Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures

Atefeh Mehrabi*, Donghyuk Lee†, Niladrish Chatterjee†, Daniel J. Sorin*, Benjamin C. Lee‡, Mike O'Connor†§
*Duke University, † NVIDIA, ‡University of Pennsylvania, §UT Austin
*atefeh.mehrabi@duke.edu, †donghyukl@nvidia.com, †nchatterjee@nvidia.com
*sorin@ee.duke.edu, ‡leebcc@seas.upenn.edu, †§moconnor@nvidia.com

***Abstract*—Achieving peak performance on sparse operations is challenging. The distribution of the non-zero elements and underlying hardware platform affect the execution efficiency. Given the diversity in workloads and architectures, no unique solution always wins. In this paper, we improve SpMM efficiency on GPUs. We propose several simple, but effective, sparse data permutations on the CSR data structure. Picking the right permutation over 1,688 datasets improves performance by 1.4×, on average, compared to plain CSR and 2.6× against NVIDIA cuSPARSE. Furthermore, we propose a set of novel features to describe sparsity patterns and their interactions with the kernel and hardware. Using these features, we develop a predictor to select the best permutation for each matrix. Predicted permutations' average gain achieves 96% of oracle gains.**

***Index Terms*—Sparse linear algebra, SpMM, GPU, HPC**

I. Introduction

Sparse matrix computation is integral to varied applications in high-performance computing [1], graph analytics [2], and machine learning [3]. Computation on sparse matrices is challenging due to their many zero-valued elements, which induce irregular memory access patterns. Sparsity degree (*i.e.,* percentage of zero-valued elements) varies from from approximately 50% in CNNs to more than 99% in big graph analytics.

In this paper, we optimize Sparse Matrix Multi-vector Multiplication (SpMM) performance on Graphics Processing Units (GPUs). GPUs can exploit data parallelism in SpMM, but the distribution of non-zero elements across rows and columns of the matrix (*i.e.,* sparsity pattern) affects load balance and data locality. Skipping operations with zero-valued operands saves time and space. But this strategy also creates imbalanced load across parallel processing elements, which harms performance and degrades resource utilization.

We study performance optimizations for SpMM with compressed sparse row (CSR) format. Standard libraries predominantly use CSR or DCSR [4]. Other formats exist, but may require expensive conversions or storage overheads. SpMM performance with CSR varies with both the sparsity pattern and the GPU design. Sparsity requires strategies to better balance load across warps within a cooperative thread array (CTA) and strategies to improve data locality to reduce memory bandwidth demand.

In this paper, we study sources of inefficiency of the CSR format on GPU performance and propose permutation techniques for matrix rows to improve data locality, load balance, or both. We improve SpMM performance by identifying the permutation technique best suited for a given matrix sparsity pattern. We do so by modeling performance as a function of summary statistics and abstract features that describe matrix structure and hardware-software interactions. In summary,

- We analyze effects of load balance and data locality when GPUs perform SpMM. The analysis motivates our proposed policies for matrix permutations.
- We propose several row permutation strategies for CSR to improve load balance and data locality on GPUs. Moreover, we select the best permutation given matrix features. We improve performance by 1.4× on average and up to 20× compared to plain CSR on two GPUs.
- We derive features that model the relationship between sparsity pattern and GPU performance. These features reveal the best permutation strategy for each matrix. For 1,688 sparse matrices, performance under our model-driven policy is 96% of the performance under an oracular policy.

This paper is a first step towards broader studies on efficient sparse computations on a variety of data structures and hardware platforms like neuromorphic accelerators [5]. Given the success of our performance models, studying more complex predictors (e.g. deeper neural networks) could extend our results to higher dimensional, sparse tensors in CNNs.

II. Background

A GPU consists of multiple streaming multiprocessors (SMs) that execute kernels in parallel. Each SM runs multiple cooperative thread arrays (CTAs), which are also known as thread blocks, that are defined by the program. Each CTA consists of multiple warps, each with a fixed number of threads (*e.g.*, 32). Threads within a warp execute the same instruction on multiple data (*i.e.*, SIMD). Each SM has fast private memory such as the L1 cache and a shared memory space (shmem) for inter-thread communication. Shared memory space of an SM is divided between different CTAs of that SM. Different SMs communicate through an L2 cache and global memory.

A. Sparse Data Structures

SpMM multiplies sparse matrix A and dense matrix B, producing output matrix C. Storing sparse matrices efficiently requires compressed representations. Among numerous formats, CSR is most widely used [6] [7].

Algorithm 1 Parallel Sparse Matrix Multi-vector Multiplication

Input: CSR A[M][N], float B[N][K]
Output: float C[M][K]

```
1: int WarpSize = 32;     //number of threads in each warp
2: int wid = threadIdx.x/WarpSize;     //warp ID
3: int twid = threadIdx.x%WarpSize;    //thread ID within a warp
4: int numWarps = blockDim.x/WarpSize      //number of warps within a block
5: for i = wid; i < M; i+= numWarps do
6:     for j = rowidx[i] + twid; j < rowidx[i + 1]; j+= WarpSize do
7:         //partial sum computation and accumulation
```

Compressed Sparse Row (CSR). CSR represents a sparse matrix with *M* rows and *nnz* non-zero elements using three arrays—`rowidx`, `colidx`, and `value`. Together, these arrays specify the row, column, and value for each non-zero element and avoid explicitly storing the many zeros. The `value` and `colidx` arrays, each with *nnz* elements, specify the value of non-zero elements and their column indices in the order of their occurrence across rows. The `rowidx` array, with *M*+1 elements, specifies where each matrix row starts and ends in the `value` and `colidx` arrays. `rowidx[i]` specifies the aggregate number of non-zero values in the first `i` rows. Starting row indices from 0, `rowidx[i+1]` − `rowidx[i]` gives the number of non-zeros in row `i`.

Densified Compressed Sparse Row (DCSR). DCSR further improves compression efficiency and reduces memory traffic [4]. The CSR `rowidx` array counts the number of non-zero elements in each row, but this array contains redundant information when matrices have many empty rows. DCSR introduces a `rowptr` array to identify non-empty rows and modifies `rowidx` to store information only for those rows. Fig. 1 illustrates the two representations for a 4×4 matrix.

$$\mathbf{A} = \begin{pmatrix} 0 & a_{01} & 0 & a_{03} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ a_{30} & 0 & 0 & 0 \end{pmatrix}$$

$\mathbf{rowidx} = (0 \;\; 2 \;\; 2 \;\; 2 \;\; 3)$
$\mathbf{colidx} = (1 \;\; 3 \;\; 0)$
$\mathbf{value} = (a_{01} \;\; a_{03} \;\; a_{30})$

$\mathbf{rowidx} = (0 \;\; 2 \;\; 3)$
$\mathbf{rowptr} = (0 \;\; 3)$
$\mathbf{colidx} = (1 \;\; 3 \;\; 0)$
$\mathbf{value} = (a_{01} \;\; a_{03} \;\; a_{30})$

Fig. 1: Sparse data formats. CSR (left). DCSR (right)

B. Output Stationary SpMM

SpMM can be parallelized for GPU computation. Tiling partitions the matrix and allows multiple processing elements to compute on different partitions, impacting data locality and computation efficiency [8]. When tiles fit in caches, communication with memory decreases and performance increases. SpMM kernels that keep tiles of matrix A, B and C in shared memory are referred to as A-stationary, B-stationary and output(C)-stationary, respectively. B- and output-stationary kernels are most common because the sparse matrix A's memory footprint is much smaller than the others' [8]. Algorithm 1 presents a psuedo code for parallel SpMM via an output-stationary kernel implementation.

In Fig. 2, sparse matrix A is partitioned horizontally and dense matrix B vertically. Sparse matrix A is stored in row-major order with CSR format, which aligns with the horizontal tiling required for output-stationary computation. Dense matrix B is stored in column-major order, which maximizes data locality in tile traversals. A and B are partitioned such that each partition holds multiple rows and columns, respectively.

Output matrix C's tiles could be computed in either row- or column-major order. Row-major computation exploits locality for one strip of A but requires expensive traversal over all strips of B. In contrast, column-major computation exploits locality for B and usually performs better due to dense matrix B's larger memory footprint. In our implementation, each CTA in one SM processes a column of C's tiles by traversing over strips of sparse A and reusing a single strip from dense B.

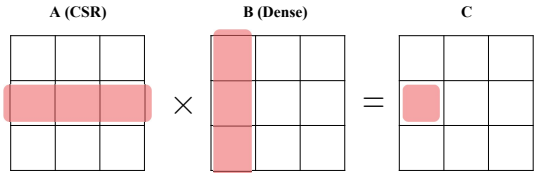


Fig. 2: Output(C)-Stationary SpMM Kernel

Suppose each CTA has N_w warps and each warp has N_t threads. Row $i \in [0, M)$ of A is assigned to warp $j = i\%N_w$. Each warp's threads compute on the non-zero elements of its assigned rows in the sparse matrix A (*e.g.*, when $N_w = 32$, warp $j = 0$ computes on rows 0, 32, 64, ...). Multiplication of non-zero elements of a sparse matrix's row happen on threads of the corresponding warp in parallel. Thread local registers accumulate partial sums without atomic operations, an advantage of the output-stationary kernel.

C. SpMM Performance Factors

Load Distribution. Matrix values are often distributed non-uniformly across rows, constraining utilization and degrading performance. A CTA cannot release its resources until its warps become idle, and a warp's computation completes when its N_t threads process their values. The longest running warp

in a CTA dictates the critical path. Imbalanced work among warps leads to cases where a CTA is blocked from accepting new tasks only because a few warps are still working. This degrades performance by wasting parallel processing elements.

A warp's computational load increases with the number of non-zero elements nnz_R in each assigned row R. If nnz_R exceeds the number of threads N_t, the warp's computation proceeds iteratively until all non-zeros are processed. When a warp with N_t threads processes row R, the maximum number of non-zeros assigned to a thread is $\left\lceil \frac{nnz_R}{N_t} \right\rceil$; we refer to this measure as *warp load*. For example, consider a warp with 32 threads that compute on two rows with 33 and 64 non-zeros. The *warp load* for both rows is two ($= \lceil \frac{33}{32} \rceil = \lceil \frac{64}{32} \rceil$), but thread utilization differs. In the second iteration, the warp for the row with 33 non-zeros utilizes only one thread whereas the warp for the row with 64 non-zeros utilizes 32 threads. Poor utilization translates into unexploited opportunities for higher throughput.

Memory Bandwidth Demand. SpMM is memory-bound. During the execution of an output-stationary kernel, each thread multiplies a non-zero in sparse matrix A with the corresponding value in dense matrix B. Because non-zeros are non-uniformly distributed in A, memory requests for B's matrix elements are irregular and span a variable number of cache lines. Rather than the number of non-zeros, it is the location of those non-zeros within a row that determines the cache lines a warp must fetch from dense matrix B. Given two rows with the same number of non-zeros, a warp may fetch a significantly different number of cache lines.

Suppose each 128B cache line holds 32 consecutive elements of a column in B. If a warp performs multiplication for 32 consecutive non-zeros in a row of A, it can obtain the corresponding 32 elements from B's column, stored in column-major order, by loading a single cache line. In contrast, if a warp performs multiplication for 32 sparse non-zeros distributed across columns (*e.g.*, columns 0, 32, 64, etc.), it must obtain the corresponding 32 elements from B by loading 32 different cache lines. Thus, the locations of non-zero elements within A's rows define access patterns in B's columns, impacting performance through data locality and cache effectiveness.

III. Data Permutation Policies

Permutations re-order sparse matrix rows to improve load distribution and data locality. We propose row permutation techniques to improve SpMM performance on GPUs yet preserve the CSR representation of the sparse matrix. Row permutations happen offline or the cost is amortized as one sparse matrix is likely to be used several times.

A. Load Balancing Permutations

We re-order rows of the CSR matrix to reduce load variation across parallel warps within a CTA, thereby improving utilization. Prior studies categorize rows based on the number of non-zeros and use different SpMV kernels for each category [9]. In contrast, we re-order rows to better align the non-zero distribution with a single output-stationary kernel, avoiding the costs of invoking multiple kernels.

Plain Sort. We sort rows by *warp load* before assigning computation. This technique clusters rows with similar numbers of non-zeros and increases the likelihood of balanced work across a CTA's warps. Although this technique does not guarantee fully balanced loads, it improves load balance across groups of consecutive rows.

Flipped Sort. We follow the same logic as Plain Sort with one difference. After every N_w=32 rows, the assignment order of rows to warps is flipped to avoid assigning the largest load in each subset of N_w rows to the first warp. Although this technique does not guarantee fully balanced loads, it reduces the likelihood of one warp consistently receiving higher loads and improves load balance across groups of consecutive rows.

Longest Processing Time (LPT) Sort: We assign rows to minimize the longest running warp's load. First, we sort rows in decreasing order of *warp load*. Then, we assign each row to the warp with the smallest load thus far. This greedy assignment minimizes maximum warp load, which defines how long a CTA is busy.

B. Cache-Aware Permutations

Rows with the same number of non-zeros could differ significantly in the distribution of those non-zeros, which determines cache access patterns for dense matrix B. When each collection of 32 contiguous columns in sparse matrix A's row holds at least one non-zero, the SpMM kernel must read the corresponding cache line for dense matrix B. Cache-aware sort strategies aim to improve cache performance by increasing locality and re-use for read cache lines.

First, we translate matrix A's sparsity patterns into matrix B's cache access patterns. We construct a bit mask for each sparse matrix row. If the i-th block of 32 contiguous columns in the row contains at least one non-zero, $b[i] = 1$ in the mask and the i-th cache line from matrix B's column is required. Otherwise, $b[i] = 0$. A sparse matrix with *N* columns produces a mask with $\lceil \frac{N}{32} \rceil$ bits.

Second, we measure data re-use between pairs of rows, which will be multiplied with the same column of B, by assessing their masks' similarities. Similar masks correlate with higher re-use as the computation observes fewer 0-1 and 1-0 transitions, which indicate a non-cached block of B is read and a cached block is not re-used, respectively. We measure similarity with the Hamming distance (*i.e.*, XOR) between two rows' masks. The following row permutation strategies maximize similarity to improve locality.

Warp-Aware Sort. To increase intra-warp data re-use, we re-order rows to minimize distances between adjacent rows in each warp. Adjacent rows are located 32 rows away from each other in the sparse matrix because rows are striped across the $N_w = 32$ warps of the CTA during load assignment. We initialize the sorted matrix such that the first 32 rows are those from the original matrix with the lowest warp load. The next 32 rows are those from the original matrix that minimizes Hamming distance to a previously sorted row

located 32 rows earlier (*i.e.* row[i] is selected to minimize the distance from row[i-32]). This incremental assignment of rows from the original matrix to the sorted matrix continues until every row has been added. At the end, each warp has been assigned rows with similar access patterns to dense matrix B.

CTA-Aware Sort. To increase inter-warp data re-use, we re-order rows to minimize distances between adjacent rows in the matrix. Adjacent rows are processed in parallel by the CTA's warps. We re-order rows to create clusters of $N_w = 32$ consecutive rows, corresponding to parallel work groups, with similar cache access patterns. We incrementally add rows to clusters, selecting the row from the original matrix most similar to the previously added row in the sorted matrix. Because clusters of consecutive rows in the matrix are assigned to parallel warps, minimizing Hamming distance between adjacent rows increases the likelihood that cache lines requested by one warp are re-used by another warp running at the same time.

We are not the first to study SpMM locality. Section VI provides a detailed discussion of related work but two particular studies are noteworthy here. First, Pichel et al. improves locality by measuring and minimizing global accumulate distance, a count of aligned cache lines between two rows [10]. Our approach is comparatively finer-grained. It further accounts for the likelihood of requesting non-cached blocks or evicting recently cached blocks and minimizes the distance in these measures for two rows that are likely to be executed within a short time frame.

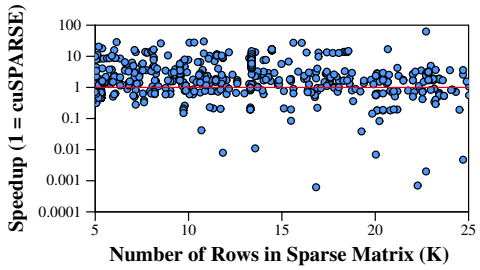
Second, Jiang et al. improves locality by re-ordering rows in a CSR matrix using the ratio of identical non-zero columns to the total non-zero columns as a measure of similarity between rows [11]. Again, our approach differs in granularity, tracking columns at coarser cache block granularity and assessing row similarity with more detailed re-use statistics rather than column patterns.

C. Hybrid Permutations

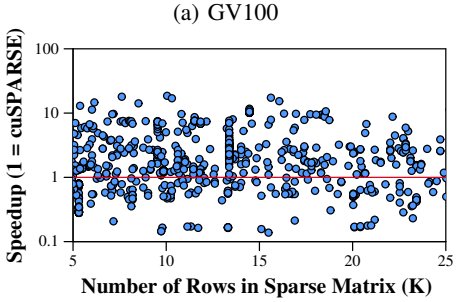
Hybrid strategies combine benefits from load balancing and cache-aware sort.

Hybrid-1 Sort adjusts load balancing strategies by considering data locality. When several rows have equal warp loads, we use locality to break ties rather than simply picking the first of these rows. We break ties by selecting the row with the smallest distance to the adjacent row, balancing load while increasing data re-use across a CTA's parallel warps.

Hybrid-2 Sort adjusts cache-aware strategies by considering warp load balance. When several rows are equally similar to the previously added row's bit mask, we break ties based on secondary considerations. We consider three variants. First, Hybrid-2.1 uses CTA-aware Sort and breaks ties based on warp load. Second, Hybrid-2.2 uses CTA-aware Sort and breaks ties based on distances between intra-warp bit masks. Third,

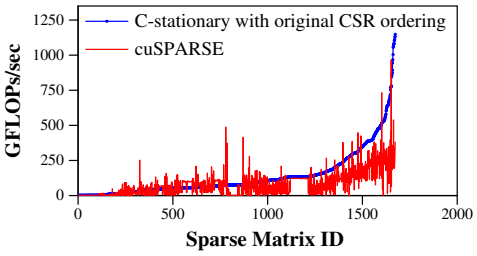


(a) GV100

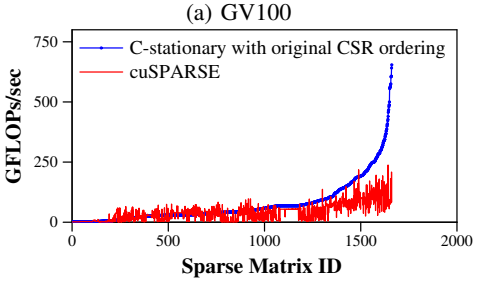


(b) 2080 Ti

Fig. 3: Speedup of C-stationary SpMM baseline kernel with original CSR ordering against NVIDIA cuSPARSE.



(a) GV100



(b) 2080 Ti

Fig. 4: Throughput of C-stationary SpMM baseline kernel with original CSR ordering against NVIDIA cuSPARSE over 1,688 matrices.

Hybrid-2.3 uses Warp-aware Sort and breaks ties based on warp load.

IV. Assessing Performance Potential

Dataset. We measure SpMM execution time, before and after permutations, for 1,688 matrices from the SparseSuite Matrix Collection that include multiple categories and diverse sparsity patterns from real problems [12]. We exclude matrices that are from duplicate categories, are non-square, or have more than 30K rows, to fit the dense matrices B and C with the same dimensions as A in GPU main memory.

Platform. We perform single-precision SpMM on two GPUs. First, the **NVIDIA Tesla GV100** has 5,120 FP32

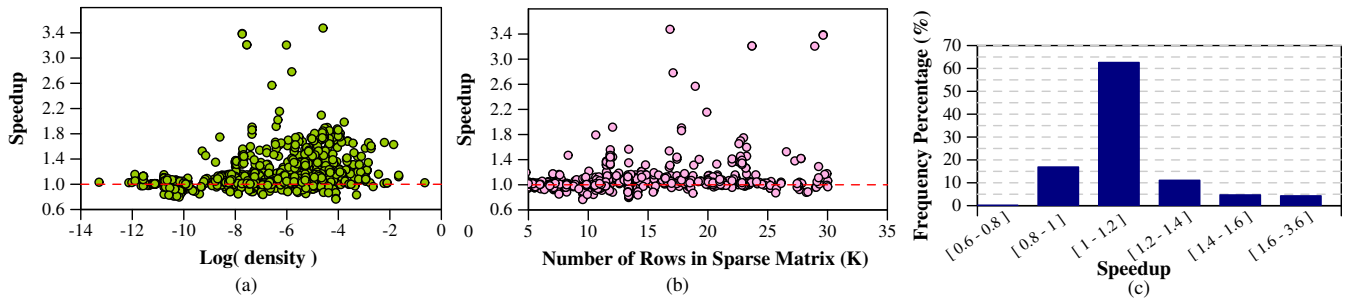


Fig. 5: Speedup after a load balancing permutation (LPT Sort) compared to original CSR on GV100. (a) plotted over density, (b) plotted over matrix size, (c) histogram of speedup values.

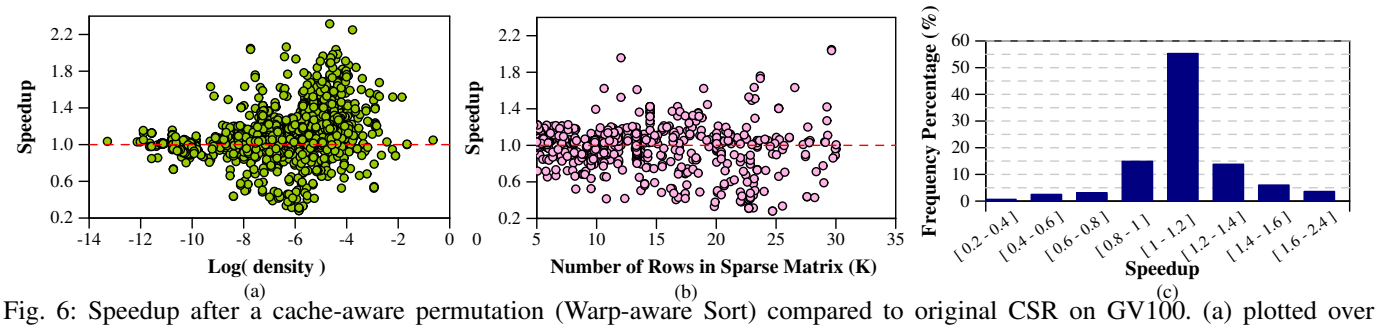


Fig. 6: Speedup after a cache-aware permutation (Warp-aware Sort) compared to original CSR on GV100. (a) plotted over density, (b) plotted over matrix size, (c) histogram of speedup values.

CUDA cores operating at 1.53 GHz. It has up to 96 KB of shared memory per SM, 6 MB L2 cache, and 16 GB HBM2 main memory with a 4 Kb bus providing 870 GB/s of bandwidth. Second, the **NVIDIA RTX 2080 Ti** has 4,352 FP32 CUDA cores operating at 1.35 GHz. It has a 5.5 MB L2 cache and an 11 GB GDDR6 main memory with a 352 b memory bus providing 616 GB/s bandwidth. We compile our CUDA codes with nvcc in CUDAToolkit v10.2 with the -o3 compiler flag.

Baseline. We use the C-stationary SpMM implementation [8], because it outperforms the state-of-the-art baseline, the NVIDIA cuSPARSE library [13]. The cuSPARSE library is a general sparse linear algebra library suite with a sophisticated API. We try both modes that cuSPARSE offers for SpMM with CSR and compare our C-stationary baseline against the better performing one. Other recent work has also used fixed-function SpMM implementations which outperform cuSPARSE [8] [11]. Fig. 3 indicates that our C-stationary SpMM outperforms cuSPARSE by 1.67-1.89×, on average, for our matrices and GPUs. Fig. 4 also reports raw throughput, in GFLOPs, for the two SpMM implementations.

A. Performance Determinants

Fig. 5 evaluates performance for one representative load-balancing permutation technique, reporting speedups on the GV100 from LPT Sort. Each point corresponds to one sparse matrix. In Fig. 5(a), matrices are ordered by their densities (x-axis) and speedups are reported relative to SpMM on the original CSR matrix (y-axis). Fig. 5(b), however, reports the speedups relative to the matrices' number of rows (x-axis). LPT Sort balances warp load and improves performance by up to 3.5×. But LPT is counterproductive for several matrices, harming performance by 0.8× in the worst case. Fig. 5(c) indicates more than 17% of matrices are harmed by LPT Sort.

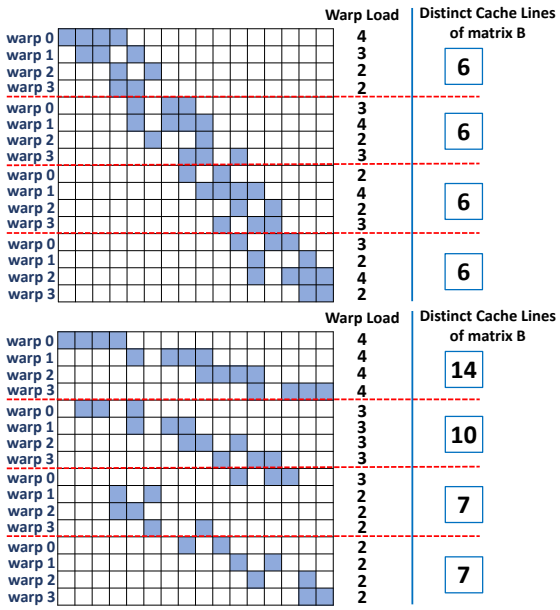


Fig. 7: Example. CSR original order (top). LPT sorted CSR (bottom).

We diagnose that load balancing permutations can harm performance by degrading data locality. Fig. 7(top) presents sparse matrix A with a diagonal non-zero pattern. Square cells in each row represent 32 consecutive matrix columns corresponding to one cache line of the dense matrix B during SpMM. Colored cells include a non-zero element. Horizontal red lines set hypothetical boundaries on rows that are sequen-

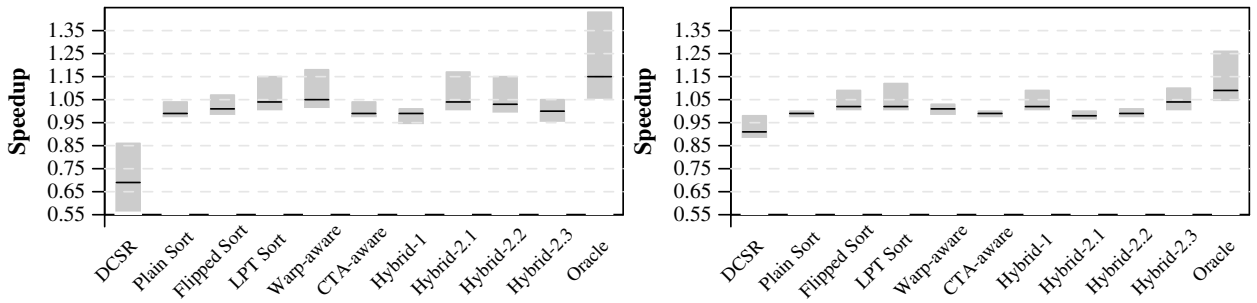


Fig. 8: Box plot for SpMM speedup distribution of different CSR based permutations. GV100 (left). 2080Ti (right)

tially assigned to warps within the CTA (*i.e.,* first row of each boundary is assigned to $warp_0$ in CTA). Numbers on the right indicate *warp load* and the number of distinct cache lines from matrix B requested by the CTA.

In Fig. 7 (bottom), LPT Sort improves load balance across warps 0-3 by changing their *warp load* from 12, 13, 10, 10 in the original matrix (top) to 12, 11, 11, 11 in the sorted matrix (bottom). But this permutation increases the distance between rows with common cache lines (*i.e.,* rows with colored cells in identical columns). As a result, the CTA computes on these rows at different times and misses the opportunity to re-use cached data. The number of distinct cache lines requested within each boundary increases after LPT sort, and the total number of distinct cache lines requested from dense matrix increases from 24 with original CSR to 38 with LPT sorted CSR.

Fig. 6 reports performance after a representative cache-aware permutation. Warp-aware Sort accounts for data locality and improves performance by up to 2.3×, but it harms performance for over 20% of matrices and by 0.3× in the worst cases as seen in Fig. 6(c). Fig. 7 reveals how re-ordering rows for better data locality could imbalance load and harm performance.

In sum, significant opportunities exist to improve performance by permuting matrix rows, but a considerable number of matrices may suffer slowdowns if the wrong permutation is chosen. Figs. 5-6 suggest there is no clear relationship between any single generic metric, such as matrix size or density, with permutation speedups. We cannot trivially determine which matrices will benefit from a permutation technique. Diversity in matrix sparsity, complex trade-offs between load distribution and data re-use, and several sort techniques make finding the proper permutation for each matrix a challenge.

In the rest of this section, we analyze the performance for varied permutation techniques. We evaluate combinations of multiple techniques to overcome disadvantages of using one permutation alone. In Section V, we provide a predictive model that associates permutation techniques' performance with features that describe matrix sparsity, enabling intelligent selection of the best technique for each matrix.

B. Performance Distributions

We measure speedup after each permutation for all 1,688 matrices on two different GPUs: the GV100 and 2080 Ti. A permutation technique's performance depends on sparsity patterns and varies across matrices. Fig. 8 uses box plots to visualize the performance distribution across sparse matrices for each technique. The bottom and top of the box indicate the first and third quartile, respectively. The middle line indicates the second quartile (i.e., the median). Table. I reports summary statistics.

TABLE I: Performance summary of various permutation techniques over 1,688 datasets on GV100 and 2080Ti.

	GV100 Speedup			2080Ti Speedup		
Permutation	Max	Mean	Min	Max	Mean	Min
DCSR	13.4×	0.85×	0.5×	20.2×	1.15×	0.7×
Plain Sort	4.1×	1.02×	0.6×	2.5×	1.0×	0.7×
Flipped Sort	3.9×	1.04×	0.8×	2.4×	1.06×	0.7×
LPT Sort	3.5×	1.10×	0.8×	1.9×	1.06×	0.6×
Warp-aware Sort	2.3×	1.07×	0.3×	2×	0.99×	0.3×
CTA-aware Sort	3.9×	1.01×	0.3×	2.2×	1.0×	0.7×
Hybrid-1 Sort	4.2×	1.0×	0.3×	2.4×	1.06×	0.5×
Hybrid-2.1 Sort	4.1×	1.10×	0.4×	2.3×	0.99×	0.7×
Hybrid-2.2 Sort	4.1×	1.09×	0.3×	2.3×	1.0×	0.6×
Hybrid-2.3 Sort	2.4×	0.97×	0.25×	2.1×	1.06×	0.3×
Oracle	13.5×	1.4×	1×	20.2×	1.37×	1×

No single permutation technique works well for all matrices. Each technique benefits only a subset of matrices, but those benefits are substantial. Most techniques report median values near one, the result of offering a nearly even mix of speedups and slowdowns across 1,688 sparse matrices. For instance, DCSR improves performance by up 13.4× and 20.2× on the GV100 and 2080 Ti, respectively. But its speedup is less than 0.85× and 0.98× for 75% of matrices on the GV100 and 2080 Ti. LPT, Warp-aware, and hybrid permutations on the GV100 and LPT and hybrid permutations on the 2080 Ti report larger medians and third quartiles, indicating versatility and greater applicability across diverse matrices.

We analyze the upper bound on performance achievable when an oracle identifies the best technique for each matrix. Fig. 8 and Table I report the oracular performance distribution across matrices. The median speedups are 1.15× and 1.09×, and 75% of the matrices report a speedup greater than 1.05× and 1.04× on the GV100 and 2080 Ti, respectively. The oracle's performance is superior to all individual permutation techniques, indicating advantages of combining multiple permutation techniques and selecting the right permutation for each matrix. The original CSR order is also considered by the oracle. Note that the oracle is 1.37-1.4× faster than our baseline SpMM with original CSR. It is also 2.31-2.65× faster than cuSPARSE, on average.

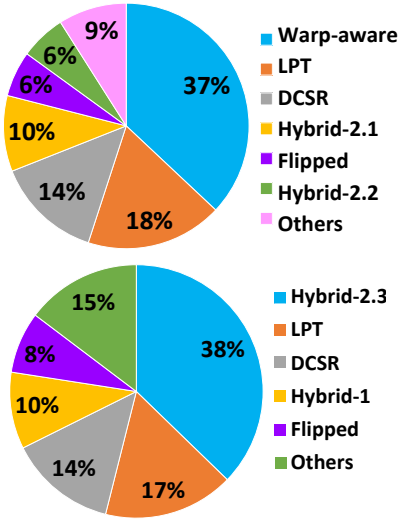


Fig. 9: Summary of the chances of different permutation policies to provide the best performance over 1,688 matrices. GV100 (top). 2080Ti (bottom)

C. Oracular Performance

The oracular analysis motivates a portfolio of permutation techniques to support diverse matrices. Fig. 9 indicates how often a technique is most preferred for each of 1,688 matrices. On the GV100, Warp-aware Sort is most popular, and 37% of matrices benefit most from this permutation. But a mix of six techniques are required to ensure that 90% of matrices are supported by their most preferred technique.

Fig. 10 illustrates how oracular performance increases with the number of permutation techniques available. When an oracle can choose between two permutations (including the original matrix), SpMM performance improves by 33% on average. The two best permutations are DCSR and Hybrid-2.1 Sort, which uses CTA-aware Sort and breaks ties based on distances between intra-warp rows. When the oracle can choose between even more permutation techniques, SpMM performance improves further. The analysis indicates diminishing marginal returns when five techniques are available. Thus, flexibility and choice benefit SpMM performance across diverse matrices.

While oracular performance gains are considerable, selecting the most preferred technique for each matrix prior to execution is non-trivial. In Section V, we propose predictors that associate permutation performance with sparsity features to produce an efficient permutation choice for each matrix.

V. Optimizing Permutation Techniques

Picking the right permutation technique for each matrix, rather than using a single technique for all matrices, is crucial for performance. The challenge is predicting the most suitable technique for a given matrix. First, we select features that summarize matrix sparsity characteristics effectively and show that these features are associated with performance from permutation techniques. Second, we design a model that predicts the best permutation technique for a matrix given its sparsity features.

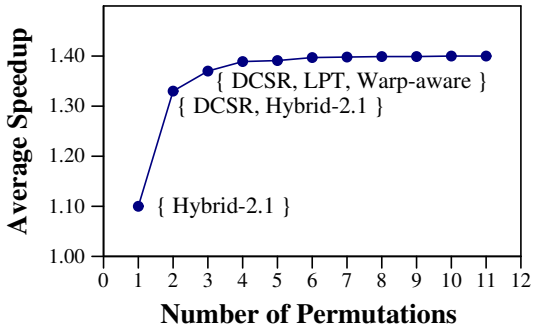


Fig. 10: Oracular Speedup on GV100

TABLE II: Sparse Matrix Feature Set

Feature	Definition
Matrix	
n_{row}	number of rows
n_{col}	number of columns
density	$nnz \times [n_{\text{row}} \times n_{\text{col}}]^{-1}$
nnz-per-row	number of non-zero elements (minimum, mean, and maximum across rows)
nnz-blocks-per-row	number of blocks of N_w consecutive columns with at least one non-zero (minimum, mean, and maximum across rows)
Load Balance	
warp load	accumulated warp load ($\left\lceil \frac{nnz_R}{N_t} \right\rceil$) for all rows assigned to a warp (minimum, mean, and maximum across warps)
Data Locality	
same-cache-lines	number of requests for same cache line of dense matrix B (minimum, mean, and maximum across cache lines)
distinct-cache-lines-per-warp	number of distinct cache lines requested per warp (minimum, mean, and maximum across warps)
total-cache-lines-per-warp	number of cache lines (distinct or not) requested per warp (minimum, mean, and maximum across warps)
adjacent-vector-distance	distance between bit masks for adjacent rows (minimum, mean, and maximum across adjacent rows)

A. Sparsity Features

We identify sets of features, for a sparse matrix, that predict the most suitable permutation technique for the (D)CSR matrix format. These features reflect insights from our performance analysis and capture sparsity patterns that affect load balance and data locality. These features also span the hardware-software interface, enabling predictors that account for interactions between the sparse matrix, parallel workers, and cache organization.

Table II lists features that describe the matrix, measure load balance, and measure data locality. Most features are expanded to describe three summary statistics (*e.g.*, minimum, mean, maximum) to produce a full set of twenty-four features. These features can be quantified, offline and efficiently, by examining matrix structure. First, matrix features quantify the distribution of non-zero elements and include measures such as the number of rows and average number of non-zeros per row. Second, load balance and data locality features describe SpMM interactions with the underlying GPU. These features include platform parameters such as the number of warps and the cache line size.

Together, these features supply information to predict the row permutation that performs best. For instance, if the *warp load* features across warps indicate a fairly balanced load (*e.g.*, similar values for minimum and maximum warp load), the CSR matrix's original sparsity pattern offers little potential performance gain for load balancing permutation techniques to exploit and cache-aware permutation techniques may be more beneficial.

B. Optimizing Performance

We construct models to solve a classification problem, in which sparse matrix features map to the permutation technique that minimizes execution time. We consider the twenty-four features in Table II. Fig. 10 indicates diminishing marginal returns beyond three permutation techniques and we consider four classes: DCSR, LPT, Warp-Aware, and the original CSR matrix order. This fourth class is required because some matrices see only slowdowns from row permutations, and keeping the original order is preferred. Note that our model imposes no limit on the number of classes.

We approach this classification problem in two steps. First, we predict execution time of permutation classes using attributes of the sparse matrix in Table II (regression). We construct a neural network with four execution time outputs, one for each permutation technique. The network uses two fully connected hidden layers with 64 and 32 neurons with L2 regularization. Second, given the predicted execution times, we select the permutation technique that minimizes execution time as the preferred choice (classification).

Figs. 5-6 indicate no clear linear relationship between features, such as density, and speedups from permutation techniques. Neural networks can describe non-linear relationships and interactions between features, making them attractive models for modeling SpMM performance, but other models might be suitable as well.

We classify in two steps because predicting raw execution time might be desirable. But also we evaluate our predictor by directly training a classifier on the same set of sparsity features to classify each matrix in a permutation class. Instead of regression values for executions times, we use softmax in the output layer and select the permutation class with highest probability as the classification result. This approach achieves average speedups within 4-5% of the oracle's.

C. Training

We implement our predictor with the Keras library and run it with the Tensorflow backend on an x86 machine. We use 80% of the 1,688 matrices to train the model and 20% to test. The loss function is mean absolute percentage error.

Fig. 11 shows how often each of the top four permutation techniques is most preferred by a sparse matrix on the GV100 and Ti 2080 within our training set. The population distribution of matrices among these four permutation classes is highly imbalanced. Therefore, a normal training procedure with equal importance for all samples from the training set is likely to lead us to a biased model.

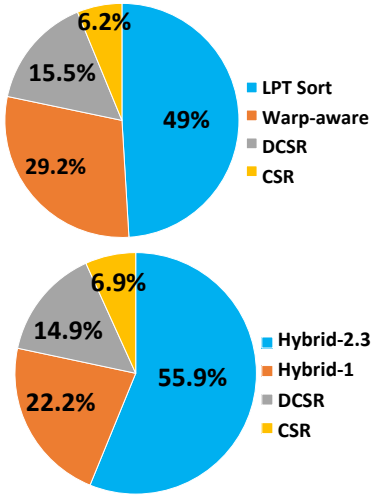


Fig. 11: Summary of the chances of four permutation policies to provide the best performance over training set on GV100 (top) and Ti 2080 (bottom).

To avoid bias, we assign a set of weights to classes that are inversely proportional to class population in Fig. 11. For instance, as the population of the LPT class is 1.7× the population of the Warp-aware class, the weight assigned to Warp-aware is 1.7× larger than LPT's. Weights define the value of each sample during training and are accounted for in the loss function. Minority classes' information is preserved due to their their greater weights in the loss function. Such weighting is common when training with imbalanced data.

D. Accuracy

For each matrix in the test set, we evaluate four predicted execution times corresponding to the four candidate permutation techniques (*i.e.,* CSR, DCSR, LPT, and Warp-aware on the GV100). Each prediction incurs an error compared to true execution time. We use absolute error percentage $(\frac{\text{True Value - Predicted Value}}{\text{True Value}} \times 100)$ to quantify model accuracy.

Fig. 12 summarizes regression accuracy by plotting an empirical cumulative distribution function. The x-axis presents error and the y-axis presents how often predictions achieve less than that error. Fewer than 30% of matrices report errors below 2% and approximately 70% report errors below 10%. Although predictions are inaccurate for a considerable number of matrices, the regression stage is intended to support the following optimization stage, which finds the best permutation technique for each matrix. As long as execution time predictions are relatively accurate, they may be sufficient for our optimization problem.

To evaluate optimization effectiveness, Fig. 13 presents classification results in a confusion matrix. Rows and columns correspond to optimal and predicted classes, respectively. Each sparse matrix is accounted for in the row for its optimal class and column that our model predicts for it. Each cell reports two values—the number of matrices assigned to it and the average performance loss for those matrices. Loss is the difference between model- and oracle-selected permutation. Note the the test set has 24, 57, 100, and 157 matrices for classes CSR, DCSR, Warp-aware, and LPT. If all matrices were

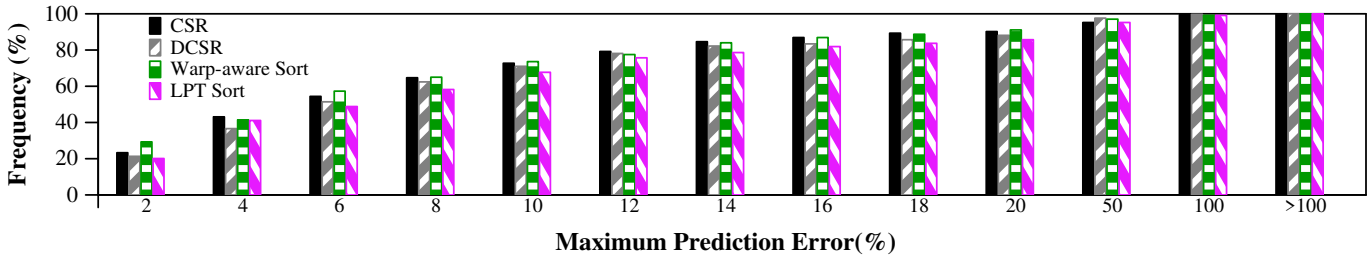


Fig. 12: Cumulative distribution of absolute errors on predicted execution time for each of the 4 classes. 70% of our predictions report errors below 10%.

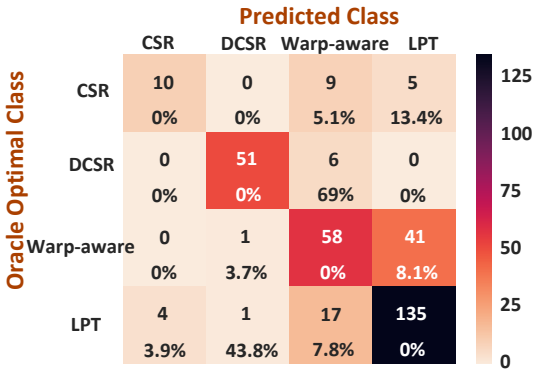


Fig. 13: Confusion matrix for classification using our predictor model on GV100.

perfectly classified, only the diagonal cells in the confusion matrix would hold non-zero values. The confusion matrix shows 82% of matrices are classified correctly using regression predictions. For example, 135 of the 157 LPT matrices are classified correctly while 4, 1, and 17 of its matrices are mistakenly classified under CSR, DCSR, and Warp-aware, respectively. However, even this analysis is conservative because the classifier may select a sub-optimal technique yet achieve a large fraction of the optimal technique's performance gain. Cells with high performance loss include very few samples. Although our model classifies one matrix originally from LPT class as DCSR class, incurring a 43% performance loss, most mis-classifications incur less than 10% performance loss relative to oracular performance.

Our proposed classifier performs nearly as well as an oracle. The neural network selects the permutation technique and improves the test set's performance by 1.37×, on average, over using the original matrix representation. An oracle that chooses the right technique for each matrix could improve performance by up to 1.41× on the GV100. Training our predictor on the Ti 2080 also achieved average gains within 4% of oracle gains. However, these averages obscure performance for individual matrices and we must also evaluate the performance distribution.

Fig. 14 presents the distribution of losses on matrices in the test set. Loss is zero when the model and oracle select the same permutation, is modest when the model selects a sub-optimal permutation that performs well, and is high when the model selects the wrong permutation strategy entirely and

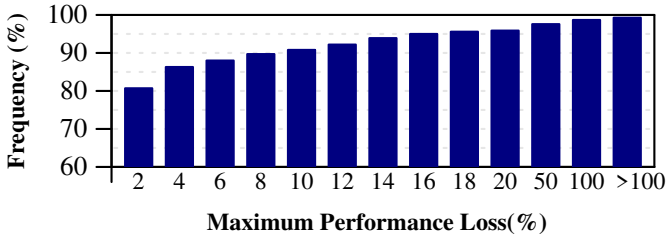


Fig. 14: Histogram of performance loss due to misclassification compared to oracle choice

causes a significant loss. The figure characterizes losses across tested matrices in a histogram. Over 86% of tested matrices perform within 4% of oracular performance and over 90% of them within 10%. On average, matrices perform within 3.8% of oracular performance.

Using our sparsity feature set, our predictor achieves high accuracy with a simple network architecture, which results in fast training. Inference to predict the best permutation strategy for each matrix in the test set also happens in real-time. High accuracy, low cost, and considerable gains, compared to blindly using one permutation for all matrices, strongly motivate using our prediction technique to select the right permutation for each matrix's unique sparsity pattern.

E. Sensitivity

Model Features. Prior approaches model the effect of different sparse matrix representations using only matrix characteristics and neglecting platform parameters [14] [15]. In contrast, we model the effect of matrix row permutations, for a given representation, and require additional features. We train our model twice, once with our matrix features alone and again with the complete feature set.

The model trained with only matrix characteristics selects techniques that improve performance by only 1.10× and incurs 31% performance losses relative to the oracle. The model trained with the complete feature set selects techniques that improve performance by 1.37× and incurs only 3.8% performance loss. Considering both hardware and software parameters, such as load balance and data locality, is crucial for classifying permutation techniques.

Neural Network Topology. The neural network's hyperparameters affect model accuracy and learning rate. A full exploration of the predictor's topology space, such as the number of hidden layers and the number of neurons in each layer, is beyond this study's scope. But we do assess model sensitivity to hyperparameters and use rules-of-thumb to constrain the hyperparameter space. We begin with one hidden

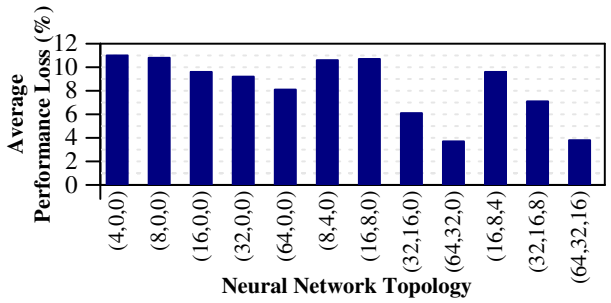


Fig. 15: Sensitivity of predictor accuracy to neural network topology. Topology (a,b,c) represents 3 hidden layers with a, b, and c neurons.

layer and examine the effects from more neurons and/or more layers on prediction accuracy using L2 regularization. Fig. 15 reports average performance loss relative to an oracle. Adding a second layer improves accuracy, but adding a third did not. Given these outcomes, we use two hidden layers with 64 and 32 neurons, respectively, to obtain accurate predictions with relatively few parameters.

VI. Related Work

Matrix Permutations. Reordering sparse matrix rows or columns improves efficiency and bandwidth [16] [17] [18]. Examples include classical techniques like Approximate Minimum Degree [18], Reverse Cuthill–McKee [16] [19], and METIS [17] for SpMV on CPUs. Pichel et al. define distance functions between matrix rows/columns [20] [21] and find a graph traversal that minimizes the distance between adjacent rows, outperforming classical reordering approaches. Pichel et al. also extend reordering to GPUs [10], outperforming all classic reordering techniques. Our cache-aware strategies are inspired by Pichel et al.

Our approach differs from prior work. First, we define a different distance metric, modeling not only the likelihood of re-using a cached block but also the event of requesting a non-cached block or evicting a recently cached block. Second, we consider code structure and thread organization in our SpMM kernel, reordering rows to minimize differences between rows that are likely to compute close in time.

Jiang et al. cluster and reorder rows to improve locality [11], measuring similarity based on the ratio between identical non-zero columns to all non-zero columns among two rows. Unfortunately, two pairs of rows with the same ratio can differ significantly in the number of reused cache lines. The ratio also misses locality when non-zero columns are not identical yet belong to the same cache block, each of which holds 32 columns. We address this problem with bit vectors that track access patterns for cache blocks. Finally, prior studies do not account for GPU load and cannot meet the needs of diverse matrices when no single permutation technique is best suited for all matrices. We are the first to develop a broad set of permutation techniques for GPUs.

Sparse Linear Algebra. How kernels are mapped to parallel processing elements influences load balance and data re-use. Load balance can be improved by distributing non-zeros over elements more effectively [22] [23] [24] or selecting an optimal kernel given the number of non-zeros in rows [9]. Different sparse matrices have varied preferences for code structure, incurring the costs of parameter tuning and kernel switching; we do not tune SpMM code in this paper.

Sparse Matrix Format. Data structure and matrix format impact performance and storage requirements. CSR is most common but alternatives have been proposed [25] [26]. Some data formats are particularly efficient for matrices with specific sparsity patterns (*e.g.,* diagonal, symmetric) [27] [28] [29]. Alternative formats and kernels can outperform CSR and the output-stationary kernel for some matrices, but they often suffer from significant pre-processing time or additional metadata storage. Furthermore, as many standard libraries and kernels are implemented for CSR, using other formats incurs additional overheads for online conversion. Unlike prior work, we do not modify matrix format or kernel implementation, avoiding matrix conversion costs.

Machine Learning. Machine learning methods can select matrix data structures using sparsity features. Decision trees use sparsity features to choose between COO, ELL, CSR, and HYB representations [15]. Support vector machines use simpler features yet account for non-linear performance topologies [14]. CNNs treat sparsity patterns as images and predict the best data structure [30] [31].

We are the first to predict the effects of row permutations for a given data structure, CSR. We use richer input features and less complex models to capture hardware-software interactions on the GPU. Simpler, fully-connected networks with only two hidden layers are sufficient for selecting row permutation techniques. Our features are easily calculated and avoid overheads associated with converting matrices into images for input into CNNs. Our model requires less parameter tuning and less training data, and it is faster in training.

VII. Conclusion

We design sparse matrix row permutations that improve load balance, data re-use, or both for SpMM computation on GPUs. We propose a heterogeneous permutation strategy to choose the best performing permutation for each matrix given its sparsity pattern. The strategy offers significant performance gains compared to using a single permutation for all matrices. We develop a predictive model that finds the best permutation for each matrix using its sparsity features and achieves 96% of oracle gains.

Acknowledgment

We thank Jason Clemons and Iuri Frosio for their insightful feedback on machine learning methods. This work was supported in part by the National Science Foundation grants CCF-2002737 and CNS-1822085.

References

[1] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM, 2000.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep., 1999.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012.

[4] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on GPUs," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 66–79.

[5] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[6] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[7] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," in *Proceedings of the ACM/IEEE Conference on Supercomputing*. IEEE, 1999, pp. 30–30.

[8] D. Fujiki, N. Chatterjee, D. Lee, and M. O'Connor, "Near-memory data transformation for efficient sparse matrix multi-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–17.

[9] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on GPUs for graph applications," in *Proceedings of the International Conference for High performance Computing, Networking, Storage and Analysis*, 2014, pp. 781–792.

[10] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, "Optimization of sparse matrix–vector multiplication using reordering techniques on GPUs," *Microprocessors and Microsystems*, vol. 36, no. 2, pp. 65–77, 2012.

[11] P. Jiang, C. Hong, and G. Agrawal, "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020, pp. 376–388.

[12] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[13] "The API reference guide for cuSPARSE, the CUDA sparse matrix library," https://docs.nvidia.com/cuda/archive/10.2/cusparse/index.html.

[14] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass SVM for SpMV on GPU," in *45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 496–505.

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

[16] E. Cuthill, "Several strategies for reducing the bandwidth of matrices," in *Sparse Matrices and Their Applications*. Springer, 1972, pp. 157–166.

[17] G. Karypis and V. Kumar, "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," *University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN*, 1998.

[18] P. R. Amestoy, T. A. Davis, and I. S. Duff, "An approximate minimum degree ordering algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 17, no. 4, pp. 886–905, 1996.

[19] E.-J. Im and K. A. Yelick, "Optimizing sparse matrix vector multiplication on SMPs," in *PPSC*, 1999.

[20] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, "Improving the locality of the sparse matrix-vector product on shared memory multiprocessors," in *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2004, pp. 66–71.

[21] J. C. Pichel, D. E. Singh, and J. Carretero, "Reordering algorithms for increasing locality on multicore processors," in *10th IEEE International Conference on High Performance Computing and Communications*. IEEE, 2008, pp. 123–130.

[22] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 58.

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

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

[25] M. Maggioni and T. Berger-Wolf, "Optimization techniques for sparse matrix–vector multiplication on GPUs," *Journal of Parallel and Distributed Computing*, vol. 93, pp. 66–86, 2016.

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

[27] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*, 2009, pp. 1–11.

[28] J. Godwin, J. Holewinski, and P. Sadayappan, "High-performance sparse matrix-vector multiplication on GPUs for structured grid computations," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. ACM, 2012, pp. 47–56.

[29] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 115–126.

[30] J. C. Pichel and B. Pateiro-López, "A new approach for sparse matrix classification based on deep learning techniques," in *International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 46–54.

[31] J. C. Pichel and B. Pateiro-López, "Sparse matrix classification on imbalanced datasets using convolutional neural networks," *IEEE Access*, vol. 7, pp. 82 377–82 389, 2019.