldsymbol{rowBlk}[i+1]$-$\boldsymbol{rowBlk}[i]$. Otherwise, $\boldsymbol{nThdBlk}[i]$ is computed using Equation (3), where N_BLKS is the total number of thread blocks, N_BLKS_l is the number of thread blocks assigned to long-row blocks, and nnz_l is the total NNZ in long-row blocks.

$$\boldsymbol{nThdBlk}[i] = max(\lfloor \frac{\boldsymbol{nnzBlk}[i] \times (N\_BLKS - N\_BLKS_l)}{nnz - nnz_l} \rfloor, 1) \tag{3}$$

The main difference between our approach and CSR-vector is that we predict the optimal TpR using ML algorithms instead of empirical formulas. Moreover, our method considers the NNZ distribution and assigns a different number of threads to blocks. Compared with CSR-adaptive, we use a fixed number of threads to process each row in short-row blocks, rather than element-wise calculation by all threads in a thread block. It keeps intermediate results in registers and thus avoids a large number of shared memory reads and writes, and block-level synchronization.

5. Performance evaluation

5.1. Setup

All following experiments are conducted on two machines equipped with different GPUs. Table 3 lists the hardware and software setup of these machines, including CPU, host memory, GPU, device memory, OS, compiler, and related libraries. Also, compared algorithms or called interfaces are provided in the Table. The evaluated floating-point precision in the following experiments is double precision.

5.2. Global thread configuration prediction

5.2.1. Model accuracy

Our testing set contains 953 sparse matrices that are randomly selected from the entire dataset presented in Section 3.2 and have not been seen by the trained ML model. Accuracy is calculated as the percentage of correctly predicted matrices in the testing set. We shuffled our dataset four times, and four groups of training and testing sets are generated. Table 4 summarizes the test accuracy on two machines, and the models that achieve the highest test accuracy given by AutoGluon are also listed. As presented in the table, the highest test accuracy among four shuffles on Tesla V100 and P100 is about 85% and 94%, respectively.

5.2.2. Performance comparison

For some matrices, although the optimal TpR is different from the predicted TpR, their corresponding performance may be the same or extremely close. The ultimate goal of predicting TpR is to improve the performance of SpMV. Therefore, we pay attention to the performance gained from the trained ML model in this section. We refer to two metrics, Performance Loss Under Best (PLUB) and Performance Gain Over (PGO), used in [7,14,19] to evaluate the performance of the ML model. Let n represent the number of sparse matrices in the testing set, t_p be the execution time of CSR-vector with predicted TpR, and t_s be the shortest execution time with all possible TpR. The calculation of PLUB is given as

$$PLUB(\%) = 100 \times \frac{1}{n} \sum_{i=1}^{n} \frac{t_p^i - t_s^i}{t_s^i}. \tag{4}$$

Let t_k represent the execution time with TpR of k, then the performance gain of the predicted TpR over k, abbreviated as $PGO(k)$, is calculated as,

$$PGO(k)(\%) = 100 \times \frac{1}{n} \sum_{i=1}^{n} \frac{t_k^i - t_p^i}{t_p^i}. \tag{5}$$

Table 4 shows the performance results collected on two machines. We make the following observations:

- The PLUB of the ML-based method has a maximum of 1.20% on two machines, which means that the CSR-vector with the ML-predicted thread configuration achieves close to 99% of maximum performance.

Table 3
Hardware (HW.) and software (SW.)

		Platform 1	Platform 2
HW.	Host	**CPU**: Intel Core i9-10980XE, 3.0 GHz, 18 cores **Mem.**: 128 GB	**CPU**: Intel Xeon E5-2680 v4, 2.5 GHz, 14 cores **Mem.**: 128 GB
	Device	**GPU**: Tesla P100, 1.33 GHz, 3,584 cores **Mem.**: 16 GB	**GPU**: Tesla V100-SXM2, 1.53 GHz, 5,120 cores **Mem.**: 16 GB
SW.	OS	64-bit Ubuntu 18.04	64-bit Ubuntu 18.04
	Compiler	**nvcc**:11.2.6; **gcc/g++**: 8.4.0	**nvcc**:11.2.67; **gcc/g++**: 7.5.0
	Library	**cuSPARSE**[35]:11.2, *cusparseSpMV* with algorithm *CUSPARSE_MV_ALG_DEFAULT*; **Ginkgo**[3]:1.3.0, CSR-based SpMV with option *-formats = "csr" -repetitions = 100 -warmup = 1 -gpu-timer = true*; **Merge**[30][31] ; **CUSP**[16]:0.5.1;	

Table 4
Accuracy (Acc.) and overall performance of ML for global thread configuration prediction on two machines.

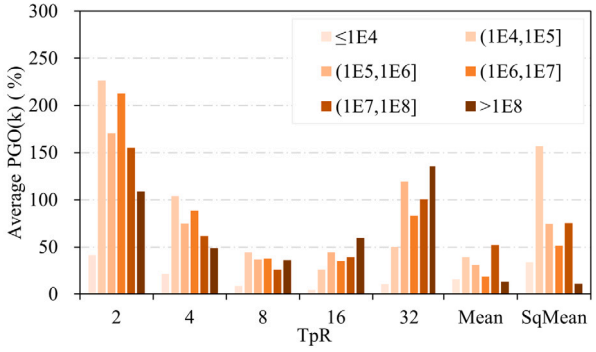
GPU	Shuffle	Optimal model	Acc.(%)	PLUB(%)	PGO(k) (%)						
					2	4	8	16	32	TpR_mean	TpR_SqMean
V100	1	LightGBM	84.26	0.56	160.71	70.39	30.27	20.91	42.34	24.44	85.78
	2	NeuralNetTorch	82.90	0.53	168.01	72.89	30.24	18.51	36.91	25.89	99.10
	3	LightGBM	82.48	0.57	149.35	64.72	27.65	19.40	41.34	27.82	94.29
	4	XGBoost	85.41	0.45	155.08	67.43	28.32	18.47	37.82	24.70	87.60
P100	1	NeuralNetMXNet	80.80	1.20	151.33	67.93	29.98	25.37	60.16	27.46	81.91
	2	LightGBM	83.74	1.09	143.84	63.56	26.82	21.57	53.08	30.02	84.95
	3	XGBoost	84.99	0.82	161.23	72.69	30.96	24.26	57.30	28.96	99.36
	4	ExtraTreesGini	94.44	0.32	144.70	64.22	27.42	22.36	53.95	30.67	85.61

Table 5
Average PLUB on matrices with different NNZ distributions.

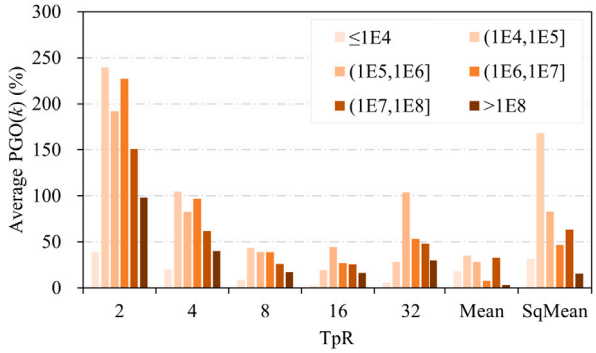
NNZ	PLUB on Tesla P100 (%)			PLUB on Tesla V100 (%)		
	Mean	SqMean	ML	Mean	SqMean	ML
≤1E4	15.53	33.79	**1.50**	18.51	32.02	**0.55**
(1E4,1E5]	39.10	156.87	**0.97**	35.46	168.98	**0.43**
(1E5,1E6]	30.84	74.37	**1.47**	29.36	83.96	**1.00**
(1E6,1E7]	18.54	51.18	**0.84**	8.01	47.22	**0.12**
(1E7,1E8]	52.06	75.36	**0.39**	39.10	73.25	**0.67**
>1E8	13.11	11.12	**0.49**	3.06	15.69	**0.02**

- Instead of just using a fixed *TpR*, or an adaptive *TpR* based on empirical formulas, CSR-vector using ML predicted thread configuration achieves an average performance improvement ranging from 18% to 168%.

In the following evaluation, we take the testing sets in the first shuffle for discussion. Fig. 6 plots the average performance gain of ML-based thread configuration over other configurations on sparse matrices with different NNZ distributions. It can be observed that, for the first five fixed rules (*TpR* = 2, 4, 8, 16, or 32), the performance gains of ML-based thread configuration on sparse matrices with moderate NNZ (falling within the interval (1E4,1E8]) are greater than those on extremely small (≤1E4) or extremely large matrices (>1E8). ML-based thread configuration brings higher performance gain over TpR of 16, 32, and *mean* on Tesla P100 than on Tesla V100. The reason is that Tesla V100 provides more computing cores and hardware architectural improvements like independent thread scheduling. As a result, the execution time of SpMV and the performance difference caused by different thread configurations are smaller on the Tesla V100 compared with the Tesla P100. According to Equation (5), we can anticipate a decrease in PGO on the Tesla V100 compared with the Tesla P100. Besides, Table 5 summarizes the average PLUB of three thread configurations on matrices with different NNZ distributions. It can be observed that the PLUB change of TpR_mean and TpR_SqMean is similar to the changing trend of their PGO. Compared with the optimal configuration, the ML-based thread configuration incurs a fairly small performance loss.



(a) Tesla P100



(b) Tesla V100

Fig. 6. Average PGO(k) on matrices with different NNZ distributions. "(1E4,1E5]" represents the average PGO(k) on matrices with NNZ more than 10,000, and less than and equal to 100,000.

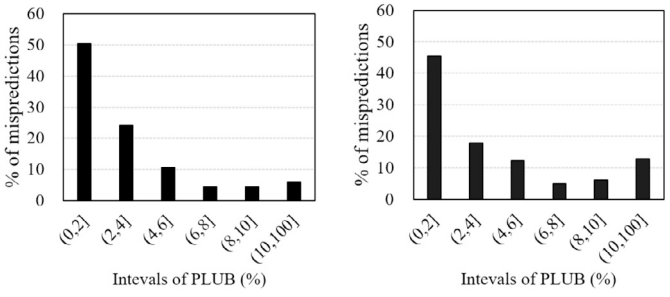
In the first shuffle, the ML model's accuracy does not exceed 85% on both machines, but its PLUB is only about 1%. Fig. 7 gives the reason, and it presents the distribution of mispredictions in different PLUB intervals. It can be observed that, on both machines, most mispredic-

Table 6
Average speedup over CSR-vector with *TpR* of 16 on two machines for the testing set. *AM* and *GM* refer to arithmetic and geometric mean speedups, respectively.

GPU	Metrics	Ginkgo	ViennaCL	cuSPARSE	Merge	ML	Optimal
V100	AM	0.37	0.87	1.17	**1.26**	1.21	1.21
	GM	0.20	0.70	1.10	0.74	1.17	**1.18**
P100	AM	0.72	0.62	1.11	0.88	1.24	**1.25**
	GM	0.32	0.42	1.01	0.44	1.19	**1.20**

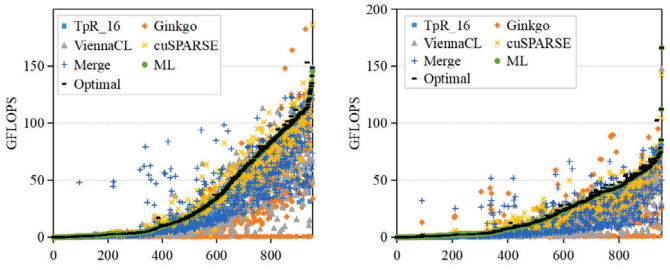
Table 7
The number of sparse matrices achieving the best SpMV performance for each algorithm.

GPU	TpR_16	Ginkgo	ViennaCL	cuSPARSE	Merge	ML	Optimal
V100	161	31	71	259	68	457	**525**
P100	248	54	23	128	45	563	**703**



(a) Tesla V100 (b) Tesla P100

Fig. 7. Distribution of mispredictions for global thread configuration in different PLUB intervals.



(a) Tesla V100 (b) Tesla P100

Fig. 8. Performance comparison of seven SpMV algorithms on two machines for the testing set.

tions fall within the interval $(0, 2\%]$, a small number of ones fall within $(2\%, 6\%]$, while only the PLUB on a few sparse matrices is greater than 6%. Although the ML model makes wrong decisions for close to 16% and 20% matrices on two machines, the predicted not-optimal thread configuration still achieves a performance close to the optimal one.

We also compared the performance with four other popular SpMV implementations, including Ginkgo [3], ViennalCL [37], cuSPARSE [35], and Merge-based SpMV [31]. In the following sections, we use *ML* and *optimal* to represent the CSR-vector with the ML-predicted and the optimal thread configurations, respectively. Fig. 8 presents their performance comparison on two machines for the testing set. Table 6 summarizes their average speedup against the CSR-vector with *TpR* of 16. We also compared the number of sparse matrices where each algorithm presents the best performance, as presented in Table 7.

We summarize our observations for this set of experiments in the following points:

- Except CSR-vector with the optimal thread configuration, ML-predicted thread configuration achieves the highest overall performance on Tesla P100. Merge-based SpMV demonstrates good performance for some extremely irregular sparse matrices (68 matrices on V100 and 45 matrices on P100). As a result, its arithmetic mean speedup is higher than the geometric mean speedup on both GPUs. On Tesla V100, Merge-based SpMV achieves higher arithmetic mean speedup but smaller geometric mean speedup than ML.
- On Telsa V100, cuSPARSE is superior to *ML* for some sparse matrices, but the latter achieves better overall performance. Specifically, cuSPARSE achieves the best performance for 259 and 128 sparse matrices in the testing set on two machines, respectively. However, *ML* achieves the best performance for about 48% and 59% sparse matrices in the testing set on two machines, respectively.
- ML-predicted thread configuration achieves a performance exceptionally close to that of the optimal configuration. As shown in Table 6, both their arithmetic and geometric mean speedups show a very small gap, and the difference between the two on all GPU devices is less than 2%.

5.3. Blockwise thread configuration prediction

To investigate the performance gains from matrix partitioning, we first tested the performance of the CSR-vector without matrix partitioning with different thread configurations, including TpR of 16, *TpR_mean*, and TpR_SqMean. Then, we tested the performance of our new SpMV algorithm with different thread configurations, including TpR_mean, TpR_SqMean, *ML*, and *optimal*. Fig. 9 presents their performance comparison.

We can observe that the matrix partitioning algorithm proposed in this paper delivers a significant performance gain over non-partitioned algorithms. The reason is that the number of threads assigned by the CSR-vector to long rows does not exceed 32, and TpR_mean and TpR_SqMean assign a smaller number of threads. Thus the calculation of long-row blocks seriously reduces the SpMV performance. By partitioning a sparse matrix into long-row and short-row blocks, and assigning a different number of threads to them, the performance loss caused by threads starvation in long-row blocks and threads idleness in short-row blocks is alleviated. As can be seen in Fig. 9, blockwise TpR_mean and TpR_SqMean outperform global TpR_mean and TpR_SqMean for most matrices. Moreover, our ML-based thread configuration achieves a performance close to that of the optimal one for almost all matrices. Table 8 summarizes their average speedups.

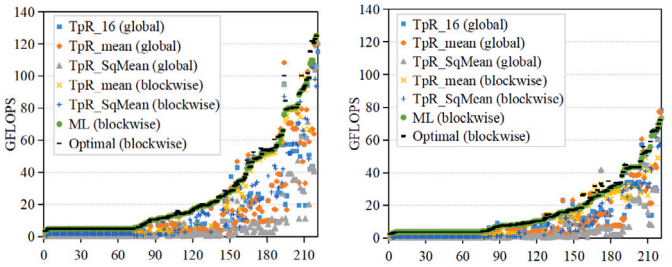
We use the trained model to predict the optimal thread configuration for short-row blocks, and the test accuracy on two machines is given in Table 9. Here, the accuracy is the proportion of matrices whose PLUB is in the interval [0,2%], which is a relaxed accuracy. The size of each short-row block after the partition is smaller than that of the

Table 8
Average speedup of SpMV with different thread configurations over the CSR-vector with TpR of 16.

GPU	Global		Blockwise			
	TpR_mean	TpR_SqMean	TpR_mean	TpR_SqMean	ML	Optimal
V100	0.86	0.34	2.63	2.12	3.00	**3.05**
P100	0.86	0.36	2.29	1.83	2.67	**2.72**

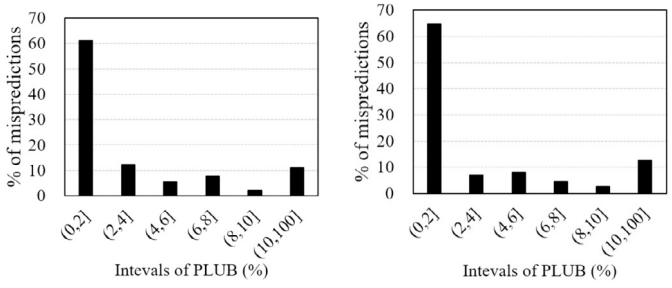
Table 9
Accuracy (Acc.) and overall performance of ML-based blockwise thread configuration prediction on two machines.

GPU	Acc.(%)	PLUB(%)	PGO(%)						
			2	4	8	16	32	TpR_ mean	TpR_ SqMean
V100	83.78	1.45	218.27	117.37	54.29	20.60	17.76	38.73	138.04
P100	81.98	2.00	216.95	96.31	35.17	17.27	46.80	25.29	127.35



(a) Tesla V100 (b) Tesla P100

Fig. 9. Performance comparison of global and blockwise SpMV with different thread configurations. The results labeled with *global* and *blockwise* were tested using CSR-vector and our proposed SpMV algorithm, respectively.

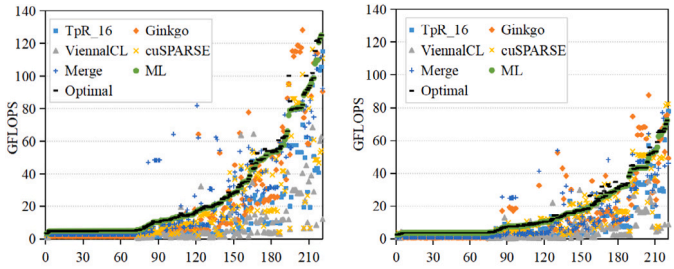


(a) Tesla V100 (b) Tesla P100

Fig. 10. Distribution of mispredictions for blockwise thread configuration in different PLUB intervals.

original matrix, and the distribution of the non-zero elements in each short-row block is more regular, which leads to a minor performance gap between different thread configurations. For example, the SpMV performance gap with the predicted thread configuration and the optimal configuration stands with only 0.01% of each other for the matrix *TSC_OPF_300*. Therefore, the relaxed accuracy can better evaluate the performance of the trained ML model. As shown in Table 9, the accuracy of the trained ML model is 84% for Tesla V100 and 82% for Tesla P100. Moreover, the PLUB on both machines does not exceed 2%. Additionally, Table 9 shows the average PGO of ML-based thread configuration against other thread configurations, and the proposed approach achieves better performance than the state-of-the-art solutions.

Fig. 10 summarizes the distribution of mispredictions in different PLUB intervals. We can observe that, on both machines, more than 60% of the mispredictions fall within the interval (0,2%], and only about 10% of the mispredictions fall within the interval (10%,100%].



(a) Tesla V100 (b) Tesla P100

Fig. 11. Performance comparison of seven SpMV algorithms on two machines for the testing set.

Although our trained model gives wrong predictions on some matrices, using the predicted thread configuration incurs a minor performance loss. In contrast, with the thread configuration of TpR_mean, 54% and 32% of mispredictions fall within the interval (10%,100%] on Tesla V100 and P100, respectively. With TpR_SqMean, 53%, and 52% mispredictions fall within the interval (10%,100%] on two machines, respectively.

We also compared the performance with several popular SpMV algorithms. Fig. 11 summarizes the results for the testing set on two machines. We sorted the experimental results by the performance of our proposed method for clear comparison. Besides, Table 10 gives the average speedups, and the number of matrices where each algorithm achieves the best performance is listed in Table 11. We summarize our observations in the following points.

- On both machines, blockwise SpMV using the ML predicted thread configuration achieves a performance close to the optimal one for almost all sparse matrices. Moreover, the performance of *ML* is superior to other compared algorithms for most matrices.
- As can be seen from Table 10, the Merge-based SpMV achieves a higher arithmetic mean speedup than our method on Tesla V100. It shows excellent performance on a small number of sparse matrices because of its strict load balance. However, our method achieves a higher geometric mean speedup of 2.64x than the Merge-based SpMV of 2.05x, which indicates that the proposed method is more stable and robust than the Merge-based SpMV.

6. Discussion

6.1. Diving deeper into model

LightGBM and XGBoost achieve high accuracy on both platforms according to Table 4. Since outstanding models are all ensemble mod-

Table 10
Average speedup of six SpMV algorithms over the CSR-vector with TpR of 16.

GPU	Metric	Ginkgo	ViennalCL	cuSPARSE	Merge	ML	Optimal
V100	AM	1.81	1.13	1.81	**3.60**	3.00	3.05
	GM	1.30	0.82	1.65	2.05	2.64	**2.67**
P100	AM	2.02	0.99	1.94	2.23	2.67	**2.72**
	GM	1.00	0.70	1.75	1.12	2.32	**2.36**

Table 11
The number of sparse matrices where each algorithm achieves the best SpMV performance.

GPU	TpR_16	Ginkgo	ViennaCL	cuSPARSE	Merge	ML	Optimal
V100	0	18	5	3	30	106	**165**
P100	3	16	6	25	13	102	**158**

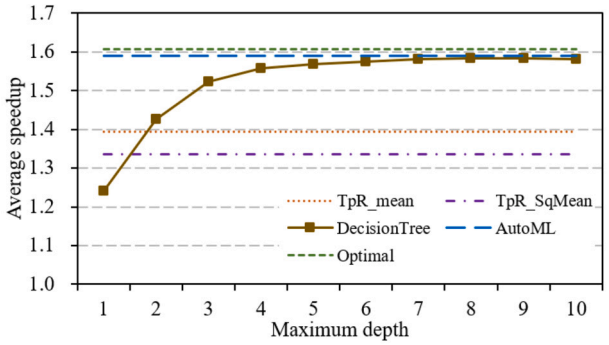


Fig. 12. The change of average speedup as the maximum depth of the decision tree increases.

els built based on decision trees (DTs), we experimentally found that a single DT can also achieve pretty good accuracy and overall performance. Taking global thread configuration prediction on Tesla P100 for example, we trained DTs with different maximum depths to observe the performance changes on the testing set. The results are presented in Fig. 12. It can be observed that: (1) a DT with a maximum depth of 2 is better than *TpR_mean* and *TpR_SqMean*; (2) when the maximum depth is greater than 5, the performance using the predicted thread configuration is close to that using the optimal one.

To observe the relationship between the features of a sparse matrix and its optimal thread configuration, we deep dive into the 5-level decision tree, and Fig. 13 presents its visualized result. It can be observed that when *max* (the maximum NNZ per row) of a sparse matrix is greater than 278.5, the thread configuration given by the decision tree is also relatively large, which is 16 or 32 in most cases, and 8 in a few cases (when $var \leq 2634.3298$, $d \leq 0.0001$, and $max\_mu \leq 693.1261$). However, the left branches of the DT's root node tend to choose relatively small thread configurations in most cases. Another important observation is that the 5-level DT does not use the feature *min*, which means that *min* contributes less to the thread configuration decision problem studied in this work. According to this tree, the most important features of a sparse matrix are *max*, *d* (density of non-zeros), *var* (variance of non-zeros per row), *m*, *n*, *nnz*, and *mean* (average NNZ per row) from the root to the leaves.

6.2. Overhead analysis

We also collected the performance of single-precision CSR-vector with different thread configuration approaches. By comparing the collected results of two precisions, we found that 72% (2,747/3,811) and 83% (3,151/3,811) sparse matrices have the same optimal thread configuration on Tesla V100 and Tesla P100, respectively. As for these sparse matrices with the different optimal *TpR* settings, their performance gap does not exceed 2% on both machines. It proves that, on a given GPU, a sparse matrix has similar preferences to thread configuration when performing SpMV calculations with different floating-point precision. Therefore, our model can be applied to both double and single-precision calculations.

As for hardware architecture, there are only a limited number of mainstream GPU architectures, which may have extremely different architectural characteristics and thus present different performances for the same sparse matrix. For example, the Volta architecture first introduced independent thread scheduling among threads in a warp, which is a very important characteristic. Therefore, we recommend building individual models for different GPUs. Collecting the performance of SpMV under different thread configurations takes about two to three hours, and the model training only costs minutes or even seconds. Moreover, these collected data and trained models on a specific GPU can be shared by all users who are using the same GPU. This work can be done by GPU manufacturers or third parties in one go and shared by users worldwide. On that account, we believe that the proposed solution is valuable.

In practical application, our proposed method includes two parts of overhead: calculating sparse matrix features and predicting the optimal thread configuration using the trained model. Our statistical results on the testing set show that the cost of calculating matrix features is approximately one SpMV of CSR-vector with TpR of 32. As for the overhead of model inference, we observed the changes in accuracy and inference cost under different parameter configurations using the LightGBM-based model. We used the m2cgen tool [9] to export the trained model into C code and collected its average time of 500 inferences. Fig. 14 and Fig. 15 present the change of model accuracy and inference cost with the number of estimators and the maximum depth of DTs. It can be observed that the inference of the model that achieves the highest accuracy takes only a few microseconds.

One naive alternative of our ML-based thread configuration is to walk over different *TpR* in the first five iterations and continue the rest of the iterations with the best one. We use T_{naive} to represent the execution time of this naive approach in the first five SpMV, and it is calculated by Equation (6).

$$T_{naive} = t_{TpR_2} + t_{TpR_4} + t_{TpR_8} + t_{TpR_{16}} + t_{TpR_{32}} \tag{6}$$

In which, t_{TpR_2} is the execution time of CSR-vector with *TpR* of 2. Let T_{ML} represent the execution time of our ML-based thread configuration method in the first five SpMV. Equation (7) gives its calculation expression.

$$T_{ML} = t_{calF} + t_{model} + t_{TpR_{ML}} \times 5 \tag{7}$$

In which, t_{calF}, t_{model}, and $t_{TpR_{ML}}$ are the time to calculate the matrix features, the time to predict the optimal thread configuration and the time to execute SpMV using the predicted thread configuration, respec-

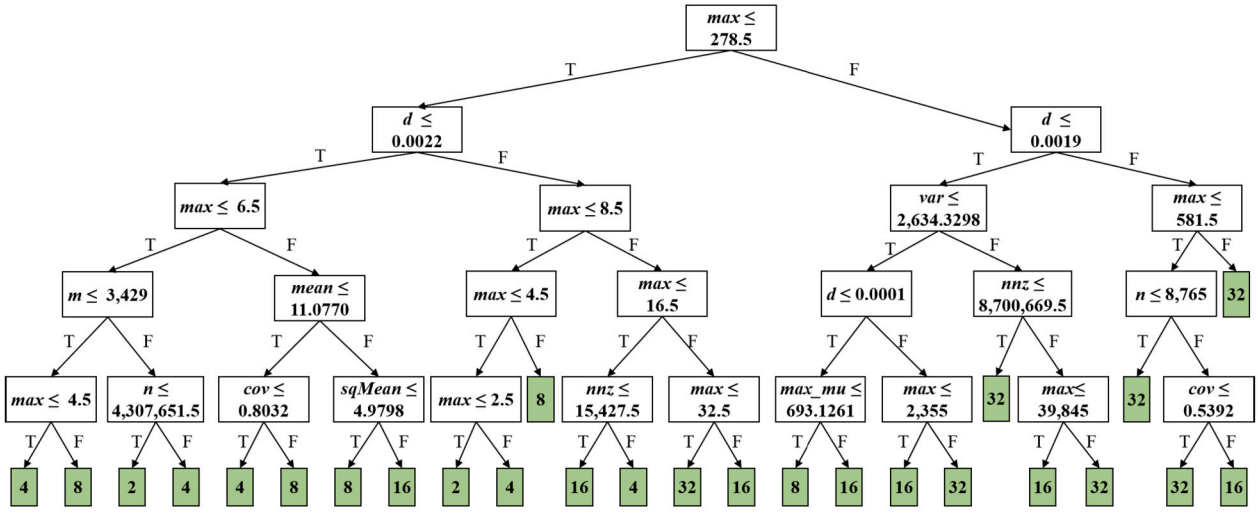


Fig. 13. Visualized 5-level decision tree. The nodes filled with green color represent the thread configurations decided by the tree.

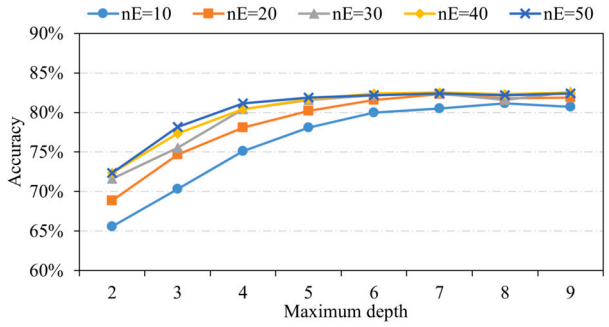


Fig. 14. The change of model accuracy based on LightGBM with the number of estimators (nE) and the maximum depth of DTs.

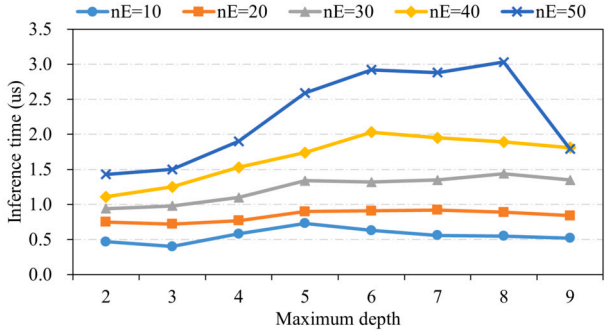


Fig. 15. The change of model inference time with the number of estimators (nE) and the maximum depth of DTs.

tively. We calculated the speedup (T_{naive}/T_{ML}) of our ML-based method over the naive method for the testing set on Tesla V100, and results show that our ML-based method achieves an average speedup of 1.28x and maximum speedup of 7.15x.

In practical use, m2cgen can help transform ML models into native codes like Java, C, Python, Go, JavaScript, and so on. Therefore, our proposed machine learning-based thread configuration scheme can finally be expressed as a function that allows users to call conveniently. Given an input sparse matrix, we first calculate its features and then take these features as the input of the function. The output of the function is the thread configuration predicted by our model. Finally, the predicted thread configuration is input into the SpMV kernel to lead thread assignment.

7. Related work

7.1. CSR-based SpMV algorithms

Since format transformation may introduce much more overhead in preprocessing, there are many researchers committed to accelerating CSR-based SpMV on GPU in recent years. Reguly et al. [36] suggested two thread configurations for CSR-vector [11]. The first is the thread configuration of TpR_SqMean. The second is a run-time tuning rule. Our investigation of more than 2,000 sparse matrices shows that the SpMV with the optimal thread configuration is about 3x on average and up to 20x better than that with the worst configuration. It indicates that inappropriate *TpR* given by the fixed rule will cause durative performance losses during the iteration process. As for the run-time tuning rule, an inappropriate or even the worst *TpR* in the initial iterations significantly affects the overall performance. Greathouse et al. [23][15] proposed a CSR-Adaptive algorithm to address the load unbalance of the CSR-vector. Ashari et al. [5] proposed an adaptive SpMV algorithm, ACSR, to reduce thread divergence and alleviate load unbalance. Merrill et al. [31] designed a merge-based SpMV algorithm to tackle the problems of load unbalancing and heavy preprocessing in SpMV. It frames the CSR-based SpMV as a logical merge of two lists: row pointer and column index arrays, and assigns equal-sized shares of the merge to each processing element, thus standing a strict load balance. Flegar et al. [22] proposed CSR-I for irregular sparse matrices, which also evenly divided the workload among all threads. They presented a decision strategy based on the deviation-to-mean ratio to choose different SpMV algorithms for sparse matrices. SURAA [33] used different SpMV algorithms to process sorted and grouped sub-matrices, and these algorithms include CSR-scalar, CSR-vector (setting threads assigned to each row equal to or greater than 32), and a dynamic kernel (its basic component is the CSR-vector assigning 32 threads to each row. Besides, [28][39][2][40] also proposed CSR-based SpMV optimization on GPU.

7.2. ML-based algorithms

The performance of the SpMV algorithm depends on many factors, including the size of the input sparse matrix, the distribution pattern of non-zero elements, the encoding format adopted, as well as the architectural features of hardware devices. Existing heuristic methods can not fully take care of all these issues, and there is no one algorithm that can rule matrices all. With the continuous development and wide application of machine learning technology, ML-based SpMV optimization

has gradually attracted the attention of researchers, and a number of works have been reported recently.

7.2.1. Format/algorithm selection

As more sparse formats and SpMV algorithms are presented, researchers propose to predict suitable formats, as well as corresponding SpMV algorithms, for different sparse matrices using ML models. Armstrong et al. [4] used reinforcement learning to determine the most efficient sparse formats. However, only three sparse matrix compressing formats are considered in their work. Li et al. [26] and Sedaghati et al. [38] proposed to predict the best format using a decision tree-based model. Benatia et al. [12] and Mehrez et al. [29] used multiclass SVM to select the best format. Nisa et al. [34] tested multiple basic and ensemble ML algorithms on format selection, including DT, SVM, multi-layer perceptrons (MLP), and extreme gradient boosting (XGBoost). As there are many sparse matrices available, Zhao et al. [41] put forward to find the best formats using a deep learning model. Considering that the overhead caused by matrix features extracting, model prediction, and format converting may offset the benefits in some application scenarios, Shen et al. [42] proposed a two-stage lazy-and-light prediction scheme. In the first stage, they predict the total number of SpMV iterations using a two-layer long short-term memory (LSTM) network. The second stage will be invoked only when the predicted value is larger than the given threshold. An XGBoost model is used to predict the best compression format in the second stage. Dufrechou et al. [19] used the bagged trees classifier to predict the best method according to a feature set of matrices, and they considered the two criteria of running time and energy consumption respectively to indicate the best method. These works have shown that choosing a suitable sparse format can greatly improve the performance of SpMV. However, an inevitable problem is to convert sparse matrices from one format to another, which is bound to introduce additional format conversion overhead. Therefore, we may not get significant performance gains from single SpMV calculation or iterative algorithms with less iterations. Differently, Elafrou et al. [20] used a DT-based classifier to predict the major performance bottleneck for input sparse matrix, and then selected the corresponding CSR-based algorithm to execute SpMV.

7.2.2. Performance prediction

Large-scale SpMV computing requires collaborative computing of multiple GPUs. We need to know the execution time of SpMV calculation for each matrix partition to design an effective task-scheduling algorithm. Therefore, the performance prediction of SpMV is important. Similar to format selection, researchers have proposed various ML-based approaches to predict the performance of SpMV. Benatia et al. [13] used SVM and MLP to predict the performance of SpMV kernels using different sparse formats. Nisa et al. [34] supplemented other ML models, including XGBoost and ensembles of MLP. Later, Barreda et al. [6,7] designed off-line estimators of execution time and energy consumption for SpMV using CNN as the base algorithms.

7.2.3. Parameter configuration prediction

In some SpMV algorithms, parameter settings are important and crucial for their performance. For example, in SpMV algorithms using blocked or sliced compression formats, the size of each block or slice significantly affects the size of the compressed matrix and the performance of the corresponding kernel. The optimal parameter setting varies with the matrices and hardware. It is difficult to design a specific rule to find the optimal parameter setting for each sparse matrix on a given hardware platform. Ahmed et al. [1] proposed an ML-based tool to predict the fastest block size setting for BCSR. Based on a feature set consisting of five basic, easy-to-compute features and nine high-complexity features, they trained and tested this tool using different machine learning methods (including decision tree, random forest, gradient boosting, ridge regressor, and AdaBoost).

7.2.4. Differences from our work

Compared with the above-mentioned ML-based work, our algorithm uses the most popular CSR format and does not require any heavy format conversion of sparse matrices. We focus on the problem of thread configuration for the CSR-based SpMV algorithm. As far as we know, our work is the first attempt to apply the ML-based approach to thread configuration optimization. Moreover, existing works take a matrix as a whole for thread configuration, without considering the distribution of non-zero elements in each sparse matrix. Our block-wised approach splits irregular matrices into different blocks so that we can find the near-optimal thread configuration for each block using our ML model.

8. Conclusion

This paper proposes a new machine learning-based thread assignment strategy for SpMV on GPUs. Based on an easy-to-calculate feature set, it can predict the approximate optimal thread configuration for a given sparse matrix. Furthermore, an optimization method for irregular sparse matrices is proposed. We first partition these matrices into multiple blocks according to the distribution of non-zero elements. Then, we predict the optimal thread configuration for each block based on its features. We also design a new SpMV kernel to accelerate the calculation of different blocks. Our experimental evaluation of two different GPU architectures shows that the proposed method delivers significant performance improvement. Finally, we discuss the connections between the features of a sparse matrix and its optimal thread configuration, and derive a lightweight inference model that costs only a few microseconds for predicting in practical use.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (grant No. 61972033).

References

[1] M. Ahmed, S. Usman, N.A. Shah, M.U. Ashraf, A.M. Alghamdi, A.A. Bahadded, K.A. Almarhabi, AAQAL: a machine learning-based tool for performance optimization of parallel SpMV computations using block CSR, Appl. Sci. 12 (14) (2022) 7073, https://doi.org/10.3390/app12147073.

[2] H. Anzt, T. Cojean, C. Yen-Chen, J.J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y.M. Tsai, W. Wang, Load-balancing sparse matrix vector product kernels on GPUs, ACM Trans. Parallel Comput. 7 (1) (2020) 2:1–2:26, https://doi.org/10.1145/3380930.

[3] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.M. Tsai, E.S. Quintana-Ortí, Ginkgo: a modern linear operator algebra framework for high performance computing, ACM Trans. Math. Softw. 48 (1) (feb 2022), https://doi.org/10.1145/3480935.

[4] W. Armstrong, A.P. Rendell, Reinforcement learning for automated performance tuning: initial evaluation for sparse matrix format selection, in: 2008 IEEE International Conference on Cluster Computing, IEEE Computer Society, Tsukuba, Japan, 2008, pp. 411–420.

[5] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, P. Sadayappan, Fast sparse matrix-vector multiplication on GPUs for graph applications, in: T. Damkroger, J.J. Dongarra (Eds.), International Conference for High Performance Computing, Networking, Storage and Analysis, SC, New Orleans, LA, USA, November 16–21, IEEE Computer Society, 2014, pp. 781–792.

[6] M. Barreda, M.F. Dolz, M.A. Castaño, P. Alonso-Jordá, E.S. Quintana-Orti, Performance modeling of the sparse matrix-vector product via convolutional neural networks, J. Supercomput. 76 (11) (2020) 8883–8900, https://doi.org/10.1007/s11227-020-03186-1.
[7] M. Barreda, M.F. Dolz, M.A. Castano, Convolutional neural nets for estimating the run time and energy consumption of the sparse matrix-vector product, Int. J. High Perform. Comput. Appl. 35 (3) (2021) 268–281, https://doi.org/10.1177/1094342020953196.
[8] M.M. Baskaran, R. Bordawekar, Optimizing Sparse Matrix-Vector Multiplication on GPUs Using Compile-Time and Run-Time Strategies, IBM Reserach Report, RC24704 (W0812-047), 2008, https://www.academia.edu/download/31213667/ibm-sparse-gpu.pdf.
[9] BayesWitnesses, m2cgen: transform ML models into a native code, v10.0.0, https://github.com/BayesWitnesses/m2cgen. (Accessed 21 May 2023).
[10] M. Belgin, C.J. Ribbens, G. Back, An operation stacking framework for large ensemble computations, in: Proceedings of the 21st Annual International Conference on Supercomputing (ICS), Association for Computing Machinery, Seattle, Washington, 2007, pp. 83–92.
[11] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), Portland, Oregon, USA, November 14–20, ACM, 2009.
[12] A. Benatia, W. Ji, Y. Wang, F. Shi, Sparse matrix format selection with multiclass SVM for SpMV on GPU, in: 2016 45th International Conference on Parallel Processing (ICPP), IEEE Computer Society, Philadelphia, PA, USA, 2016, pp. 496–505.
[13] A. Benatia, W. Ji, Y. Wang, F. Shi, Machine learning approach for the predicting performance of SpMV on GPU, in: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS), 2016, pp. 894–901, https://10.1109/ICPADS.2016.0120.
[14] A. Benatia, W. Ji, Y. Wang, F. Shi, BestSF: a sparse meta-format for optimizing SpMV on GPU, ACM Trans. Archit. Code Optim. 15 (3) (sep 2018), https://doi.org/10.1145/3226228.
[15] M. Daga, J.L. Greathouse, Structural agnostic SpMV: adapting CSR-adaptive for irregular matrices, in: 2015 IEEE 22nd International Conference on High Performance Computing (HiPC), Bengaluru, India, 2015, pp. 64–74.
[16] S. Dalton, N. Bell, L. Olson, M. Garland, Cusp: generic parallel algorithms for sparse matrix and graph computations, version 0.5.0, http://cusplibrary.github.io/, 2014.
[17] T.A. Davis, Y. Hu, The university of Florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1–25, https://doi.org/10.1145/2049662.2049663.
[18] Y. Deng, S. Mu, Electronic design automation with graphic processors: a survey, Found. Trends Electron. Des. Autom. 7 (1–2) (2013) 1–176, https://doi.org/10.1561/1000000028.
[19] E. Dufrechou, P. Ezzatti, E.S. Quintana-Ortí, Selecting optimal SpMV realizations for GPUs via machine learning, Int. J. High Perform. Comput. Appl. 35 (3) (2021), https://doi.org/10.1177/1094342021990738.
[20] A. Elafrou, G. Goumas, N. Koziris, BASMAT: bottleneck-aware sparse matrix-vector multiplication auto-tuning on GPGPUs, in: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP), Washington, District of Columbia, Association for Computing Machinery, New York, NY, USA, 2019, pp. 423–424.
[21] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, A.J. Smola, AutoGluon-tabular: robust and accurate AutoML for structured data, CoRR, arXiv: 2003.06505 [abs], 2020.
[22] G. Flegar, E.S. Quintana-Ortí, Balanced CSR sparse matrix-vector product on graphics processors, in: F.F. Rivera, T.F. Pena, J.C. Cabaleiro (Eds.), Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings, in: Lecture Notes in Computer Science, vol. 10417, Springer, 2017, pp. 697–709.
[23] J.L. Greathouse, M. Daga, Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC, New Orleans, LA, USA, November 16-21, IEEE Computer Society, 2014, pp. 769–780.
[24] X. He, K. Zhao, X. Chu, AutoML: a survey of the state-of-the-art, Knowl.-Based Syst. 212 (2021) 106622, https://doi.org/10.1016/j.knosys.2020.106622.
[25] K.H. Law, On updating the structure of sparse matrix factors, Int. J. Numer. Methods Eng. 28 (10) (1989) 2339–2360, https://doi.org/10.1002/nme.1620281010, https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.1620281010.
[26] J. Li, G. Tan, M. Chen, N. Sun, SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication, SIGPLAN Not. 48 (6) (2013) 117–126, https://doi.org/10.1145/2499370.2462181.
[27] W. Liu, B. Vinter, CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication, in: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS), Association for Computing Machinery, Newport Beach, California, USA, 2015, pp. 339–350.
[28] Y. Liu, B. Schmidt LightSpMV, Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs, in: 26th IEEE International Conference on Application-Specific Systems, Architectures and Processors, ASAP, Toronto, ON, Canada, July 27–29, IEEE Computer Society, 2015, pp. 82–89.
[29] I. Mehrez, O. Hamdi-Larbi, T. Dufaud, N. Emad, Machine learning for optimal compression format prediction on multiprocessor platform, in: 2018 International Conference on High Performance Computing Simulation (HPCS), IEEE Computer Society, Orleans, France, 2018, pp. 213–220.
[30] D. Merrill, Merge-based parallel sparse matrix-vector multiplication, https://github.com/dumerrill/merge-spmv. (Accessed 5 October 2023).
[31] D. Merrill, M. Garland, Merge-based parallel sparse matrix-vector multiplication, in: J. West, C.M. Pancake (Eds.), Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC, Salt Lake City, UT, USA, November 13–18, IEEE Computer Society, 2016, pp. 678–689.
[32] J.M. Montgomery, F.M. Hollenbach, M.D. Ward, Calibrating ensemble forecasting models with sparse data in the social sciences, Int. J. Forecast. 31 (3) (2015) 930–942, https://doi.org/10.1016/j.ijforecast.2014.08.001.
[33] T. Muhammed, R. Mehmood, A. Albeshri, I. Katib, SURAA: a novel method and tool for loadbalanced and coalesced SpMV computations on GPUs, Appl. Sci. 9 (5) (2019) 947, https://doi.org/10.3390/app9050947.
[34] I. Nisa, C. Siegel, A.S. Rajam, A. Vishnu, P. Sadayappan, Effective machine learning based format selection and performance modeling for SpMV on GPUs, in: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE Computer Society, Vancouver, BC, Canada, 2018, pp. 1056–1065.
[35] NVIDIA, Nvidia CuSPARSE library, https://docs.nvidia.com/cuda/archive/11.1.1/cusparse/index.html, 2020.
[36] I. Reguly, M. Giles, Efficient sparse matrix-vector multiplication on cache-based GPUs, in: 2012 Innovative Parallel Computing (InPar), San Jose, CA, USA, 2012, pp. 1–12.
[37] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, S. Selberherr, ViennaCL—linear algebra library for multi- and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439, https://doi.org/10.1137/15M1026419.
[38] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, P. Sadayappan, Automatic selection of sparse matrix representation on GPUs, in: Proceedings of the 29th ACM on International Conference on Supercomputing (ICS), Association for Computing Machinery, Newport Beach, California, USA, 2015, pp. 99–108.
[39] M. Steinberger, R. Zayer, H. Seidel, Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU, in: W.D. Gropp, P. Beckman, Z. Li, F.J. Cazorla (Eds.), Proceedings of the International Conference on Supercomputing, ICS, Chicago, IL, USA, June 14–16, ACM, 2017, pp. 13:1–13:11.
[40] Y.M. Tsai, T. Cojean, H. Anzt, Sparse linear algebra on AMD and NVIDIA gpus - the race is on, in: High Performance Computing - 35th International Conference, ISC High Performance, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings, in: Lecture Notes in Computer Science, vol. 12151, Springer, 2020, pp. 309–327.
[41] Y. Zhao, J. Li, C. Liao, X. Shen, Bridging the gap between deep learning and sparse matrix format selection, in: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Vienna, Austria, PPoPP '18, Association for Computing Machinery, 2018, pp. 94–108.
[42] W. Zhou, Y. Zhao, X. Shen, W. Chen, Enabling runtime SpMV format selection through an overhead conscious method, IEEE Trans. Parallel Distrib. Syst. 31 (1) (2020) 80–93, https://doi.org/10.1109/TPDS.2019.2932931.

Jianhua Gao received her BS degree in College of Mathematics from Taiyuan University of Technology, China, in 2017. She is currently working toward the PhD degree in the School of Computer Science and Technology, Beijing Institute of Technology. Her research interests mainly include parallel and high performance computing.

Weixing Ji received the PhD degree in School of Computer Science and Technology from Beijing Institute of Technology in 2008. He is currently a Full Professor of School of Computer Science and Technology with Beijing Institute of Technology. His research interests mainly include parallel computing and program analysis and optimization.

Jie Liu received her B.S. degree in the School of Computer Science and Technology, Chongqing University. She received her M.S. degree in the School of Computer Science and Technology, Beijing Institute of Technology. Her research interests include computer architecture, parallel and high performance computing.

Yizhuo Wang received the Ph.D. degree in Computer Science from Beijing Institute of Technology, China, in 2005. Now he is an Assistant Professor of Computer Science at Beijing Institute of Technology. His research interests include parallel programming, fault tolerance computing and computer architectures.

Feng Shi was born in 1961. He received the B.S. degree in School of Physics from Peking University, China, in 1983 and the Ph.D. degree in School of Computer Science and Technology from Beijing Institute of Technology, China, in 1999. He is currently a full professor of School of Computer Science and Technology with Beijing Institute of Technology. His research interests mainly include computer architecture and embedded system.