CB-SpMV:A Data Aggregating and Balance Algorithm for for Cache-Friendly Block-Based SpMV on GPUs

Xing Cong
Beihang University
BeiJing, China
congxing@buaa.edu.cn

FuKai Sun
Beihang University
BeiJing, China
sunfukai@buaa.edu.cn

YiFan Chen
Beihang University
BeiJing, China
chenyifan@buaa.edu.cn

Chenhao Xie*
Beihang University
BeiJing, China
xiechenhao@buaa.edu.cn

Yi Liu
Beihang University
BeiJing, China
yi.liu@buaa.edu.cn

Depei Qian
Beihang University
BeiJing, China
depeiq@buaa.edu.cn

Abstract

Sparse matrix-vector multiplication (SpMV) is crucial in computational science, engineering, and machine learning. Despite substantial efforts to improve SpMV performance on GPUs through various techniques, issues related to data locality, hardware utilization, and load balancing persist, leaving room for further optimization. This paper presents CB-SpMV, a cache-friendly SpMV optimization algorithm, using a novel data convergent and adaptable 2D blocking structure. The matrix in CB-SpMV is divided into independent sub-blocks, with virtual pointers aggregating different types of intra-block data for better cache-level data locality. To enhance hardware utilization, a block-aware column aggregation strategy and the selection of sub-block formats are proposed to accelerate computation and adapt to varying sparse matrices. Finally, an inter-block load-balancing algorithm is designed to ensure efficient workload distribution across thread blocks. Experimental evaluations on 2,843 matrices from the SuiteSparse Collection show that CB-SpMV significantly improves cache hit rates and achieves average speedups of up to 3.95× over state-of-the-art methods like cuSPARSE-BSR, TileSpMV, and DASP on NVIDIA A100 and RTX 4090 GPUs.

CCS Concepts

• **Computer systems organization** → **Single instruction, multiple data**; • **General and reference** → **Performance**; • **Computing methodologies** → **Parallel programming languages**.

Keywords

Data Structure and Memory Optimization, Blocked Sparse Matrix, SpMV on GPUs

*Corresponding author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725746

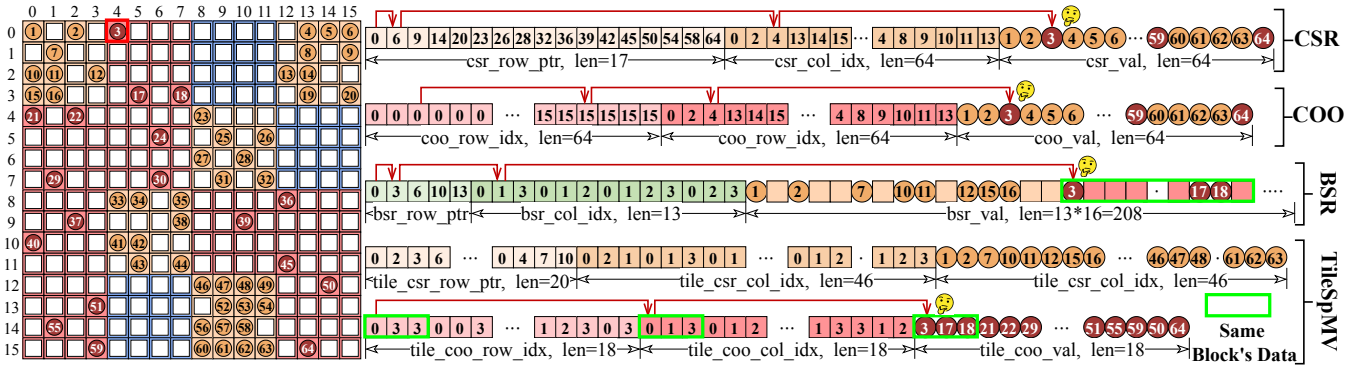
ACM Reference Format:
Xing Cong, FuKai Sun, YiFan Chen, Chenhao Xie, Yi Liu, and Depei Qian. 2025. CB-SpMV:A Data Aggregating and Balance Algorithm for for Cache-Friendly Block-Based SpMV on GPUs. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3721145.3725746

1 Introduction

Sparse matrix operations, a cornerstone of computational science and engineering, are essential in numerical simulations, data analysis, and machine learning. Sparse matrices, characterized by irregularly distributed non-zero elements, pose challenges such as poor memory locality and load imbalances in parallel computing, which hinder computational efficiency. Among these operations, sparse matrix-vector multiplication (SpMV) is one of the most fundamental and widely studied kernels. Research has focused on optimizing memory access through novel storage formats (e.g., CSR5[33], LSRB-CSR[32], TileSpMV[39]), dynamic storage format selection(e.g., Alphasparse [16]), and leveraging Tensor Core hardware(e.g., DASP[34]). Despite these advancements, issues related to cache efficiency, hardware utilization, and load balancing persist, leaving room for further optimization.

Regarding data locality, both the widely studied CSR format and the state-of-the-art block-based format[24, 39, 40] store the coordinate array and value array separately. This design necessitates frequent access to the coordinate array during SpMV operations, which involves large memory jumps to retrieve elements from the value array. Such a skipping storage structure significantly reduces the hit rates of the GPU's L1 and L2 caches. A detailed analysis of this issue is provided in the following section 2.2.

Regarding hardware utilization, although previous research works[39] introduce block-based SpMV for data locality and a variety of formats to handle sub-blocks with different sparsity levels, it still suffers from inadequate hardware utilization on GPUs in the case of highly sparse sub-blocks. For instance, when a sub-block contains few non-zero elements, the GPU still needs a warp, the basic thread unit of GPUs, to process, which leads to most of the warp threads remaining idle. Such a similar situation also exists for Tensor Core accelerated SpMV[34]. In the data layout of DASP, the 16×8 TCU(Tensor Core Unit) produces 256 values per cycle, but only the 16 diagonal values are relevant for SpMV, limiting the tensor core's effective utilization.



(a) An example matrix A of size 16-by-16 (b) The representation of matrix data in GPU memory

Figure 1: (a) illustrates a 16×16 example matrix divided into 4×4 sub-blocks, with 13 non-zero sub-blocks highlighted. (b) depicts the data layout of the matrix in GPU global memory across four sparse storage formats (CSR, COO, BSR, TileSpMV), along with the memory access patterns for retrieving the third non-zero element located at (0,4) in the example matrix.

Regarding load balance, block-based methods usually process each sub-block at the warp level. However, the number of non-zero elements within each sub-block varies significantly. Consequently, the total number of non-zero elements that each thread block (generally composed of eight warps [13]) needs to process can differ substantially. This imbalance results in uneven workloads across streaming multiprocessors (SMs), ultimately leading to performance degradation. To accelerate SpMV calculation by improving the data locality, hardware utilization, and load balance simultaneously, this paper proposes Cache-friendly Block-based SpMV, **CB-SpMV**, including a novel data convergent and adaptable 2D block structure and a series of optimization policies: **Intra-Block Data Aggregating** to unite the different storage data formats within the sub-block; **Computation Adaptation** for handling blocks with varying sparsity, and **TB(Thread Bolck)-Load-Balance** to ensure equitable workload distribution across thread blocks.

The design of CB-SpMV stems from a key insight: after partitioning a matrix into sub-blocks, the data within each sub-block is independent and self-contained, with the coordinates of non-zero elements relative to the sub-block itself. Leveraging this property, CB-SpMV was developed to treat each sub-block as an independent unit, compactly storing the different types of coordinates and non-zero values via format uniting and an efficient virtual pointer structure. This design significantly reduces scattered memory access, enhancing data locality and improving L1 and L2 cache hit rates. However, achieving optimal performance with CB-SpMV introduces additional challenges, particularly in maintaining block independence while addressing issues such as load balancing and the trade-off between parallelism and utilization.

To address the hardware utilization challenge for varying sparsity levels, we apply a block-aware column aggregation strategy for sparse sub-blocks and choose either CSR or Dense formats for denser sub-blocks, optimizing computation efficiency. Additionally, a parallel load balancing algorithm leveraging priority queues ensures equitable workload distribution, further enhancing performance. Experimental evaluations on 2,843 matrices from SuiteSparse demonstrate the effectiveness of CB-SpMV. On the RTX 4090 GPU, the method increases L1 and L2 cache hit rates by 82% and 19%, respectively, compared to TileSpMV[39], and achieves improvements of 15.62× and 10.05× over cuSPARSE-BSR[14]. It also delivers speedups of 2.95×, 3.06×, and 2.76× over cuSPARSE-BSR, TileSpMV, and DASP[34]. The contributions of this paper are summarized as follows:

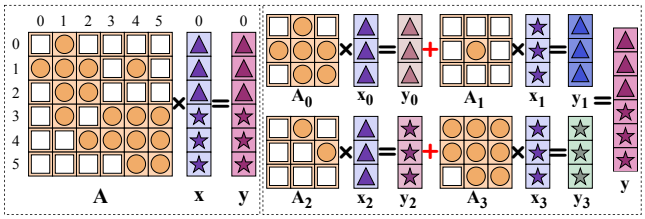
- **2D Blocking Structure:** We introduce an innovative 2D blocking structure whose design enhances sub-block independence and enables faster, more convenient sub-block mapping to warps.
- **Intra-Block Data Aggregation:** We propose a data aggregating method using a virtual pointer structure to unite the format of different data within the sub-block.
- **Computation Adaptation Strategy:** To address varying sparsity levels among sub-blocks, we propose a computation adaptation strategy that optimizes hardware utilization and improves parallel efficiency.
- **Thread-block Load Balancing Optimization:** We develop a parallel load balancing algorithm to mitigate workload disparities among thread blocks.
- **CB-SpMV Framework:** Combining them all, we propose CB-SpMV, a cache-friendly block-based framework for SpMV on GPUs, leveraging data aggregation and balancing to boost performance. Experiments on two GPUs demonstrate that CB-SpMV achieves higher cache hit rates and outperforms SOTA methods.

The remainder of this paper is organized as follows: Section 2 describes the background of the sparse format and SpMV, with a comprehensive discussion of its data locality, hardware utilization, and load balance challenges. The proposed CB-SpMV and relative optimization method are presented in Section 3. Section 4 presents the evaluation results of the CB-SpMV on RTX 4090 and A100. We discuss the related work in Section 5 and conclude this paper in Section 6.

2 Background, Motivation and Challenge

2.1 Sparse Matrix Storage Format

Sparse matrices, characterized by a few non-zero elements per row, require specialized storage formats to improve access efficiency. The COO format is widely used for its simplicity and compatibility with data storage, stores non-zero elements as `(row_idx, col_idx, coo_val)` triplets, enabling rapid construction and easy conversion to other formats. The CSR format, among the most widely adopted,



(a) General SpMV (b) Blocked SpMV

Figure 2: An example of an SpMV that multiplies a 6-by-6 sparse matrix A by a vector x to get a vector y.

organizes data into `row_ptr`, `col_idx`, and `csr_val` arrays, offering compact storage and efficient row-wise operations. With the rise of block accelerators such as Tensor Cores, block-based formats like the Block Compressed Sparse Row (BSR) format have gained traction. BSR partitions matrices into fixed-size blocks, using block-level arrays (`blk_row_ptr`, `blk_col_idx`) and storing block data in `bsr_val`, including both non-zero and zero elements. This structure is particularly effective for dense matrices. Advanced methods such as TileSpMV[39] further optimize block-based SpMV by employing mixed storage formats at the block level to reduce zero storage overhead, as illustrated in Fig.1.

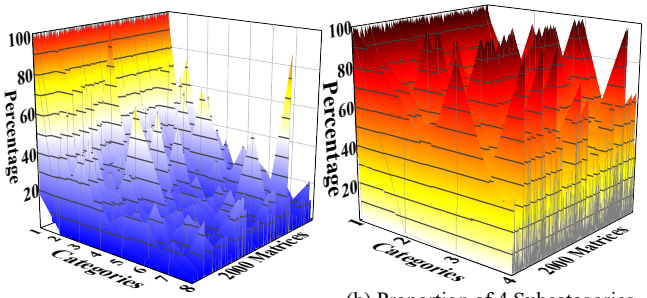
2.2 SpMV and Motivation

SpMV calculates the product of a sparse matrix A and a dense vector x, iterating over non-zero elements to update the output vector y. As a core operation in sparse linear algebra, SpMV is widely used in scientific computing and engineering applications as a core operation in sparse linear algebra. Fig.2 and Alg.1 illustrate its computation and logic.

Algorithm 1 A pseudocode of parallel CSR SpMV.

```
1:  s_x ← x                                   ▷ s_x is in shared memory
2:  for i = 0 to m in parallel do
3:      sum ← 0
4:      for j = row_ptr[i] to row_ptr[i+1] do
5:          //m + 1 elements are spanned from row_ptr to col_idx
6:          val_x ← s_x[col_idx[j]]
7:          //nnz elements are spanned from col_idx to csr_val
8:          sum ← sum + val_x × csr_val[j]
9:      end for
10:     y[i] ← sum
11: end for
```

However, performing SpMV with the CSR format introduces significant data locality issues. As shown in Fig.1, accessing the element at (0,4) and other elements begins with querying `row_ptr` to determine the starting positions of non-zero elements, which exhibits high locality due to adjacent data. However, subsequent accesses to `col_idx` and `csr_val` involve significant memory jumps, leading to a marked reduction in cache efficiency. These jumps recur within rows, further degrading data locality and overall cache performance. Unlike CPUs, which rely on hardware prefetchers to mitigate memory latency[23], GPUs depend on high parallelism to hide latency, offering limited prefetching capabilities. For example, on an RTX 4090 GPU with 128 SMs, each with 128 KB of

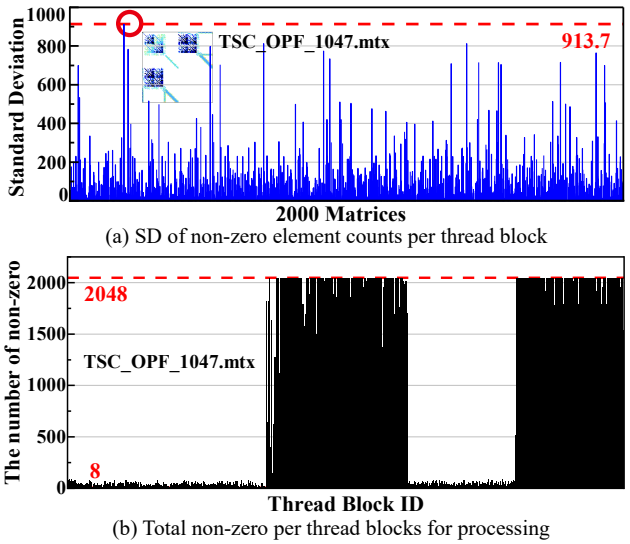


(a) Proportion of Block Categories (b) Proportion of 4 Subcategories in 1-32 Block Categories

Figure 3: Distribution of non-zero elements in 2000 matrices (with over 10,000 non-zero elements) under 16×16 block partition: (a) Proportion of sub-blocks across eight categories (1-32, 33-64, ..., 225-256); (b) Further subdivision of the 1-32 category into four subcategories (1-8, 9-16, 17-24, 25-32).

L1 cache, the theoretical L1 cache per thread is only 64B when scheduling multiple TBs per SM. Similarly, the shared 72MB L2 cache is insufficient to accommodate large memory spans, making it challenging to leverage GPU caches effectively. Other formats like COO and BSR also suffer from locality challenges. COO incurs higher jumps due to the direct traversal of non-zero elements. At the same time, BSR improves locality by processing data in blocks but at the cost of storing zero elements, reducing efficiency for sparse matrices. TileSpMV addresses some of BSR's limitations by compressing sub-blocks but fails to fully exploit the inherent locality of BSR-dense sub-blocks, reintroducing issues similar to CSR, as shown in Fig.1. These limitations motivate the design of a novel sparse storage structure that preserves BSR's locality advantages while eliminating zero-element storage, enabling more efficient SpMV computation.

Besides the data locality issue, we also observe that hardware under-utilization and load imbalance limit the efficiency of block-based SpMV. Regarding the hardware under-utilization, the BSR format faces a critical limitation due to the significant variation in sub-block sparsity after partitioning. For example, Nvidia GPUs comprise 32 threads in a wrap. If the sub-blocks contain only a few non-zero elements, most threads will be idle, wasting computational resources. Fig.3(a) illustrates the distribution of non-zero elements in sub-blocks for 2000 SuiteSparse matrices (each with over 10,000 non-zero elements) using a 16×16 block size. Sub-blocks were categorized into eight ranges (1–32, 33–64, ..., 225–256), with over 90% of sub-blocks in most matrices falling within the 1–32 range, averaging 81.89% across all matrices. In Fig.3(b), the 1–32 range is further divided into four subcategories, revealing that sub-blocks with 1–8 non-zero elements constitute 59.36% of the total, and those with 9–16 elements account for 20.35%. This indicates that 59.36% of sub-blocks have a warp-level thread utilization below 75%, while 20.35% falls below 50%, leading to significant efficiency losses. Although TileSpMV[39] partially addresses this by consolidating sparse sub-blocks and processing them with CSR5[33] while handling the rest with TileSpMV, this approach incurs additional overhead from merging and consolidation. Hence, more efficient solutions to this issue are still needed.



(a) SD of non-zero element counts per thread block

(b) Total non-zero per thread blocks for processing

Figure 4: Standard deviation and distribution of non-zero elements per thread block: (a) Standard deviation across 2000 matrices; (b) Distribution for a specific matrix (`TSC_OPF_1047.mtx`).

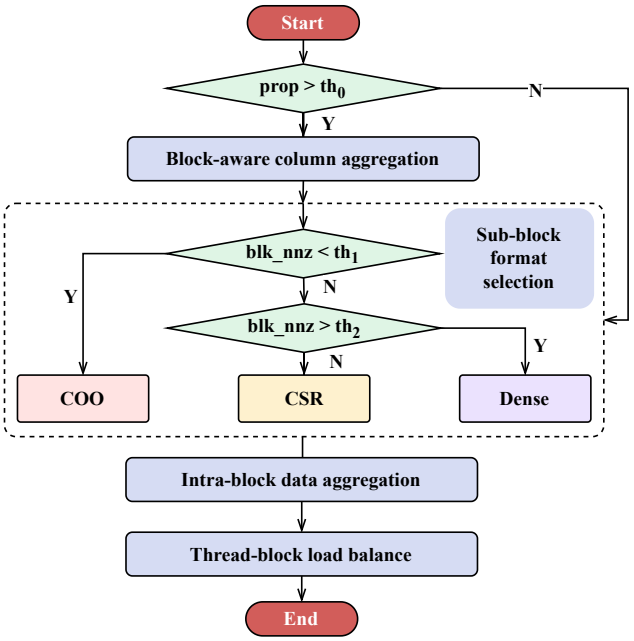


Figure 5: The overview flow chart of CB-SpMV

Regarding load imbalance, thread blocks(TBs) serve as the basic scheduling units for SMs in GPU computation, with each block comprising multiple warps. Due to varying sparsity among sub-blocks, the number of non-zero elements assigned to each thread block often differs significantly. Fig.4 highlights this imbalance across 2000 matrices (over 10,000 non-zero elements each) with a 16×16 block size. The standard deviation of non-zero elements per thread block varies widely, peaking at 913.7 for `TSC_OPF_1047.mtx`, indicating severe load imbalance. Since GPU scheduling assigns thread blocks to SMs in a round-robin manner, the execution time for an SM is dictated by its slowest thread block, typically the one with the heaviest load. This imbalance in workload distribution among thread blocks can significantly degrade performance without proper mitigation.

2.3 Software and Hardware Constraints

We further observe that the programming model and GPU itself raise significant constraints when designing mixed precision and better data locality block-based SpMV. The constraints can be categorized into three aspects. First, GPU functions are designed to process homogeneous data that contracts with sparse structures. Taking `memcpy()` and `malloc()` as an example, only homogeneous data types are supported during data transferring. However, sub-block data often include mixed types and precisions—e.g., `int` for coordinates and `float` or `double` for numerical values—making it impossible to transfer these data types simultaneously, which incurs high data management overhead. Second, the GPU's memory alignment mechanism also introduces organizational challenges. Different data types, such as `int`(4B) and `double`(8B), may not align contiguously in memory, resulting in inefficient storage and access patterns. Due to these two constraints, although the state-of-the-art block-based structures partition large matrices into sub-blocks, they cannot fully capture the data locality within sub-blocks. Moreover, the parallel executed model poses significant difficulties in mapping imbalanced computing tasks to GPU hardware. For example, Nvidia GPUs process threads within warps in a SIMD manner and group multiple warps to form a thread block for resource scheduling. Since each thread runs the same code and is scheduled simultaneously by a single thread block, it is difficult to achieve highly flexible task assignments for high hardware utilization and load balance.

Thus, by considering only the block-based SpMV itself without breaking these software and hardware constraints, new bottlenecks will emerge in this kind of SpMV.

3 CB-SpMV

To overcome the challenges of block-based SpMV, we propose a cache-friendly block-based SpMV approach, CB-SpMV, specifically designed for GPUs. This approach incorporates a novel mixed-precision method and an adaptable 2D sparse structure tailored to matrices with varying sparsity. In our design, CB-SpMV efficiently transforms the input sparse matrix into the proposed high-locality 2D sparse structure during the data loading phase using innovative data aggregation and balancing algorithms. To fully leverage the benefits of the CB-SpMV format, we also redesign the computation logic of the SpMV kernel. Notably, the threads within each block are assigned in the original manner to ensure the high level of parallelism inherent to GPUs is preserved.

As shown in Fig.5, the overview flow chart of CB-SpMV incorporates three main components: an intra-block data aggregation to enhance data locality, a block-aware column aggregation, and an efficient inter-block load balancing algorithm to improve warp and block-level parallelism as well as hardware utilization. First, input data is loaded as a block-based COO format, similar to HiCOO [29]. After checking the characteristics of the input matrix, the

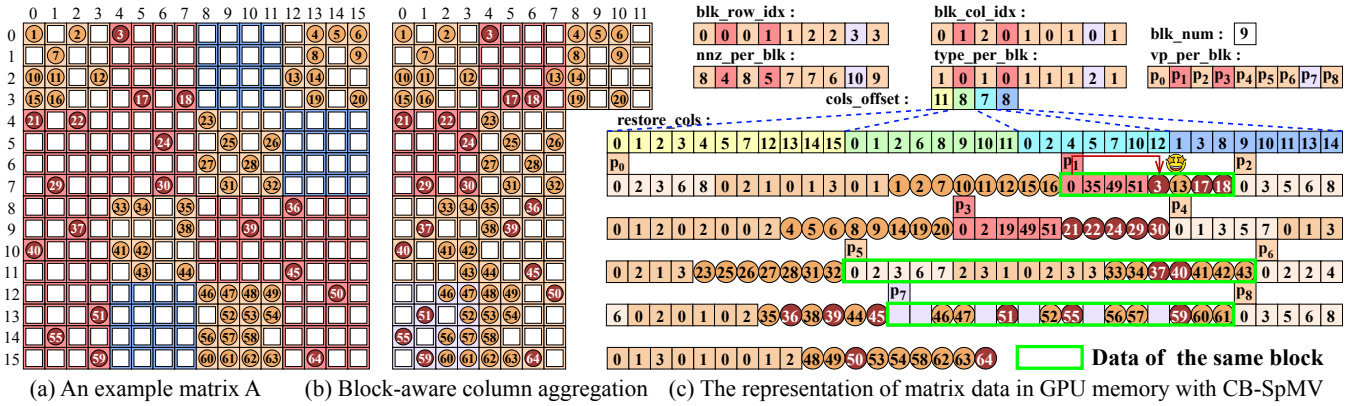


Figure 6: The storage format and memory organization of CB-SpMV. (a) illustrates a 16-by-16 example matrix, where different colors indicate that sub-blocks adopt distinct storage formats. (b) depicts the matrix after column aggregation. (c) shows the 2D metadata of CB-SpMV. In comparison, CB-SpMV aggregates different types of data within the sub-block for better data locality.

input data passes through the column aggregation to increase non-zero value density. Then, the compressed matrix is transformed into a 2D sparse structure, and the sub-block format is selected for efficient processing. After that, the various types of data within sub-blocks are aggregated to capture the data locality. Finally, the load balancing is achieved via an inter-block exchange algorithm, ensuring an even distribution of non-zero elements among thread blocks, ultimately improving computational efficiency. We set different thresholds to convert the sparse matrix for optimal data locality and parallelism, with the detailed threshold setting method introduced alongside each component.

3.1 2D Sparse Structure

In this work, CB-SpMV divides the input sparse matrix into uniform 16×16 sub-blocks to balance data locality, parallelism, and thread utilization, which serve as the fundamental computational units, with each sub-block mapped to a warp for intra-block computation. This block size ensures efficient warp-level execution on NVIDIA GPUs, avoiding under-utilization from sparse sub-blocks or reduced parallelism from overly large blocks. The method employs a two-level metadata structure: a high-level structure for locating sub-blocks and a low-level structure for managing data within each sub-block. Fig.6 illustrates this 2D sparse structure.

The high-level metadata adopts the COO format for efficient localization of sub-block positions and warp binding. It consists of five arrays: `blk_row_idx` and `blk_col_idx` for row and column indices of non-zero sub-blocks, `nnz_per_blk` for the number of non-zero elements in each sub-block, `vp_per_blk` for the starting GPU memory addresses of sub-block data, and `type_per_blk` to specify the storage format of each sub-block. The low-level structure defaults to the COO format for rapid conversion from the input data to the CB-SpMV structure. To treat the different types of data within the sub-blocks as a united structure, these sub-blocks are then aggregated into a single structure array (`blk_data`) on the CPU and packed contiguously into a one-dimensional meta-data (`mtx_data`) on the GPU via intra-block data aggregation.

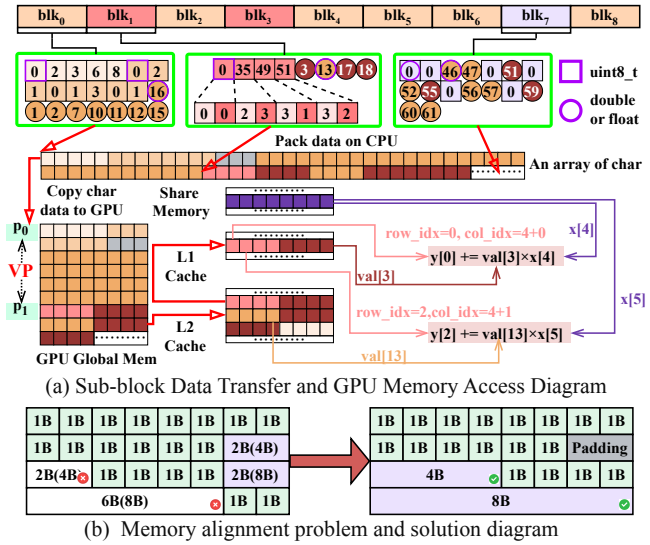


Figure 7: Sub-block Data Transfer and GPU Memory Access: (a) Packaging of sub-block data on the CPU, transfer to GPU global memory, and access via L1, L2 caches, and shared memory; (b) Memory alignment issue and its resolution.

3.2 Intra-Block Data Aggregation

To improve data locality and cache efficiency, CB-SpMV employs an intra-block data aggregating strategy, as shown in Fig.7 to pack and compress data within sub-blocks tightly. This approach leverages the independence of non-zero element coordinates within each sub-block after matrix partitioning, treating all sub-block data as a single unit for fast and unified data packaging and transfer.

First, **coordinate compression** is utilized to reduce memory consumption by encoding the row and column indices (`row_idx` and `col_idx`) of 16×16 sub-blocks into a compact format. Each index requires only 4 bits, and the two indices are combined into a single `uint_8` type using bitwise operations, as depicted in Fig.7. This approach is extended to other sparse formats, except for dense formats where index storage is unnecessary. Second, a **virtual pointer** (VP)

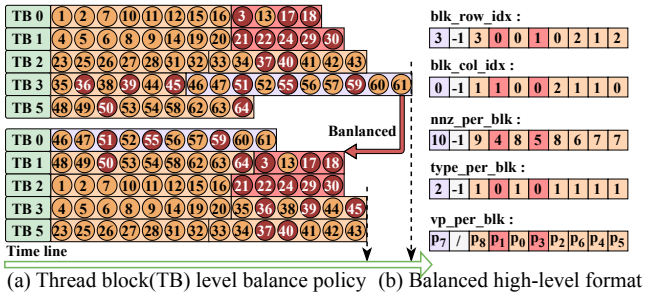
mechanism is introduced to enable efficient data transfer of mixed sub-block data. The VP points to a contiguous memory region on the GPUs, allocated as `char` data. Then, sub-block data of different types and precisions is aggregated into a single uniform-format sequence on the CPU, which is then transferred to the GPU in a single operation. Access on the GPU is performed via pointer offsets based on the VP and the number of non-zero elements in each sub-block, ensuring both efficiency and simplicity. Finally, since misalignment may lead to incorrect computations when the GPU accesses multiple bytes simultaneously. As shown in Fig.7(b), a **padding strategy** is applied to address potential memory misalignment caused by GPU's data alignment constraints.

3.3 Computational Adaptation

Although CB-SpMV can have higher data locality after intra-block data aggregation, we also face the under-utilize challenge to implement SpMV: high sparsity sub-blocks cannot fully occupy the threads within a warp, while dense sub-blocks stored in the COO format incur memory overhead and require atomic operations, reducing parallel efficiency. To address these challenges, CB-SpMV introduces a computational adaptation strategy tailored to sub-block sparsity.

3.3.1 Sparse Sub-blocks. For sparse sub-blocks, a block-aware **column aggregation** strategy is proposed. As shown in Fig.6(b), columns with all-zero elements in a sub-block are removed, and the remaining columns are shifted forward. Two arrays, `restore_cols` and `cols_offset`, map aggregated columns back to their original indices and record the number of non-zero columns, respectively. This ensures that each non-zero sub-block contains at least 16 non-zero elements, mitigating sparsity. However, block-aware column aggregation introduces stridden memory access to the x vector, as aggregated column indices may no longer be contiguous. This prevents pre-loading x into shared memory, slightly increasing global memory access. To balance trade-offs, we set a threshold, th_0, to determine if column aggregation is applied based on the sparsity of sub-blocks. As we mentioned, over 90% of sub-blocks in most matrices have lower than 32 non-zero values, which we call these sub-blocks are super-sparse sub-blocks. Considering the rare cases where the proportion of super-sparse sub-blocks to the total number of blocks is low after matrix partitioning, storing the x data of these sub-blocks in shared memory mitigates the performance gap. Consequently, column aggregation is not applied to these matrices. Thus, in our work, th_0 is set to a relatively small value of 0.15.

3.3.2 Dense Sub-blocks. For excessively dense sub-blocks, CB-SpMV employs a **format selection** strategy, storing sub-blocks in one of three formats: COO for low-density blocks(the number of non-zero elements in a block is less than th_1), Dense for high-density blocks(the number of non-zero elements in a block is more than th_2), and CSR for intermediate sparsity. Refer to previous work TileSpMV[39], th_1 and th_2 are set to 32 and 128, respectively. This approach minimizes branching overhead while optimizing intra-block computation using the `shfl` function to enhance warp-level performance, as Alg.3 and Alg.4 illustrate.



(a) Thread block(TB) level balance policy (b) Balanced high-level format

Figure 8: Block level load balancing policy and high-level structure after balancing

3.4 Inter-thread-block Load Balance

To address load imbalance among thread blocks, we proposed an inter-block exchange strategy to adjust the sub-block allocation for each thread block while maintaining the total number of sub-blocks of each thread block to be approximately equal. This involves reorganizing the metadata arrays `blk_row_idx`, `blk_col_idx`, `nnz_per_blk`, `type_per_blk`, and `vp_per_blk` by priority queue(pq), as shown in Fig.8. The figure shows our load balancing example for the matrix in Fig.6, with two warps per thread block(TB), and the simplified pseudocode is shown in Alg.2.

Algorithm 2 A pseudocode of Thread Block Load Balance.

```
1:  //The blk_idx_array contains 3 items: ori,end,nnz.
2:  parallel sort(blk_idx_array, cmp_nnz);
3:  //The pq contains 3 items: loads(min-heap), tb_id, warps.
4:  for i = 0 to blk_num − 1 in parallel do
5:      pqtop ← pq.top(), then pq.pop()
6:      // Mapping sub-block to target thread block
7:      blk_idx_array[i].end ← pqtop.tb_id × 8 + pqtop.warps
8:      pqtop.loads ← pqtop.loads + blk_index_array[i].nnz
9:      pqtop.warps ← pqtop.warps + 1
10:     if pqtop.warps < 8 then
11:         pq.push(pqtop)
12:     end if
13: end for
14: parallel sort(blk_idx_array, cmp_end);
15: for i = 0 to blk_num − 1 in parallel do
16:     vp_per_blk[i] ← vp_per_blk_old[blk_idx_array[i].ori]
17:     //The high-level array (eg.blk_row_idx,...) op similarly
18: end for
```

This approach leverages two key innovations:

(1) The high-level independent COO structure enables flexible rearrangement of sub-block indices while ensuring efficient localization within the global matrix.
(2) All data in each sub-block is stored contiguously, allowing the virtual pointer (VP) to be directly retrieved for efficient computation access.

Compared to previous methods that use CSR as the high-level format and store sub-block data non-contiguously, this strategy

achieves superior load balancing by enabling dynamic and efficient sub-block remapping without altering the total number processed per thread block.

3.5 Kernel Implementation

To accommodate the proposed optimization measures, the SpMV kernel's computation logic and algorithm have been redesigned to fully exploit the CB-SpMV format's advantages.

Algorithm 3 A pseudocode of CB-SpMV with COO.

```
for each block matrix A_k in the matrix A in parallel do
    blk_nnz ← the nnz of A_k
    VP ← the virtual pointer of A_k
    // Unpacking data
    blk_row_idx ← the row block index of A_k
    padding ← (blk_nnz × size(Idx)) mod size(Val)
    padding ← padding ? size(Val) − padding : 0
    coo_idx ← VP
    coo_val ← VP + blk_nnz × size(Idx) + padding
    for i = 0 to blk_nnz in parallel do
        row_idx ← blk_coo_idx[i] & 15
        col_idx ← blk_coo_idx[i] ≫ 4
        y_idx ← blk_row_idx × BLK_SIZE + row_idx
        if column aggregation is not used then
            // Corresponding x is preloaded into sm.
            atomicADD(y_{y_idx}, coo_val[i] × s_x_{col_idx})
        else
            //Obtain the global pointer of x.
            offset ← cols_offset[blk_row_idx] + col_idx
            col_idx_ori ← restore_cols[offset]
            atomicADD(y_{y_idx}, coo_val[i] × d_x_{col_idx_ori})
        end if
    end for
end for
```

For sub-blocks stored in the COO format, a warp of 32 threads is allocated to process all non-zero elements in one sub-block. Even if the number of non-zero elements is less than 32, the column aggregation strategy ensures that each sub-block contains at least 16 elements, improving hardware utilization to at least 50%. This marks a significant improvement over traditional approaches, which often leave most threads idle in sparse sub-blocks. The packed data for each sub-block is parsed using the metadata arrays `nnz` and VP to extract `row_idx`, `col_idx`, and `val`. The computation results are then partially accumulated into the corresponding `y` vector using the `atomicAdd` operation, ensuring correctness across concurrent threads.

For sub-blocks stored in CSR or Dense formats, the kernel logic is optimized to leverage GPU-specific instructions. Since 32 threads collaboratively compute 16 `y` vector elements, the unpacking process is followed by a warp-level reduction using the `shfl` instruction. This operation minimizes access to shared or global memory, enhancing computational efficiency. The dense format further benefits from optimized memory coalescing due to its regular structure, reducing cache misses. Alg.3 and Alg.4 provide pseudocode for the redesigned COO and Dense kernels. These kernels are categorized based on whether column aggregation is applied. For sub-blocks with column aggregation, the `x` vector is directly loaded from global

Algorithm 4 A pseudocode of CB-SpMV with Dense.

```
for each block matrix A_k in the matrix A in parallel do
    blk_nnz ← nnz of A_k
    VP ← virtual pointer of A_k
    blk_row_idx ← row block index of A_k
    dense_val ← VP
    for tid = 0 to warpsize − 1 in parallel do
        sum ← 0
        row_ft ← (tid < BLK_SIZE)?tid : tid − BLK_SIZE
        start ← (tid < BLK_SIZE)?0 : BLK_SIZE/2
        end ← (tid < BLK_SIZE)?BLK_SIZE/2 : BLK_SIZE
        for col = start to end do
            val ← dense_val[row_ft × BLK_SIZE + col]
            if column aggregation is not used then
                sum+ = val × s_x_col
            else
                offset ← cols_offset[blk_row_idx] + col
                col_idx_ori ← restore_cols[offset]
                sum+ = val × d_x_{col_idx_ori}
            end if
        end for
        if tid < BLK_SIZE then
            sum ← __shfl_xor_sync(0xffffffff, sum, BLK_SIZE/2)
                                ▷ Aggregate partial results within warp
        end if
    end for
    for tid = 0 to BLK_SIZE in parallel do
        y_idx ← blk_row_idx × BLK_SIZE + tid
        atomicAdd(y_{y_idx}, sum)
    end for
end for
```

memory into registers, requiring a remapping from aggregated to original column indices. Conversely, for sub-blocks without column aggregation, the corresponding portion of the `x` vector is pre-loaded into shared memory to minimize memory latency during computation. This distinction ensures that the kernel adapts dynamically to different sub-block sparsity levels, maximizing performance while maintaining simplicity.

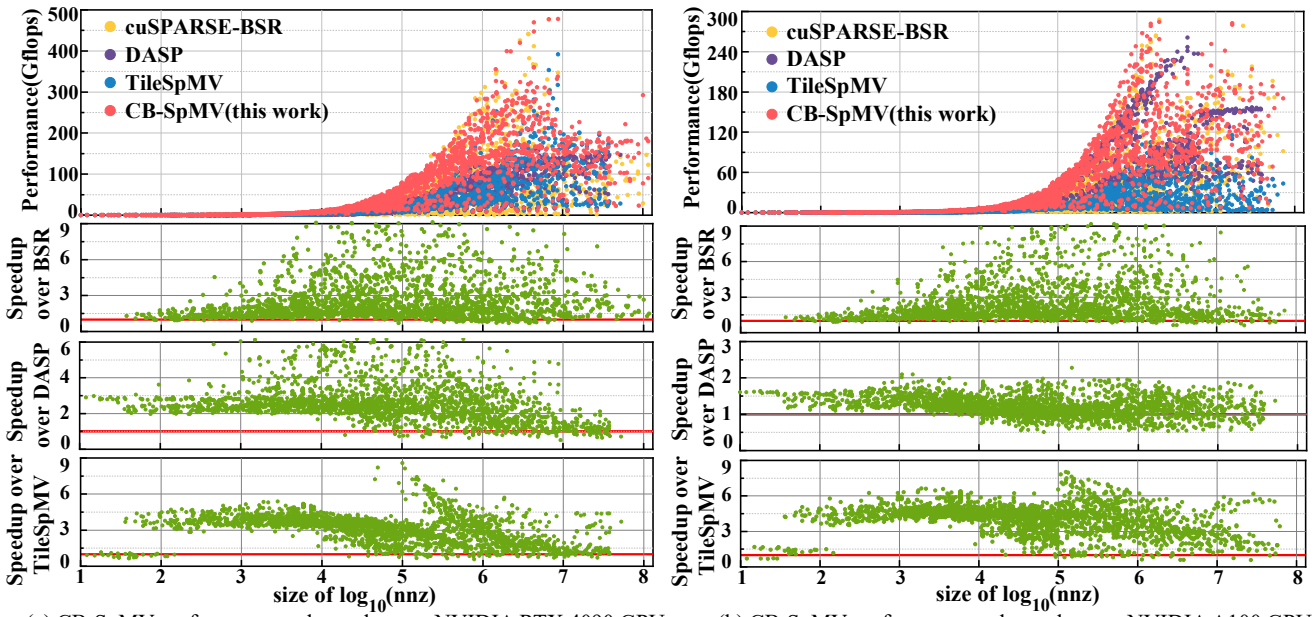
4 Evaluation

4.1 Experimental Setup

The experimental platform consists of two types of NVIDIA GPUs: the NVIDIA A100 (Ampere architecture) and RTX 4090 (Ada Lovelace architecture). Both GPUs are configured with driver version 550.135 and CUDA version 12.4. On this platform, the proposed method is comprehensively compared with the following state-of-the-art approaches: The `cusparse_bsrmv()`[14] kernels from cuSPARSE v12.4 for SpMV using BSR formats; the state-of-the-art block-based SpMV method TileSpMV[39]; and the latest based on CSR format and Tensor Core-accelerated SpMV method DASP[34]. The table below lists the specifications of the tested GPUs and the algorithms. The evaluation dataset comprises 2843 matrices from the SuiteSparse Matrix Collection[15].

4.2 SOTA Technology Comparison

In this subsection, we evaluate the performance of CB-SpMV (with `val` in FP64 format) against the SOTA block-based SpMV algorithms



(a) CB-SpMV performance and speedups on NVIDIA RTX 4090 GPU

(b) CB-SpMV performance and speedups on NVIDIA A100 GPU

Figure 9: Performance comparison between CB-SpMV and the SOTA SpMV algorithm on RTX4090 and A100 GPUs

Table 1: The two GPUs and four algorithms evaluated.

Two NVIDIA GPUs	Four Algorithms
(1) NVIDIA A100 (Ampere), 6912 CUDA cores @ 1410 MHz, 40 GB, B/W 1.56 TB/s	(1) cuSPARSE [14]
(2) NVIDIA RTX4090 (Ada Lovelace), 16384 CUDA cores @ 2520 MHz, 24GB, B/W 1.01 TB/s	(2) TileSpMV [39]
	(3) DASP [34]
	(4) CB-SpMV (this work)

(e.g., cuSPARSE-BSR[14], TileSpMV[39]) and other recent SpMV methods (DASP[34]), focusing on Gflop/s (Gflops) as the primary performance metric. We run each kernel 1000 times to obtain average Gflops values. For cuSPARSE-BSR, we use the best-performing block size from 2×2, 4×4, 8×8, and 16×16 while for TileSpMV and our work, we set the block size as 16x16. Additionally, we analyze the impact of the intra-subblock data aggregation strategy on L1 and L2 cache hit rates.

4.2.1 Performance Comparison. As shown in Fig.9, CB-SpMV significantly outperforms BSR on both GPUs. On the RTX 4090, CB-SpMV achieves an average speedup of 2.95× and a maximum speedup of 37.54× (on `TSOPF_FS_b39_c30`). On the A100, the average and maximum speedups are 2.99× and 54.27× (on `boyd1`). These improvements stem from CB-SpMV's use of sparse formats for low-level sub-blocks, avoiding the dense storage overhead of BSR and reducing underutilization in sparse sub-blocks. CB-SpMV also consistently outperforms TileSpMV. For smaller matrices (*nnz* less than 10^5), the primary gains arise from enhanced data locality via intra-subblock data aggregation. For larger matrices, strategies like column aggregation, format selection, and load balancing further boost performance. On the RTX 4090 and A100, CB-SpMV achieves average speedups of 3.06× and 3.95×, with maximum speedups of 8.56× (on `piston`) and 10.34× (on `rgg_n_2_21_s0`).

The comparison with DASP, which leverages Tensor Core designs, is particularly noteworthy. On the RTX 4090 and A100 GPUs, CB-SpMV achieves average speedup factors of 2.76× and 1.21×, respectively. The reduced performance advantage of the A100 is primarily due to its design as a high-performance computing GPU, offering significantly higher FP64 Tensor Core computational capabilities compared to the RTX 4090, which is optimized for graphics processing. Nevertheless, despite DASP's reliance on the powerful Tensor Core hardware, CB-SpMV still achieves approximately 20% speedup on the A100. This is likely because DASP's data layout strategy fails to fully exploit the potential of Tensor Cores, as discussed in Section 1. Addressing this issue will be a key focus of our future work.

4.2.2 Cache Hit Rate Comparison. To vividly illustrate the improvements in data locality achieved by sub-block data aggregation and the associated strategies, we selected 15 representative matrices, as shown in the table 2. Using the Nsight Compute tool on the RTX 4090, we analyzed parameters such as `l1tex__t_sector_hit_rate` for L1 Cache hit rate and `lts__t_request_hit_rate` for L2 Cache hit rate. These metrics were used to evaluate the Gflops and average L1 and L2 cache hit rates for the three methods mentioned earlier when applied to these 15 matrices.

As shown in Fig.10, CB-SpMV achieves L1 cache hit rates 14.4× and 1.93× higher than BSR and TileSpMV, respectively, and L2 hit rates 5.89× and 1.19× higher. Additionally, matrices with more dense sub-blocks, such as `mouse_gene` and `sme3Da`, exhibit higher

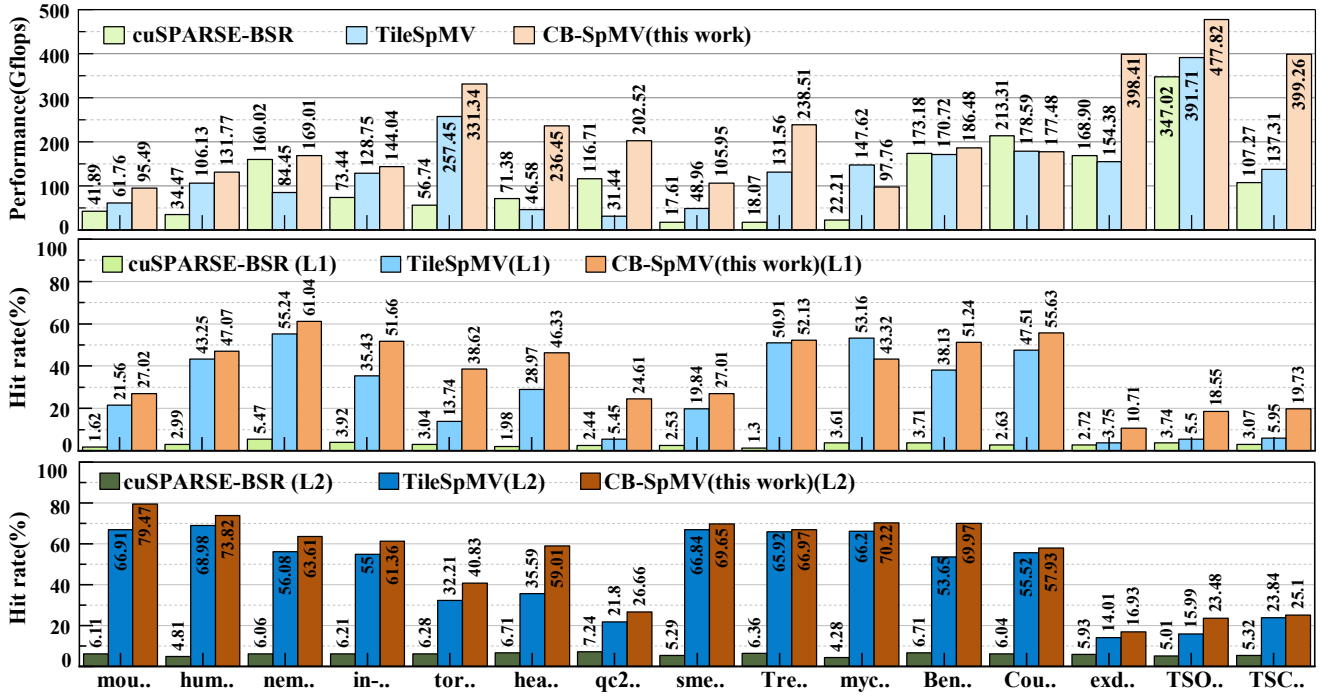
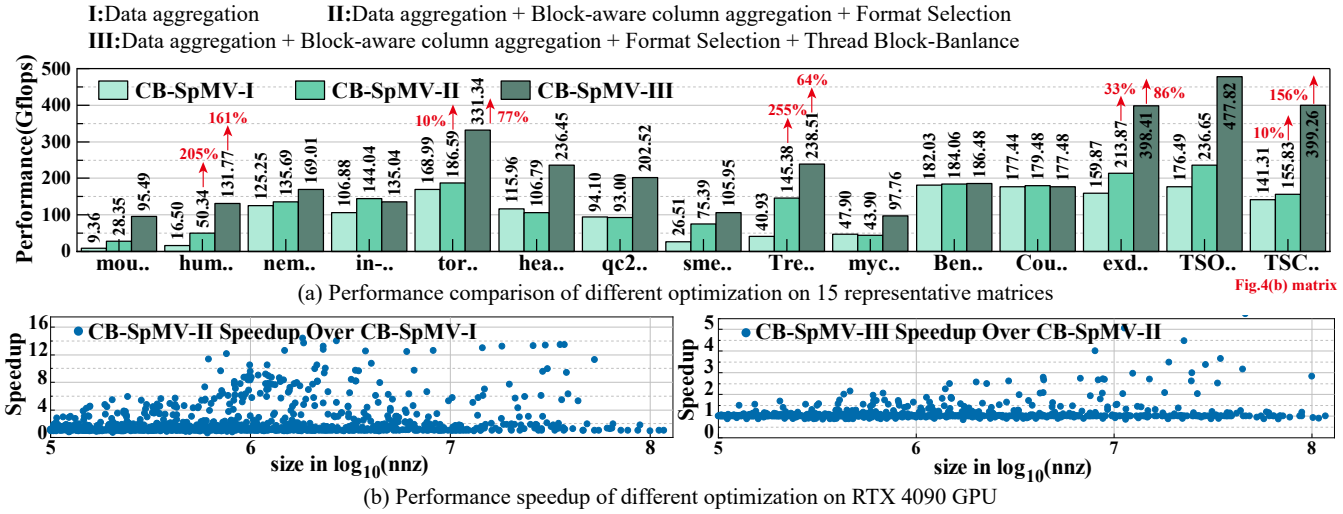


Figure 10: Performance comparison of 15 typical sparse matrices on RTX 4090 GPU and L1 & L2 Cache hit ratio



(a) Performance comparison of different optimization on 15 representative matrices

(b) Performance speedup of different optimization on RTX 4090 GPU

Figure 11: Performance comparison of 15 typical sparse matrices using different optimization measures using CB-SpMV on RTX 4090 GPUs.

L2 cache hit rates. This is because dense sub-blocks, after aggregation, often require larger storage spaces, leading to more frequent evictions and reloads in the L1 cache. In contrast, matrices with more sparse sub-blocks, such as `nemeth07` and `CoupCons3D`, tend to have higher L1 cache hit rates. This is because sparse sub-blocks contain fewer data elements, allowing the L1 cache to accommodate more of their content. These patterns underscore the adaptability of CB-SpMV in improving data locality across matrices with varying structural characteristics.

4.3 Ablation Performance Analysis

To address the challenges outlined in Section 2, we proposed targeted optimization strategies in Section 3. This section evaluates their individual and combined impacts on CB-SpMV's performance, focusing on 15 representative matrices with $nnz > 100,000$, as preprocessing selectively applies these optimizations based on matrix characteristics.

Table 2: Information of the 15 Representative Matrices.

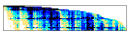
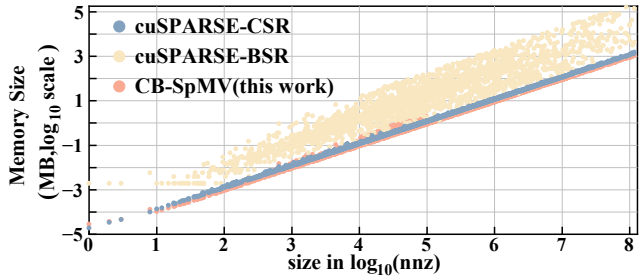
Name	Plot	Size ($m \times n$)	nnz
mouse_gene		45101×45101	28967291
human_gene1		22283×22283	24669643
nemeth07		9506×9506	394812
in-2004		1382908(m=n)	16917053
torso1		116158×116158	8516500
heart2		2339×2339	682797
qc2534		2534×2534	463360
sme3Da		12504×12504	874887
Trec14		3159×15905	2872265
mycielskian15		24575×24575	11111110
BenElechi1		245874×245874	13150496
CoupCons3D		416800×416800	22322336
exdata_1		6001×6001	2269501
TSOPF_RS_b6..		35696×35696	8781949
TSC_OPF_1047		8140×8140	2016902

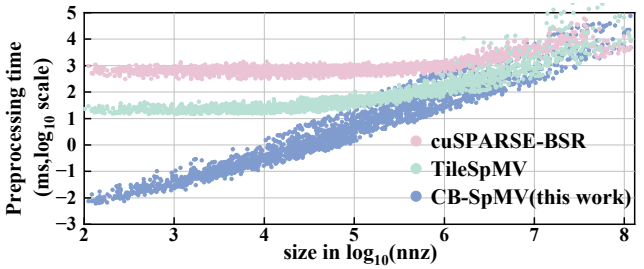
Fig.11 highlights the contributions of each optimization strategy. Compared to using only intra-block data aggregation (CB-SpMV-I), the column aggregation and format selection strategy achieves an average speedup of 2.22×, and the thread block-level load balancing strategy provides an additional 1.09× improvement compared to using sub-block data aggregation, column aggregation and format selection (CB-SpMV-II). When combined, these strategies deliver an average performance gain of 2.37×. The column aggregation and format selection strategy addresses issues in matrices with extreme sub-block sparsity or density. For example, in `human_gene1`, this optimization mitigates reduced parallelism caused by dense sub-blocks, achieving a 205% improvement. Conversely, for `exdata_1`, it enhances warp utilization in sparse sub-blocks, delivering a 33% gain. The thread block-level load balancing strategy effectively distributes the workload among thread blocks, resolving imbalances in matrices with mixed dense and sparse regions, such as `torso1` and `exdata_1`, achieving 77% and 86% improvements, respectively.

4.4 Overhead Analysis

In this section, we evaluate the advantages and disadvantages of the proposed method in terms of storage overhead and preprocessing time and compare it with other sparse matrix storage formats.



(a) Comparison of memroy size of SpMV methods



(b) Comparison of preprocessing costs of SpMV methods

Figure 12: Comparison of memory usage and preprocessing time across different SpMV storage formats.

4.4.1 storage overhead. To evaluate storage overhead, we modeled the space usage of CSR, BSR, and CB-SpMV under simplified assumptions: position information is stored as `int32`, and numerical values are stored as `FP64`. For a matrix of size $m \times n$ with nnz non-zero elements, the CSR format requires $(m+1)\times4+nnz\times4+nnz\times8$B. For the BSR format, assuming a block size of 16×16, with $nnzb$ non-zero sub-blocks and blk_m row blocks, the required storage is $256 \times 8 \times nnzb + (blk_m + 1) \times 4 + nnzb \times 4$B. For CB-SpMV, where each sub-block uses COO storage, the storage overhead is $nnzb\times(4+4+4+1+8)+nnz\times(1+8)$B. Column aggregation costs are excluded, as the reduction in block count offsets high-level storage costs. As shown in Fig.12(a), CB-SpMV achieves storage efficiency comparable to CSR, thanks to its compressed sub-block storage and smaller indices. In contrast, BSR incurs significantly higher overhead due to storing numerous `FP64` zero elements within sub-blocks. This demonstrates that CB-SpMV effectively balances storage efficiency and computational adaptability.

4.4.2 preprocessing time overhead. The preprocessing time required to convert COO matrices into the formats used by various algorithms is shown in Fig.12(b). CB-SpMV consistently outperforms TileSpMV and cuSPARSE-BSR for matrices with nnz less than 10^6. For matrices with nnz greater than one million, CB-SpMV remains comparable to those of TileSpMV and BSR. Despite the modest increase in preprocessing time for larger matrices, CB-SpMV remains competitive, particularly for iterative solvers that require repeated SpMV operations. The preprocessing overhead is a small trade-off for the significant performance improvements achieved during computation.

5 Related Work

There has been extensive prior research on SpMV; here, we focus only on reviewing works relevant to this study.

Block-based Methods: In recent years, with the rise of block and tensor core, block-based approaches about GPUs have garnered significant attention in these studies[3, 5, 7, 8, 17, 28, 35, 38–40, 42, 45, 48]. Researchers like Labini and Bernaschi [28] leveraged reordering and blocking techniques and dense accelerators such as NVIDIA Tensor Cores to achieve efficient sparse matrix multiplications. Buluç[8–11] et al. introduced the CSB format to exploit block layouts and cache locality, which Martone further improved using recursive methods. Yan[47] et al. developed the BCCOO format to store dense 2D blocks, and Niu[39] et al. extended this idea with TileSpMV by supporting seven internal sub-block structures. Compared to these studies, our approach aggregates and packs data within each sub-block, treating each as an independent unit with tightly adjacent data. This method achieves better memory locality among block-based algorithms, significantly improving the cache hit rate for blocked matrix computations. Beyond advancing the theoretical understanding of block-based sparse matrix SpMV, this approach also opens new avenues for applying blocked matrices to other computations, such as SpMM and SpGEMM.

Formats for SpMV: Widely adopted strategies for accelerating SpMV computations focus on designing novel storage formats and optimizing algorithms[12, 16, 19, 21, 25, 33, 44, 50]. Among them, numerous ELL and CSR-based formats have been proposed to enhance performance. Notable examples include the HYB format combining ELL and CSR/COO[4] and the clSpMV framework integrating multiple formats. Variants of ELL[1, 3, 27, 31, 33, 46] and CSR[2, 26, 33, 36, 43] formats have demonstrated significant advantages, especially on GPUs. Yesil et al.[49] further improved data locality by separating matrices into dense and sparse regions with customized representations. Despite these advances, the irregular distribution of nonzero elements in matrices limits the efficiency of single-format solutions. Our approach segments matrices into independent sub-blocks and applies computation-specific optimizations such as column aggregation and adaptive format selection, achieving enhanced flexibility and performance across diverse matrices.

Load Balance: Load balancing is a critical factor in improving parallel sparse matrix-vector multiplication (SpMV) performance[1, 6, 20, 22, 23, 30, 33, 36, 37, 41, 43]. The uneven distribution of nonzero elements often causes workload imbalances, reducing efficiency. Various strategies have been proposed to address this issue, including the Merge-Path algorithm for fine-grained workload decomposition[36] and methods for balancing computation and communication loads in distributed systems[37]. Osama[41] et al introduces an abstraction model that decouples load balancing from task execution, using a hierarchical structure of work units, tiles, and sets, along with a programmable scheduling interface, to achieve GPU static and dynamic load balancing. Additionally, thread block-level strategies such as DTC-SpMM have significantly improved sparse matrix-matrix multiplication (SpMM)[18]. Despite these advances, achieving ideal thread block-level load balancing in SpMV remains challenging. Our proposed CB-SpMV format introduces a priority queue mechanism to adjust sub-block task assignments dynamically, significantly improves workload distribution and computational efficiency between different thread blocks, and offers new insights for optimizing sparse matrix computations.

6 Conclusion

In this work, we propose a novel cache-friendly SpMV computation method, CB-SpMV, which is designed to optimize data locality and computational efficiency through a block-based structure. Specifically, the matrix is partitioned into independent and uniformly sized sub-blocks, where various types of data within each sub-block are aggregated using virtual pointers to improve access patterns. To handle the challenges posed by varying sparsity levels across sub-blocks, we introduce a block-wise column aggregation strategy that consolidates sparse data and a format-adaptive selection mechanism that chooses suitable storage for each sub-block. Additionally, a thread block-level load-balancing algorithm is developed to mitigate the imbalance in processing non-zero elements across thread blocks. Experimental results on two GPUs demonstrate that our CB-SpMV method achieves higher cache hit rates and superior performance compared to state-of-the-art SpMV approaches.

Acknowledgments

The work is supported by the Natural Key Research and Development Program of China (2023YFB3002902) and National Natural Science Foundation of China (No.62322201 and U23B2020).

References

[1] Hartwig Anzt, Terry Cojean, Chen Yen-Chen, Jack Dongarra, Goran Flegar, Pratik Nayak, Stanimire Tomov, Yuhsiang M Tsai, and Weichung Wang. 2020. Load-balancing sparse matrix vector product kernels on gpus. *ACM Transactions on Parallel Computing (TOPC)* 7, 1 (2020), 1–26.

[2] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 781–792.

[3] Arash Ashari, Naser Sedaghati, John Eisenlohr, and P Sadayappan. 2014. An efficient two-dimensional blocking strategy for sparse matrix-vector multiplication on GPUs. In *Proceedings of the 28th ACM international conference on Supercomputing*. 273–282.

[4] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[5] Deshun Bi, Xiaowen Tian, Shengguo Li, and Dezun Dong. 2023. Efficiently Running SpMV on Multi-Core DSPs for Block Sparse Matrix. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 1912–1919.

[6] Haodong Bian, Jianqiang Huang, Lingbin Liu, Dongqiang Huang, and Xiaoying Wang. 2021. Albus: A method for efficiently processing spmv using simd and load balancing. *Future Generation Computer Systems* 116 (2021), 371–392.

[7] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. 2014. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput.* 40, 5-6 (2014), 47–58.

[8] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[9] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–11.

[10] Aydin Buluç and John R Gilbert. 2012. Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.

[11] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. 2011. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication.

In *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 721–733.

[12] Genshen Chu, Yuanjie He, Lingyu Dong, Zhezhao Ding, Dandan Chen, He Bai, Xuesong Wang, and Changjun Hu. 2023. Efficient Algorithm Design of Optimizing SpMV on GPU. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 115–128.

[13] Shane Cook. 2012. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

[14] NVIDIA Corporation. 2024. cuSPARSE: GPU-Accelerated Sparse Matrix Library. https://developer.nvidia.com/cusparse. Version 12.4.

[15] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[16] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. Alphasparse: Generating high performance spmv codes directly from sparse matrices. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[17] Ryan Eberhardt and Mark Hoemmen. 2016. Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 663–672.

[18] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 253–267.

[19] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse matrix-vector multiplication on GPGPUs. *ACM Transactions on Mathematical Software (TOMS)* 43, 4 (2017), 1–49.

[20] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. 2017. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw.* 43, 4, Article 30 (Jan. 2017), 49 pages. https://doi.org/10.1145/3017994

[21] Jianhua Gao, Weixing Ji, Zhaonian Tan, Yizhuo Wang, and Feng Shi. 2022. Taichi: A hybrid compression format for binary sparse matrix-vector multiplication on gpu. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3732–3745.

[22] Joseph L. Greathouse and Mayank Daga. 2014. Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 769–780. https://doi.org/10.1109/SC.2014.68

[23] Jihu Guo, Rui Xia, Jie Liu, Xiaoxiong Zhu, and Xiang Zhang. 2024. CAMLB-SpMV: An Efficient Cache-Aware Memory Load-Balancing SpMV on CPU. In *Proceedings of the 53rd International Conference on Parallel Processing*. 640–649.

[24] Haonan Ji, Huimin Song, Shibo Lu, Zhou Jin, Guangming Tan, and Weifeng Liu. 2022. Tilespmspv: A tiled algorithm for sparse matrix-sparse vector multiplication on gpus. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[25] Kwangrae Kim and Ki-Seok Chung. 2024. CAMPuS: Concurrent Acceleration of Memory Access and Parallel Processing in Near-Memory SpMV Architecture. *IEEE Access* (2024).

[26] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: an extended compression format for spmv on shared memory systems. *ACM SIGPLAN Notices* 46, 8 (2011), 247–256.

[27] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.

[28] Paolo Sylos Labini, Massimo Bernaschi, Werner Nutt, Francesco Silvestri, and Flavio Vella. 2022. Blocking Sparse Matrices to Leverage Dense-Specific Multiplication. In *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 19–24.

[29] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: Hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 238–252.

[30] Wenxuan Li, Helin Cheng, Zhengyang Lu, Yuechen Lu, and Weifeng Liu. 2023. Haspmv: Heterogeneity-aware sparse matrix-vector multiplication on modern asymmetric multicore processors. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 209–220.

[31] Yun Liang, Wai Teng Tang, Ruizhe Zhao, Mian Lu, Huynh Phung Huynh, and Rick Siow Mong Goh. 2017. Scale-free sparse matrix-vector multiplication on many-core architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 12 (2017), 2106–2119.

[32] Lifeng Liu, Meilin Liu, Chongjun Wang, and Jun Wang. 2015. LSRB-CSR: A low overhead storage format for SpMV on the GPU systems. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 733–741.

[33] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

[34] Yuechen Lu and Weifeng Liu. 2023. DASP: Specific Dense Matrix Multiply-Accumulate Units Accelerated General Sparse Matrix-Vector Multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[35] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[36] Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 678–689.

[37] Hongli Mi, Xiangrui Yu, Xiaosong Yu, Shuangyuan Wu, and Weifeng Liu. 2023. Balancing computation and communication in distributed sparse matrix-vector multiplication. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 535–544.

[38] Juan J Navarro, Elena García-Diego, Josep-L Larriba-Pey, and Toni Juan. 1996. Block algorithms for sparse matrix computations on high performance workstations. In *Proceedings of the 10th international conference on Supercomputing*. 301–308.

[39] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 68–78.

[40] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. 2022. TileSpGEMM: A tiled algorithm for parallel sparse general matrix-matrix multiplication on GPUs. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 90–106.

[41] Muhammad Osama, Serban D Porumbescu, and John D Owens. 2023. A programming model for GPU load balancing. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 79–91.

[42] James O'Neil and Daniel B Szyld. 1990. A block ordering method for sparse matrices. *SIAM J. Sci. Statist. Comput.* 11, 5 (1990), 811–823.

[43] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[44] Abdul Rehman Tareen, Marius Meyer, Christian Plessl, and Tobias Kenter. 2024. HiHiSpMV: Sparse Matrix Vector Multiplication with Hierarchical Row Reductions on FPGAs with High Bandwidth Memory. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 32–42.

[45] Richard W Vuduc and Hyun-Jin Moon. 2005. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications: First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005. Proceedings 1*. Springer, 807–816.

[46] Chenhao Xie, Jieyang Chen, Jesun Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin Barker, Mark Raugas, and Ang Li. 2021. Fast and scalable sparse triangular solver for multi-gpu based hpc architectures. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–11.

[47] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. *Acm Sigplan Notices* 49, 8 (2014), 107–118.

[48] Wangdong Yang, Kenli Li, and Keqin Li. 2018. A parallel computing method using blocked format with optimal partitioning for SpMV on GPU. *Journal of computer and system sciences* 92 (2018), 152–170.

[49] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up SpMV for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[50] Yichen Zhang, Shengguo Li, Fan Yuan, Dezun Dong, Xiaojian Yang, Tiejun Li, and Zheng Wang. 2023. Memory-aware optimization for sequences of sparse matrix-vector multiplications. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 379–389.