The Journal of Supercomputing
https://doi.org/10.1007/s11227-024-05949-6

Block-wise Dynamic Mixed-Precision for sparse matrix-vector multiplication on GPUs

Zhixiang Zhao[1] · Guoyin Zhang[1] · Yanxia Wu[1] · Ruize Hong[1] · Yiqing Yang[1] · Yan Fu[1]

Accepted: 29 January 2024

Abstract

Sparse matrix-vector multiplication (SpMV) plays a critical role in a wide range of linear algebra computations, particularly in scientific and engineering disciplines. However, the irregular memory access patterns, extensive memory usage, high bandwidth requirements, and underutilization of parallelism hinder the computational efficiency of SpMV on GPUs. In this paper, we propose a novel approach called block-wise dynamic mixed-precision (BDMP) to address these challenges. Our methodology involves partitioning the original matrix into uniformly sized blocks, with each block's size determined by considering architectural characteristics and accuracy requirements. Additionally, we dynamically assign precision to each block using a precision selection method that takes into account the value distribution of the original sparse matrix. We develop two distinct SpMV computation algorithms for BDMP: BDMP-PBP (Precision-based partitioning) and BDMP-TCKI (Tailored compression and kernel implementation). BDMP-PBP partitions the matrix into two independent matrices for separate computations based on block precision, offering flexibility for integration with other optimization techniques. Meanwhile, BDMP-TCKI focuses on achieving significant thread-level parallelism and memory utilization by tailoring an appropriate compressed storage format and kernel implementation for each block. We compare BDMP with NVIDIA's cuSPARSE library and three state-of-the-art SpMV methods, including SELLP, MergeBase, and BalanceCSR, using matrices from the University of Florida's SuiteSparse dataset collection. BDMP-PBP and BDMP-TCKI show average speedups up to 2.64× and 2.91× on Turing RTX 2080Ti, and up to 2.99× and 3.22× on Ampere A100. The results demonstrate that BDMP enables the optimization of computation speed without compromising the precision necessary for reliable results.

Keywords SpMV · GPU · Mixed-precision · Block-wise

Extended author information available on the last page of the article

Published online: 11 March 2024

1 Introduction

Sparse matrix-vector multiplication holds a great significance in the realm of sparse linear algebra. It plays a crucial role in the performance of iterative solvers, especially when dealing with large systems of linear equations and eigenvalue problems. Given that iterative methods might need hundreds or even thousands of matrix-vector products for convergence, SpMV's efficiency becomes paramount [1].

Optimizing SpMV is of exceptional significance in contemporary high-performance computing environments, particularly for modern applications. Among various parallel computing architectures, the graphics processing unit (GPU) stands out as the most prevalent and widely utilized [2]. Therefore, ensuring an efficient implementation of SpMV on GPUs is of paramount importance in maximizing the computational capabilities and performance of these applications. However, effectively harnessing SpMV operations on GPUs poses significant challenges due to irregular memory access patterns, excessive memory utilization, high bandwidth demands, and underutilization of parallelism.

Extensive research effort has been dedicated to optimizing SpMV on GPUs, including the application of novel sparse matrix formats, architecture-specific optimizations to existing formats, and automated performance tuning of matrix formats and parameters [3–17]. The emergence of novel sparsity matrix formats aims to address the irregular memory access patterns commonly encountered in SpMV operations. This entails reorganizing the data arrangement to better align with the GPU's memory access patterns, resulting in improved efficiency. Architecture-specific optimizations primarily focus on enhancing data proximity, minimizing memory latency, and fully exploiting the inherent parallelism of the GPU. Moreover, employing machine learning or heuristic techniques for automated performance enhancement enables the selection of the most suitable matrix format and parameter tuning for a given sparse matrix. These approaches are not mutually exclusive and can complement each other effectively.

Furthermore, recent independent research has focused on implementing mixed-precision strategies on GPUs to reduce data transfer between the host and device and effectively exploit the unique features of GPU hardware architecture, thereby enhancing the performance of the SpMV kernel [18–23].

In numerical computation, the cost associated with computation and communication scales with the size of the floating-point format. Communication includes activities such as accessing main memory, transferring data within or between computing cores. The cost of data transfers consists of a fixed access latency and the transfer time, which is determined by the ratio between the message size and the data transfer rate. Ignoring latency, the communication cost increases proportionally with the size of the floating-point format in terms of bits. Specifically, considering widely used IEEE754 double-precision (64-bit) and single-precision (32-bit) formats, the runtime difference for communication operations is approximately a factor of two.

Simultaneously, the cost of conducting arithmetic operations is strongly influenced by the hardware's support for computations in a particular floating-point

format. Beginning with the Turing architecture, NVIDIA introduced Tensor Cores, an innovative component designed to enhance performance for half-precision (FP16) and single-precision (FP32) computations. In subsequent GPU architectures, such as the Ampere architecture, NVIDIA has continued to optimize and refine Tensor Cores, further improving their capabilities for handling lower-precision computations. For instance, on the Turing RTX 2080 Ti, single-precision performance surpasses double-precision performance by a factor of approximately 32. Notably, Ampere NVIDIA's A100 GPU goes a step further by offering a single-precision performance that is an impressive 22 times higher than that of the RTX 2080 Ti. Due to the successive advancements in NVIDIA's GPUs, a notable discrepancy has arisen between the computational capabilities and memory bandwidth. Consequently, the expense associated with data access and communication has progressively amplified when compared to arithmetic operations. As a memory-bound algorithm, SpMV may potentially benefit from compressing all data in the cache before initiating communication with remote processors or main memory.

Hence, considering the variations in performance for computation and communication across different precision formats, an essential focus of current research is to improve the performance of numerical algorithms by carefully combining precision formats. The primary objective of these mixed-precision algorithms is to expedite application execution by leveraging lower-precision formats while maintaining high accuracy in the output.

It is important to note that mixed-precision optimizations and the aforementioned methods for enhancing SpMV operations on GPUs are not mutually exclusive but can be simultaneously employed to capitalize on their respective strengths and complement each other. By integrating these diverse approaches, we can strive to maximize the computational capabilities and performance of modern applications that heavily rely on SpMV in GPU-based high-performance computing environments.

In this paper, we present a novel approach called block-wise dynamic mixed-precision (BDMP), aiming to achieve high memory bandwidth utilization, achieve high thread-level parallelism, and reduce data transfer overhead. The BDMP method involves dividing the original matrix into uniformly sized blocks, strategically determined considering architectural characteristics and accuracy requirements. We introduce a precision selection method that analyzes the value distribution of the original sparse matrix, enabling us to optimize the computation process for increased efficiency without compromising the precision necessary for reliable results. Within the BDMP framework, we develop two distinct SpMV computation algorithms: BDMP-PBP (Precision-based partitioning) and BDMP-TCKI (Tailored compression and kernel implementation). BDMP-PBP facilitates partitioning the matrix into two independent sub-matrices for separate computations, based on their respective block precision. This design offers flexibility for seamless integration with other optimization techniques. On the other hand, BDMP-TCKI focuses on achieving significant thread-level parallelism and optimal memory utilization. This tailored approach substantially enhances parallel processing capabilities and memory bandwidth utilization, further accelerating the SpMV computations. The main contributions of this paper are as follows:

- We introduce the BDMP method and explore a set of optimizations that reduce memory transfer overhead, achieve high memory bandwidth utilization, and achieve high thread-level parallelism
- We propose a dynamic precision selection method based on the value distribution of the sparse matrix, determining the optimal precision for each sparse block.
- We develop two SpMV algorithms, BDMP-PBP and BDMP-TCKI, to enhance flexibility and memory utilization, respectively.
- We find that BDMP can achieve obvious speedups over state-of-the-art SpMV algorithms while maintaining a high degree of precision.

The rest of the paper is organized as follows: Section 2 provides a background on SpMV. Section 3 presents our proposed BDMP method, which includes a detailed explanation of the BDMP-PBP and BDMP-TCKI SpMV algorithms. In Sect. 4, we conduct a comprehensive evaluation to assess the performance benefits of our methods on the GPU. Finally, Sect. 5 concludes the paper by summarizing the findings and discussing potential directions for future research.

2 Backgrounds

2.1 Sparse matrix compression storage format

A sparse matrix, mostly filled with zeros, benefits from compressed storage formats, saving memory space and enhancing computational efficiency. The mainstream-compressed storage formats for sparse matrices are coordinate (COO), compressed sparse row (CSR), ELLPACK (ELL), and diagonal (DIA). Figure 1 shows these mainstream-compressed storage formats. The COO format is the most intuitive and simplest format, storing only the non-zero elements along with their corresponding indices and values. The arrays *val*, *row_index*, and *col_index* are responsible for storing the values of non-zero elements, their respective row indices, and column indices. The size of these arrays is equal to *nnz*, which represents the total number of non-zero elements in the sparse matrix. However, despite its simplicity and intuitiveness, the COO format encounters performance issues. These issues arise from challenges in thread decomposition and the unpredictability of coefficient usage in calculations. Additionally, the format requires increased memory usage due to the explicit storage of both the row and column indices for each non-zero element.

The CSR format is another commonly used approach that compresses the matrix data by removing zero entries. It consists of three arrays: *val*, *col_index*, and *row_ptr*. The *val* array stores all the non-zero elements in row-major order. *col_index* saves the column index for each non-zero entry, while *row_ptr* holds the starting index in the *val* array of the first non-zero element in each row. Although CSR improves memory usage and access patterns compared to COO, it still faces challenges, particularly in regard to inefficient parallelism due to potential load imbalance among rows and data locality issues in certain computations.

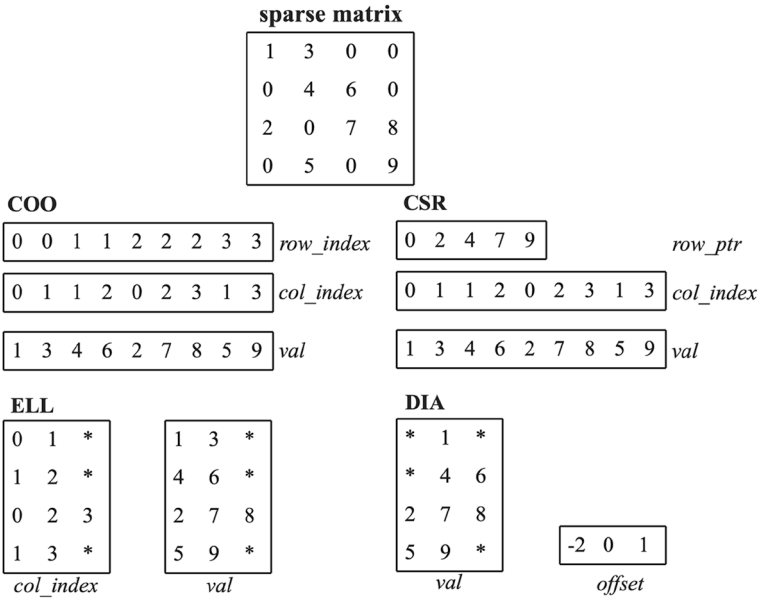

Fig. 1 Mainstream sparse matrix storage formats

The ELL format is known for its regular memory access pattern, enhancing performance for vector operations. It consists of two arrays: *val* and *col_index*. *val* stores non-zero elements row by row. Each row contains a fixed number of elements equal to the maximum number of non-zero entries in any row of the matrix. *col_index* stores their corresponding column indices. Despite its efficiency in vector operations, ELL has a significant limitation: its memory footprint can become substantial when dealing with matrices that have rows with widely varying numbers of non-zero elements.

The DIA format focuses on the diagonal structure of a sparse matrix, storing non-zero elements along diagonals in a dense matrix called *val*. Additionally, it uses another array, *offset*, to store the relative positions of these diagonals. While DIA can be memory-efficient for matrices with a significant number of non-zero diagonals, it performs poorly if the matrix lacks a strong diagonal structure, leading to memory wastage in storing many zero elements.

Besides, the HYB format combines the ELL and COO structures, effectively storing the initial K non-zeros per row in ELL (with zero padding for rows with fewer non-zeros). The remaining non-zeros are stored using the COO format. The parameter K is selected such that at least one-third of the matrix rows contain K or more non-zeros. This choice is guided by the distinct advantages of ELL's efficiency and COO's consistent performance. As a result, the HYB format offers significant improvements in kernel execution time across a wide range of sparse matrices. However, these advantages come at the cost of increased data organization complexity, more intricate program logic.

Advanced storage formats like the quad-tree and block compress formats offer unique advantages in handling sparse matrices. The quad-tree format recursively divides the matrix into four quadrants, each represented as a node in a tree structure. This recursive division continues until a specified criterion is met, such as the quadrant reaching a certain size or a level of sparsity. The array *node* stores information about each node, including its position in the matrix and the size of its quadrant. The array *data* contains the sparse matrix data for each leaf node, formatted according to specific application requirements, such as CSR. Finally, the array *child_indices* holds the indices of the child nodes for each node, facilitating efficient traversal of the tree structure [24–26]. This format is particularly effective for matrices with nonuniform sparsity patterns, allowing for adaptive compression based on local matrix properties. However, while efficient for matrices with nonuniform sparsity, the quad-tree format can lead to overhead in terms of both storage and computation. The recursive division of the matrix into quadrants adds complexity to the data structure, potentially resulting in increased memory usage and computational overhead, especially for matrices that do not exhibit a clear hierarchical sparsity pattern.

The block compress format groups non-zero elements of a sparse matrix into blocks, each stored as a dense sub-matrix, offering an efficient way to handle matrices with clustered non-zero elements. This format includes several critical arrays to manage the data effectively [3, 27–29]. The array *block_data* holds the dense matrix data for each block, containing all non-zero elements within that block. The array *block_position* indicates the starting position of each block in the original matrix, similar to the *row_ptr* in CSR format. Additionally, the array *block_col_ind* stores the column indices for each block, specifying the column location of the block in the matrix. This format is particularly beneficial when non-zero elements of a matrix are clustered in small regions, combining the storage efficiency of CSR for sparse areas with the computational efficiency of dense matrix operations for non-zero blocks. However, the block compress format excels primarily when non-zero elements exhibit block-like sparsity patterns. Its performance can degrade if the matrix does not show such patterns, as blocks may include many zero elements, leading to inefficient memory use. Additionally, determining the optimal block size can be challenging and often requires domain-specific knowledge, since the block size significantly affects both performance and storage efficiency.

2.2 Accuracy differences

In the realm of scientific computation, understanding the distinction between single and double precision is paramount, as it involves delving into the IEEE 754 standard for floating-point arithmetic. A single-precision floating-point number occupies 32 bits, comprising a sign bit, 8 bits dedicated to the exponent, and 23 bits for the significand. On the other hand, a double-precision floating-point number necessitates 64 bits, accommodating a sign bit, 11 bits for the exponent, and 52 bits for the significand. The larger bit allocation in double precision permits a broader spectrum of representable numbers with enhanced precision.

Accuracy evaluation often starts with the assessment of relative and absolute errors. The absolute error, denoted as E_{abs}, is the unaltered disparity between the true value (T) and the approximated value (A). This discrepancy is mathematically expressed in Eq. (1).

$$E_{\text{abs}} = |T - A| \tag{1}$$

Despite its straightforwardness, the absolute error does not consider the scale of the values under comparison. For instance, an absolute error of 1×10^{-3} might be inconsequential if the true value is approximately 1×10^{6}, but crucial if the true value is merely 1×10^{-3}. Hence, the absolute error does not always offer a precise portrayal of the accuracy of an approximation.

Similarly, the relative error, expressed in Eq. (2) and calculated as the absolute difference between the true value and the approximation divided by the true value, can also be misleading.

$$E_{\text{rel}} = \frac{|T - A|}{|T|} \tag{2}$$

Although the relative error considers the magnitude of the numbers being compared, it may still lead to misleading interpretations due to its sensitivity to the magnitudes of T and A.

Given these limitations, the concept of significant digits becomes crucial [30]. The significant digits of a numerical value provide meaningful information about its precision, independent of the magnitudes of the values. The criterion for significant digits can be formulated as Eq. (3).

$$E_{\text{rel}} < 5 \times 10^{-n} \tag{3}$$

Here, n represents the number of significant digits, which is usually rounded down to the nearest whole number

By considering Eqs. (2) and (3), we can infer that a single-precision float typically has approximately 6 to 7 significant digits, while a double-precision float offers approximately 15 to 17 significant digits. Consequently, significant digits are often the preferred method for quantifying accuracy differences between single-precision and double-precision numbers in scientific computation.

2.3 GPU architecture

In GPU architecture, the fundamental unit is the streaming multiprocessor (SM), with each GPU composed of a variable number of these SMs. Within each SM is several CUDA cores, the critical computational units are responsible for executing instructions and facilitating parallel computing. SMs utilize the single instruction, multiple data (SIMD) execution paradigm, managing groups of threads-termed warps that concurrently execute identical instructions on distinct data elements. Notably, a warp typically contains 32 threads and represents the basic unit for scheduling and task execution within an SM. The count of both

CUDA cores within each SM and the overall number of SMs varies across different GPU models and generations.

The memory architecture of GPUs plays a pivotal role in their design, incorporating various memory classifications with different levels of accessibility and speed. These categories include global, shared, local, constant, and texture memory. Global memory, while offering the most storage capacity, is the slowest in terms of access speed. It serves as the primary data repository for GPU computations and is accessible to all threads. In contrast, shared memory, despite its limited size, surpasses global memory in terms of speed. It is shared among a block of threads, facilitating efficient data exchange and interaction. Local memory is allocated on a per-thread basis and is used to store private variables and function call stacks. Constant memory is designed for frequent access to constants during GPU computations, acting as read-only memory. Texture memory, specifically tailored for texture mapping operations, enables high-efficiency texture sampling. To enhance GPU application performance, it is essential to effectively utilize these different memory types. Doing so ensures efficient data access, distribution, and caching, which in turn reduces memory latency and maximizes memory bandwidth utilization.

NVIDIA's Turing and Ampere architectures bring several advancements to GPU memory architecture, with Turing notably integrating L1 cache and shared memory for enhanced efficiency and adopting GDDR6 technology for higher bandwidth. Ampere furthered this evolution, improving both single- and double-precision floating-point performance and enlarging the L2 cache size, thus reducing memory access latency and enhancing computational performance.

Beginning with NVIDIA's Fermi architecture, GPUs have progressively included distinct pathways for single-precision (FP32) and double-precision (FP64) floating-point compute units, with significant advancements in parallel execution capabilities. This design evolution is evident in architectures like Pascal, Volta, Turing, and Ampere. Notably, the Volta architecture introduced separate FP32 and FP64 cores within the streaming multiprocessors (SMs), enabling simultaneous execution of single- and double-precision tasks without interference [31].

The Turing and Ampere architectures further refined this approach. For instance, the Turing architecture's 2080 Ti GPU demonstrates this capability with theoretical peak performances of 14.2 TFLOPS in FP32 and 0.44 TFLOPS in FP64. Meanwhile, the Ampere architecture's A100 GPU boasts 19.5 TFLOPS and 9.7 TFLOPS in FP32 and FP64 performances, respectively. This segregation of floating-point units enhances parallel efficiency in mixed-precision computing tasks and optimizes resource utilization, significantly boosting GPU performance in various computing environments [32, 33].

We target two NVIDIA platforms in this work: the RTX 2080 Ti (an older, but widely adopted GPU) and the A100 (a current GPU used in many high-performance computing centers). The architectural parameters for these GPUs can be found in Table 1.

Table 1 Architectural parameters for RTX 2080 Ti and A100

Device	RTX 2080 Ti	A100
GPU architecture	Turing	Ampere
Compute capability	7.5	8.0
Memory size (GB)	11	40
Memory bandwidth (GB/s)	616	1555
Theoretical peak FP16 (TFLOPS)	125	312
Theoretical peak FP32 (TFLOPS)	14.2	19.5
Theoretical peak FP64 (TFLOPS)	0.44	9.7
SMs	68	108
CUDA cores per SM	64	64
Tensor cores per SM	8	4
L1 size (KB)	64	192
L2 size (MB)	5.5	40

2.4 Related work

To address the irregular storage access patterns in sparse matrices, a block strategy is employed. This approach involves partitioning the sparse matrix into smaller, more manageable blocks. Subsequently, these blocks are rearranged to align more favorably with the memory access architecture of the GPU. This effective reduction in overall memory latency, combined with the efficient utilization of the GPU's parallel computing capabilities, is evident in various formats, including Sliced ELLPACK, SELL-C-σ, SELLP, BCCOO, BCCOO+, Merge, BalanceCSR, BRCSD, RBDCS, and DIA-adaptive [5, 16, 34–40]. Nevertheless, a challenge that remains is determining the optimal size for the row piece in these formats, which calls for further research to enhance their performance.

The hybrid compression strategy amalgamates multiple compression techniques to efficiently represent the sparse matrix. By applying diverse compression approaches to different matrix sections, a higher overall compression ratio is achieved. Consequently, this not only diminishes the memory footprint but also enhances memory access efficiency. Frequently, the hybrid compression strategy is combined with the block strategy to further optimize performance. Examples of such combinations include HDC, HYB-R, BCE, and TileSpMV [8, 41–43].

The index compression method aims to reduce the size of indices in sparse matrices. It utilizes encoding techniques to compress row and column indices, resulting in a noteworthy reduction in memory demands. These compressed indices can be processed more efficiently by the GPU, leading to decreased memory access latency and improved overall SpMV performance. DCSR, RPCSR, Delta coding, BRO, and BCOO serve as exemplars of index compression formats [7, 44–46].

The mixed-precision strategy is an effective approach for managing and optimizing the storage and computation of sparse matrices. This method employs varying numerical precision levels for different components of the matrix, thereby optimizing both storage and computation. It effectively reduces memory usage while maintaining sufficient precision for computations and takes advantage of the GPU's ability to process lower-precision data at higher speeds, enhancing computational

efficiency. Mixed-precision strategies find extensive applications in numerical linear algebra, including operations like matrix multiplication, factorization, linear systems, least squares, and eigenvalue and singular value decomposition. They incorporate key algorithmic ideas such as iterative refinement, precision adaptation to data, and exploiting mixed-precision block-fused multiply-add (FMA) operations. These algorithms strike a balance between lower-precision performance and higher-precision accuracy, achieving results comparable to fixed precision algorithms but at lower cost, or using slightly higher precision for improved accuracy. Significant works in this area include data-driven methodologies, the tSparse algorithm, research on mixed-precision block FMA operations, multiple-precision SpMV kernels in various matrix storage formats, and row-wise mixed-precision SpMV methods [18, 19, 21, 23, 47]. A survey highlights the performance benefits and insights into the numerical stability of these algorithms, marking a significant contribution to the field of numerical analysis and computation [48, 49].

It is worth noting that these strategies can, and often are, combined to further optimize both memory and computational performances in GPU-based sparse matrix operations.

3 Methodology

In this section, we present an extensive exposition of our BDMP method, which aims to minimize data transfer overhead and optimize memory bandwidth utilization. The BDMP methodology consists of two fundamental components: block partitioning and dynamic precision selection. Block partitioning is a process that divides the original matrix into uniformly sized blocks. This segmentation strategically considers two key factors: the architectural characteristics of the GPU and the accuracy requirements of the computation. Section 3.1.1 will provide a comprehensive introduction to the block partitioning process. On the other hand, dynamic precision selection allocates precision to each block dynamically, adjusting it based on the data characteristics in the sparse matrix. This approach aims to strike a balance between the computational accuracy offered by double-precision data types and the memory and computational efficiency of single-precision data types. Section 3.1.2 will provide detailed elaboration on the dynamic precision selection method.

Building on the BDMP framework, we introduce two SpMV algorithms: BDMP-PBP (precision-based partitioning) and BDMP-TCKI (tailored compression and kernel implementation). Within the BDMP-PBP algorithm, the original matrix is partitioned into two independent sub-matrices based on the block precision. This method allows for computations to be performed separately based on varying precision needs, subsequently reducing data transfer overhead and enabling smooth integration with other optimization techniques. A more detailed explanation of the BDMP-PBP algorithm will provide in Sect. 3.2.1. On the other hand, BDMP-TCKI focuses on enhancing parallelism and maximizing memory bandwidth utilization through tailored compression and kernel implementation. It employs a tailored compression strategy that selects the most suitable compression format for each block and

a specialized kernel implementation that utilizes warp-level kernel for different formats. Section 3.2.2 will provide detailed elaboration on the BDMP-TCKI algorithm.

3.1 BDMP method

3.1.1 Block partitioning

In the BDMP methodology, block partitioning involves dividing the original matrix into uniformly sized blocks, where the block height ω and block width σ are crucially important. These blocks are then stored in either single or double precision.

From a computational accuracy standpoint, both ω and σ should not exceed 32 as their maximum value. The primary reason for this limitation is the balance between parallelism and accuracy. While larger blocks offer more opportunities for parallel processing, they can also introduce numerical instability and increased errors, especially in sparse matrix operations [48].

$$\delta = \frac{|A_{\text{block}} - A_{\text{exact}}|}{|A_{\text{exact}}|} \tag{4}$$

When blocks are stored in single precision, the accuracy of each computation within the block may decrease due to the restricted precision of this format. This reduction in accuracy is due to the accumulation of round-off errors in floating-point computations, which become more pronounced with larger block sizes. Therefore, maintaining small block dimensions can help alleviate this accumulation of errors [49]. The degree of accuracy loss can be expressed in Eq. (4).

Where A_{block} and A_{exact} represent the results of the block-based computation and the exact computation, respectively. This ratio δ indicates the relative error, demonstrating the extent of precision loss with larger blocks.

From a hardware architecture perspective, when selecting block dimensions ω and σ, it is essential to take into account considerations related to memory access and computational efficiency. For memory access, the goal is to consolidate memory operations as much as possible to minimize overhead. Typically, the memory access unit is either 64B or 128B for NVIDIA GPUs. As a result, both the block height ω and block width σ should ideally be multiples of 16, regardless of whether the blocks are stored in single or double precision. This is because each floating-point number in single precision occupies 4B, and each double occupies 8B. Ensuring that both ω and σ are multiples of 16 enables coalesced memory requests, thus reducing the number of memory operations. Moreover, opting for excessively large values for ω and σ can result in a significant amount of redundant memory access, which can ultimately compromise computational efficiency.

Regarding computational considerations, it is crucial to take into account the SIMD execution unit of a GPU, known as a warp, which typically has a size of 32. Therefore, choosing block dimensions that are multiples of 16 ensures that each block can be efficiently processed by a warp. This approach helps prevent thread divergence and contributes to the overall improvement of computational efficiency. Additionally, large values for ω and σ can result in considerable redundant

computation, which wastes computational resources and diminishes the overall performance.

In summary, both the block height ω and width σ are critical in determining computational accuracy and addressing hardware considerations. From an accuracy standpoint, it is recommended that neither ω nor σ exceeds 32, ensuring precise block computations and minimizing error accumulation in floating-point operations. From a hardware perspective, to optimize memory access and computational efficiency, both ω and σ should ideally be multiples of 16. This alignment allows efficient processing by the GPU's SIMD execution unit. Considering all these factors, this article recommends selecting a block size of 16×16 to achieve both computational precision and efficiency.

3.1.2 Dynamic precision selection

After introducing the concept of block partitioning and its advantages in the BDMP methodology, we explore further into the precision of the data stored within these blocks. The choice between storing elements within the blocks as single-precision or double-precision values significantly affects the balance between computational accuracy and memory usage. This decision is based on whether the value of an element in a block lies within a range of $(-\lambda, \lambda)$. If it does, the element is stored in single precision; otherwise, it is stored in double precision. Consequently, the determination of the λ threshold emerges as a central component in the BDMP approach.

Building on this premise, we consider the traditional fixed threshold method, widely adopted in the field. This method sets a constant value for λ, typically $\lambda = 1$, regardless of the characteristics of the matrix value distribution. Although this method is straightforward and easy to implement, it exhibits inherent limitations. For matrices with a small value distribution range, the fixed threshold method could lead to storing all elements in single precision, compromising the overall computational accuracy. Conversely, for matrices with a larger value distribution range, all elements could be stored in double precision, needlessly increasing memory usage and impairing computational efficiency. The key limitation of the fixed threshold method is its inability to adapt to the specific characteristics of different matrices, lacking flexibility to handle diverse computational scenarios of various matrices. Hence, a more adaptive, flexible approach is needed to determine the precision of the elements stored in the blocks.

To address these limitations, we propose the dynamic threshold method. In contrast to the fixed threshold approach, this method derives the value of λ from the characteristics of the matrix value distribution itself. Specifically, λ is calculated as expressed in Eq. (5).

$$\lambda = f \times (\text{mean}(|A|) + 3 \times \text{std}(|A|)) \tag{5}$$

Here, mean($|A|$) and std($|A|$) represent the mean and standard deviation of the absolute values of the elements in matrix A, respectively, and f is a scaling factor.

The dynamic threshold method offers two key advantages. Firstly, it provides flexibility in adapting to matrices with diverse value distribution ranges,

Table 2 Comparison of matrix storage requirements including X and Y-vectors

Storage format	Storage cost
Single	$8NNZ + 12N + 4$ bytes
Double	$12NNZ + 20N + 4$ bytes
Mixed	$8NNZ_s + 12NNZ_d + 28N + 8$ bytes

overcoming the limitations associated with the fixed threshold method. Secondly, the threshold is adjustable, catering to computational applications that require either high precision or high speed.

In this work, we employ the dynamic threshold method with selected parameters $f = 0.5$. Our observation indicates that these chosen thresholds can achieve a high degree of precision while maintaining a high level of computational efficiency, as detailed in Sect. 4.3.

3.2 SpMV algorithm for DBMP BDMP

Before delving into the intricate details of the algorithms, we shall examine the storage demands of mixed-precision computation and compare them with those of single-precision and double-precision computation. To maintain a comprehensive perspective, we assume that the matrix A is stored in the compressed sparse row (CSR) format.

Table 2 provides an overview of the storage requirements for single-precision, double-precision, and mixed-precision matrices, including the X-vector in both single-precision and double-precision formats, as well as the y vector. In this context, N represents the number of rows, NNZ stands for the total number of nonzeros, and NNZ_S and NNZ_D denote the single-precision and double-precision nonzeros, respectively.

It is apparent from the table that when a significant proportion of the non-zeros can be represented using single precision, the amount of data transmitted through the memory system, when compared to a pure double-precision approach, is reduced by $4NNZ_s - 8N - 4$ bytes. As the number of NNZ_s increases, the savings in memory space become even more considerable.

Furthermore, Table 1 shows that GPUs have significantly higher single-precision computational prowess than double-precision capabilities. For example, the RTX 2080 Ti graphics card, based on the Turing architecture, displays a single-precision computational power that is 32 times greater than its double-precision performance. Similarly, the Ampere A100 graphics card exhibits a single-precision computational capacity twice that of its double-precision capacity. Consequently, by employing mixed-precision computation, we can fully leverage the single-precision computational capabilities of GPUs, thus enhancing computational efficiency.

3.2.1 BDMP-PBP algorithm

The BDMP-PBP algorithm partitions the original matrix into single-precision and double-precision sub-matrices based on block precision. SpMV computation is performed independently on these two sub-matrices, and the results are added together to obtain the final outcome.

One of the primary advantages of this approach is the improved parallelism, as the computations for single-precision and double-precision are completely independent, enhancing parallelism between single-precision and double-precision floating-point units. Moreover, it offers strong flexibility as, based on the characteristics of the two resulting sub-matrices, it can be integrated into any existing storage format or computation method.

Figure 2 shows an example of a 16×16 sparse matrix. This matrix is partitioned into two distinct sub-matrices: one highlighted in green, represented in single precision, and another in yellow, represented in double precision. As detailed in Sect. 2.1, the green sub-matrix exhibits a significant number of non-zero along its diagonals, making the DIA format the optimal choice for its storage. In contrast, the yellow sub-matrix has a uniform distribution of non-zero elements across its rows; thus, it is best stored using the ELL format.

Algorithm 1 CSR-vector algorithm for BDMP-PBP

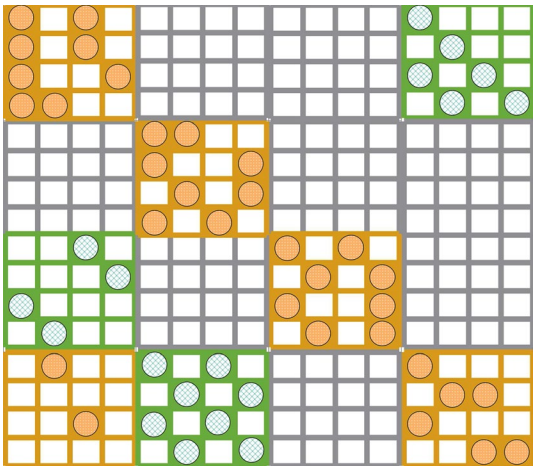

Fig. 2 BDMP-PBP partitioning of a sparse matrix of size 16×16

Input: val, col_index, row_ptr, x, y
Output: y
1: **for** $i = 0$ to $N - 1$ **do**
2: $y[i] = 0$
3: **for** $j = row\_ptr[i]$ until $row\_ptr[i + 1]$ **do**
4: $y[i] += val[j] * x[col\_index[j]]$
5: **end for**
6: **end for**

Given the extensive research on optimal storage formats, this paper refrains from exhaustive exploration of the most advantageous storage and computational methods for the resulting sub-matrices. For illustrative purposes, we opt for the widely recognized CSR-vector algorithm. We chose this due to its inherent generality, ensuring that the highlighted advantages stem from the mixed-precision approach and not from optimizations specific to storage formats. Algorithm 1 shows pseudocode for CSR-vector.

3.2.2 BDMP-TCKI algorithm

While the BDMP-PBP algorithm effectively reduces memory transfer costs, enhances parallelism, and improves both computational efficiency and versatility, there exists an opportunity for additional optimization. To further this effort, we introduce the BDMP-TCKI algorithm. The BDMP-TCKI algorithm is intricately designed to not only augment parallel processing capabilities but also ensure exceptional data locality and maximize memory bandwidth utilization.

We adopt a tailored compression strategy, selecting the optimal format for each matrix block. This selection process involves choosing from a range of formats including COO, CSR, ELL, and HYB (CSR and ELL), based on the distinctive characteristics of each block.

$$D = \frac{nnz}{\omega \times \sigma} \tag{6}$$

Specifically, we assess the block's sparsity using its density (D), which represents the proportion of non-zero elements to the total element count. The density of a block can be calculated as expressed in Eq. (6).

Additionally, to evaluate the balance of non-zero distribution across rows, we utilize the coefficient of variation (CV), which is the ratio of standard deviation to the average row length (ANZ).

The average row length can be calculated as expressed in Eq. (7). Here, nnz_i represents the number of non-zero elements in row i.

$$ANZ = \frac{1}{\omega} \sum_{i=0}^{\omega-1} \mathrm{nnz}_i \tag{7}$$

The coefficient of variation as expressed in Eq. (8).

$$CV = \frac{\sqrt{\frac{1}{\omega}\sum_{i=0}^{\omega-1}(\mathrm{nnz}_i - ANZ)^2}}{ANZ} \tag{8}$$

The metric CV provides a normalized measure of the dispersion in the distribution of non-zero elements across rows. In essence, a lower CV indicates a more uniform distribution of non-zero values, suggesting that rows have similar counts of these values. Conversely, a higher CV indicates that some rows may have significantly more non-zero values than others, implying a less balanced distribution.

For highly sparse blocks, the COO format undoubtedly offers the smallest memory footprint. Therefore, if the density D of the block falls below 0.02, the COO format is promptly considered the most optimal choice for that specific block.

If the density D is greater than 0.02, we rely on the CV values to identify the optimal compression format. We use the threshold values t_l (lower threshold) and t_u (upper threshold) to guide this selection process. If CV is below t_l, indicating that the number of non-zero entries in rows is relatively balanced, the ELL format is chosen for its better space efficiency. If CV exceeds t_u, indicating that the distribution of non-zero entries is irregular, the HYB format, consisting of ELL and CSR parts, is selected for potential efficiency. Between the thresholds t_l and t_u, we favor the CSR format, given that its general pattern typically provides superior performance.

Figure 3 presents the flowchart that elucidates the procedure for selecting the most suitable compression format based on the sparsity pattern of the block.

In our work, for selecting the optimal compression format, we set the threshold values t_u and t_l to 0.9 and 0.2, respectively. These threshold values were chosen based on extensive preliminary experiments and empirical observations, as detailed in Sect. 4.4.2.

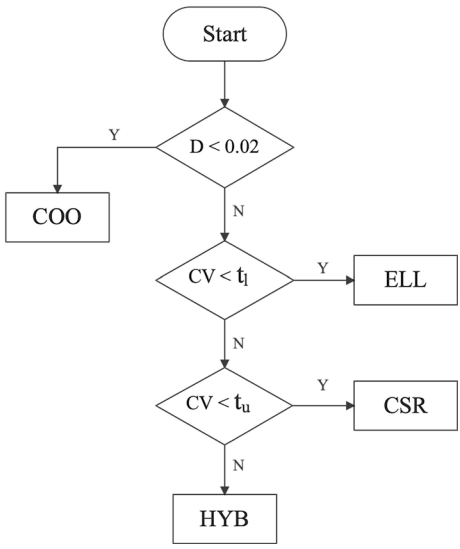

Fig. 3 The flowchart of the format selection for BDMP-TCKI

Moreover, we employ bit compression for row and column indices to minimize memory consumption. Our bit compression approach leverages the insight that the row and column indices of non-zero elements within a block are integers that fall below 16. Therefore, we can use 4 bits to represent the row and column indices of each non-zero element. To enhance load balancing in SpMV, each warp is assigned a fixed number of blocks. For elongated block rows, they are distributed among multiple warps. Subsequently, results are consolidated using atomic addition.

Additionally, we have meticulously designed the kernel implementations at the warp level to ensure that they align seamlessly with the compression format of each respective block. In the warp-level COO-SpMV algorithm tailored for highly sparse blocks, a warp, consisting of 32 threads, is responsible for all non-zero elements. These threads use atomic operations to ensure accurate result updates. After this, partial sums are atomically aggregated within the shared memory, as shown in Algorithm 2.

Algorithm 2 COO warp-level kernel for BDMP-TCKI

Input: $COOVal$, $COORowIdx$, $COOColIdx$, x, y
Output: y
1: **for** $i = 0$ until $nnz\ of\ the\ block$ **do**
2: $rowidx \leftarrow COORowIdx[i]$
3: $colidx \leftarrow COOColIdx[i]$
4: $atomicAdd(y[rowidx], COOVal[i] * x[colidx])$
5: **end for**

In the warp-level CSR-SpMV algorithm, a warp, composed of 32 threads, processes a block of 16 rows. Each pair of consecutive threads is assigned to a unique row. Before computation, a 16-entry segment of the vector *x* is loaded into the on-chip scratchpad shared memory to optimize data locality. After calculations, partial values of *y* are combined. This aggregation employs the *shuffle* operation, which is a warp-level instruction that allows data exchange among threads within the warp, as shown in Algorithm 3.

Algorithm 3 CSR warp-level kernel for BDMP-TCKI

Input: $CSRRowPtr$, $CSRColIdx$, $CSRVal$, x, y
Output: y
1: **for** $ti = 0$ to 31 **in parallel do**
2: $sum \leftarrow 0$
3: $ri \leftarrow ti/2$
4: $vi \leftarrow ti\%2$
5: **for** $j = CSRRowPtr[ri] + vi$ until $CSRRowPtr[ri+1]$ **do**
6: $csrCol \leftarrow CSRColIdx[j]$
7: $sum+ = s * x_{\text{warp}}[csrCol] * CSRVal[j]$
8: **end for**
9: $sum+ =$ __shfl_down_sync(**0xffffffff**, $sum, 1$)
10: $sum+ =$ __shfl_down_sync(**0xffffffff**, sum, ti)
11: **end for**

In the warp-level ELL-SpMV algorithm, 32 threads manage non-zero entries in a column-major format. Every thread in a half warp (comprising 16 threads) is assigned a row and completes its task once the ELL width is reached. To optimize memory access of the x vector, its segment is loaded into registers and accessed via register *shuffle* commands. In post computation, the results from each thread are aggregated, producing the final result for the block, as shown in Algorithm 4.

Algorithm 4 ELL warp-level kernel for BDMP-TCKI

Input: $EllVal$, $EllIdx$, $ellwidth$, x, y
Output: y
1: **for** $ti = 0$ to 31 **in parallel do**
2: $sum \leftarrow 0$
3: $elllen \leftarrow ellwidth * 16$
4: **for** $j = ti$ until $elllen$ **do**
5: $ellcol \leftarrow EllIdx[j]$
6: $x_{\text{gathered}} \leftarrow$ __shfl_sync(**0x0000ffff**, $x_{ti}, ellcol$)
7: $sum+ = EllVal[j] * x_{\text{gathered}}$
8: **end for**
9: **end for**

4 Evaluation

4.1 Experimental setup

Our experimental platform includes two NVIDIA GPUs: a GeForce RTX 2080 Ti (Turing architecture) and an A100 (Ampere architecture). Table 1 provides the architectural parameters of these GPUs. We expect that our results would generalize to most GPU systems.

To evaluate the effectiveness of the BDMP method, we compare it with NVIDIA's cuSPARSE library package and three other state-of-the-art methods: SELLP,

MergeBase, and BalanceCSR [35, 37, 50, 51]. To provide a summary, we averaged the results across 50 runs for each suite.

The evaluation dataset comprises all 2,851 sparse matrices in the SuiteSparse Matrix Collection [52], which is a comprehensive set of sparse matrices that is continuously updated and arises in numerous real-world applications. The numerical linear algebra community extensively uses this collection for developing and evaluating sparse matrix algorithms. From this collection, we excluded 35 matrices due to their large memory requirements. We also take a deeper look at a subset of matrices commonly used in performance evaluation in other works [3].

4.2 Accuracy evaluation method

In this study, we adopt significant digits (detailed in Sect. 2.2) as a metric to assess the accuracy of the BDMP algorithms.

For the general scenario where an $m \times n$ sparse matrix multiplies an n-element dense vector, we ascertain the number of significant digits in the resultant m-sized vector y. The number of significant digits for each element y_i is determined using Eq. (3). Subsequently, we calculate the proportion of elements that have more than seven significant digits relative to the total element count. As seven significant digits are considered to yield adequately precise computational outcomes, this proportion serves as a metric for gauging the accuracy of the SpMV computation, elucidated further in Eq. (9).

$$\text{Accuracy Ratio} = \frac{\sum_{i=1}^{m} I(\text{NumSD}(y_i) >= 7)}{m} \tag{9}$$

$$I(x) = \begin{cases} 1, & \text{if } x \text{ is true,} \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

where $I(x)$represents the indicator function, as defined in Eq. (10), and $\text{NumSD}(y_i)$ signifies the number of significant digits for the element y_i.

In this study, we establish a threshold of seven significant digits. This decision stems from the observation that computational accuracy is deemed high when the number of significant digits exceeds seven [18]. A superior accuracy ratio signifies enhanced computational precision. We deem a computation as adequately accurate if the accuracy ratio surpasses 95%.

4.3 Scaling factor threshold

We examine the influence of f on the performance and accuracy of the BDMP algorithms on 55 sparse matrices. Figure 4 illustrates the average speedup of BDMP-PBP and BDMP-TCKI compared to cuSPARSE CSR-SpMV across two GPUs. It also presents the compliance rate of BDMP-PBP on the A100 GPU for various threshold values of f.

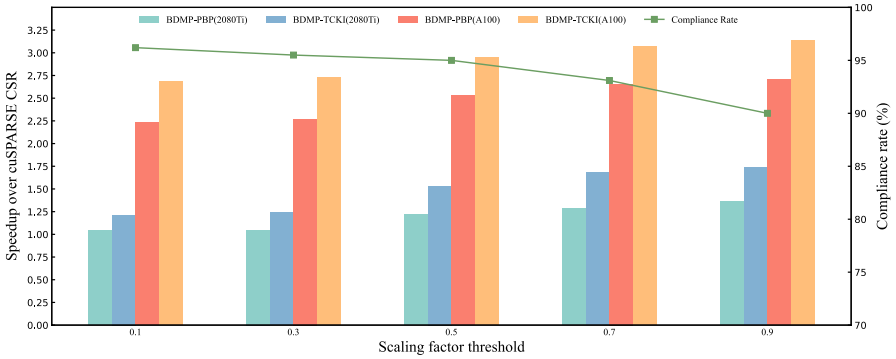

Fig. 4 Impact of scaling factor f on performance and accuracy

This compliance rate represents the proportion of matrices achieving an accuracy ratio above 95%. Both BDMP-PBP and BDMP-TCKI produce identical accuracy outcomes since they utilize the same precision selection methodology. Given that both GPUs exhibit architectural consistency and rigorously conform to IEEE floating-point standards, we opted to exclusively present the accuracy metrics of BDMP-PBP on the A100 GPU for conciseness.

From Fig. 4, we can observe an effect from alterations in the scaling factor f. Both GPUs experience an increase in speedup and a decrease in accuracy ratio as f increases from 0.1 to 0.9. This trend aligns with our predictions.

A higher f value signifies a larger threshold. This results in fewer elements in double precision, leading to a decrease in computational accuracy. However, an increasing f also means more elements are stored in single precision. This change encourages a rise in single precision floating-point operations and reduces memory transfer overheads, leading to enhanced performance.

In contrast, a lower f value results in more elements in double precision, which improves computational accuracy. However, this change also increases memory transfer overheads, leading to a decline in performance. Besides, the compliance rate for accuracy surpasses 95% when f is less than 0.5. This suggests a high level of computational accuracy for the majority of matrices.

Given these observations, we recommend setting the scaling factor threshold at 0.5. This value offers an equilibrium between accuracy and performance, and we employ it in subsequent experiments.

4.4 Performance evaluation

4.4.1 Evaluation of accuracy

We evaluated the accuracy of the BDMP method, focusing specifically on the BDMP-PBP algorithm on the A100 GPU, as detailed in Sect. 4.3. For this analysis, we selected the 23 matrices from a subset of matrices, as these matrices have sufficient computation to fully engage the floating-point units and involve significant

data movement. To demonstrate accuracy, we employed the concept of significant digits, as elaborated in Sect. 2.2.

Our findings are presented in Tables 3 and 4. Table 3 illustrates the number of significant decimal digits achieved using single precision, compared to double precision, for computing CSR-vector SpMV on the A100 GPU. Similarly, Table 4 shows the significant decimal digits achieved using mixed precision, relative to double precision, in the same context.

In both Tables 3 and 4, accuracy is quantified in terms of significant decimal digits, with each column representing a level of accuracy in significant decimals. Considering that single precision inherently limits accuracy to a maximum of six decimal digits compared to double precision, and mixed precision typically falls between single and double precision in accuracy, our analysis extends up to seven significant decimal digits. The 'ge8' column denotes instances where accuracy exceeds seven significant decimal digits, thereby indicating a higher accuracy level than single precision. A higher value in this column signifies greater

Table 3 Comparing significant digits in single versus double precision

Matrix	0	1	2	3	4	5	6	7	ge8
wiki-Vote	0	0	0	5	89	146	367	5263	2427
pdb1HYS	0	0	0	0	1.246	3343	15,264	12,830	3734
rma10	0	0	0	0	0	499	32,903	7943	5490
cant	0	0	0	0	76	393	45,096	9956	6930
consph	0	0	0	0	318	3286	58,493	12,398	8839
2cubes_sphere	0	0	34	146	1381	6328	70,889	14,031	8683
filter3D	0	0	11	247	1492	11,381	78,691	7376	7239
cage12	0	0	0	0	459	1334	89,083	17,997	21,355
shipsec1	0	0	2	285	1298	27,402	83,493	16,822	11,572
G2_circuit	0	0	437	4376	6372	8481	15,502	78,721	36,213
crashbasis	0	1	1294	3592	7310	8238	12,466	96,373	30,726
scircuit	0	0	0	0	539	8296	125,497	33,930	2736
cont-300	0	0	1	784	11,309	9403	118,230	18,709	22,459
hvdc2	0	0	0	0	0	2941	101,358	76,070	9491
co2010	0	0	0	0	0	753	125,447	65,481	9381
mac_econ_fwd500	0	0	0	0	529	2135	135,481	65,974	2381
bmw3_2	0	0	0	0	0	4218	135,391	76,161	11,592
Si87H76	0	0	0	0	0	7391	124,491	77,095	31,392
Lin	0	0	0	0	481	8293	151,392	85,540	10,294
offshore	0	0	0	0	0	1348	151,481	94,006	12,954
web-Stanford	0	0	0	0	84	4128	189,569	72,430	15,692
coAuthorsDBLP	0	0	0	21	1815	9634	173,568	102,628	11,401
ins2	0	0	0	0	0	2462	201,582	89,887	15,481

ge8 represents greater than or equal to eight

Table 4 Comparing significant digits in BDMP versus double precision

Matrix	0	1	2	3	4	5	6	7	ge8	CR (%)
wiki-Vote	0	0	0	1	47	101	234	4963	2951	95.38
pdb1HYS	0	0	0	0	397	1085	309	21,661	12,965	95.08
rma10	0	0	0	0	0	0	0	21,903	24,932	100
cant	0	0	0	0	0	1	2956	25,096	34,398	95.27
consph	0	0	0	0	0	0	2398	38,493	42,443	97.12
2cubes_sphere	0	0	0	0	151	2984	927	49,031	48,399	95.99
filter3D	0	0	0	1	342	1984	983	69,230	33,897	96.89
cage12	0	0	0	0	0	24	3593	30,934	95,677	97.22
shipsec1	0	0	0	0	32	4893	934	79,342	55,673	95.84
G2_circuit	0	0	257	3935	2452	1994	8953	59,389	73,122	88.28
crashbasis	0	1	467	2893	5109	4892	7931	57,292	81,415	86.68
scircuit	0	0	0	0	0	3503	3849	54,950	108,696	95.70
cont-300	0	0	0	356	3893	2943	8230	63,420	102,053	91.47
hvdc2	0	0	0	0	0	0	0	0	189,860	100
co2010	0	0	0	0	0	0	0	30,465	170,597	100
mac_econ_fwd500	0	0	0	0	0	0	3497	129,302	73,701	98.31
bmw3_2	0	0	0	0	0	0	0	0	227,362	100
Si87H76	0	0	0	0	0	0	0	0	240,369	100
Lin	0	0	0	0	0	0	0	42,463	213,537	100
offshore	0	0	0	0	0	0	0	0	259,789	100
web-Stanford	0	0	0	0	0	0	475	29,574	251,854	99.83
coAuthorsDBLP	0	0	0	0	0	34	1073	59,603	238,357	99.62
ins2	0	0	0	0	0	0	0	0	309,412	100

ge8 represents greater than or equal to eight; CR represents compliance rate

accuracy. Additionally, the sum of values across each row in these tables corresponds to the total number of rows in the respective sparse matrix.

The data in Tables 3 and 4 reveal that the BDMP method consistently provides a higher number of decimal digits of accuracy compared to single precision across all matrices. In the last column of Table 4, the accuracy of the BDMP method is quantified using the compliance rate, as defined in Eq. (9). It is observed that the compliance rate exceeds 95% for 20 of the 23 matrices, with three matrices falling below 95%. The lowest compliance rate is 86%. This decrease in accuracy is directly associated with the distribution of non-zero elements within the matrices. Specifically, the number of single-precision storage blocks in each matrix, determined by the distribution of non-zero elements, impacts the overall precision accuracy. A higher number of single-precision storage blocks in a matrix tends to decrease the accuracy of the results. Therefore, by examining the distribution of non-zero elements in matrices, we can predict, to a certain extent, the accuracy of the BDMP method.

Furthermore, we used the same random seed to populate the X-vectors and Y-vectors for all tests, observing no noticeable differences in the correctness of the final results. We also examined the impact of varying the magnitude of random values in the X-vector across the ranges [1, 10, 100] on accuracy. We observed no significant change in the accuracy of the final result.

4.4.2 Variation threshold for BDMP-TCKI algorithm

A reduced threshold t_l suggests a more even distribution of non-zero elements across rows, which favors the ELL format, thereby boosting performance. Yet, when t_l is too minimal, matrix blocks ideally suited for the ELL format get stored in CSR, causing inefficient compression and a consequent performance decline. Therefore, in our tests for the BDMP-TCKI algorithm, we restricted t_l values to between 0.1 and 0.3.

In relation to the threshold t_u, a more elevated value denotes a pronouncedly irregular distribution of non-zero elements. Setting t_u too low might lead to the storage of many matrix blocks in the HYB format, even when they are better suited to the CSR format. Such a choice can incur unnecessary storage and computational burdens. Consequently, our tests focused on t_u values spanning from 0.9 to 1.0, ensuring the HYB format is reserved for blocks with the most skewed distributions.

To assess the implications of these threshold decisions on SpMV performance, we executed tests using varying t_u and t_l values for a subset of matrices on the 2080Ti and A100 GPUs. These experimental outcomes are summarized in Fig. 5.

Fig. 5 reveals that BDMP-TCKI's performance on both GPUs is intricately tied to the threshold choices. Specifically, when t_u stands at 0.9 and t_l at 0.2, BDMP-TCKI realizes peak performance across the majority of matrices.

Notably, BDMP-TCKI's performance displays greater sensitivity to t_l than t_u. This heightened sensitivity to t_l stems from its direct role in matrix block selection for the ELL format, renowned for efficiently managing uniformly dispersed non-zero elements. An ill-chosen t_l can induce unwarranted storage overheads by compelling matrix blocks toward the CSR format when they might be better aligned with ELL. On the other hand, t_u dictates the switch between CSR and HYB formats; the performance implications of which are less drastic compared to the shift from ELL to CSR.

Based on these observations, we recommend setting the values of t_u to 0.9 and t_l to 0.2, as they yield near-optimal performance. We employ these threshold values in our subsequent experiments.

4.4.3 Performance comparison of SpMV

Figure 6 showcases the relative speedups of SELLP, MergeBase, BalanceCSR, BDMP-PBP, and BDMP-TCKI in comparison with the cuSPARSE CSR for the entire suite on both the 2080Ti and A100 GPUs.

Specifically, the upper subplot illustrates the outcomes on the NVIDIA RTX 2080Ti, whereas the lower subplot reveals the metrics for the NVIDIA A100 GPU. Within each of subplot, the left-hand subplots display scatter plots that

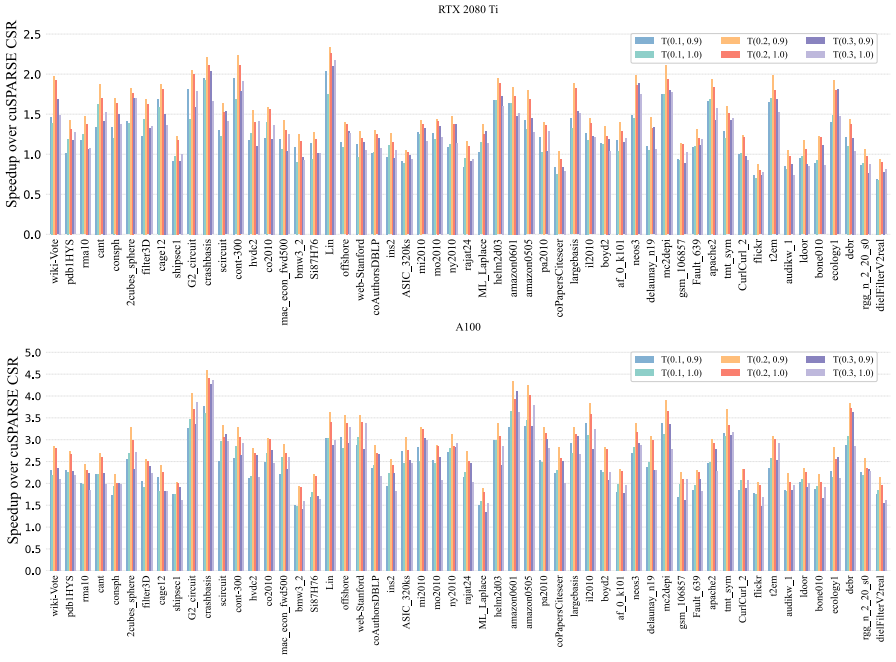

Fig. 5 The effect of t_u and t_l threshold on SpMV performance for BDMP-TCKI algorithm

compile the speedup data across all tested matrices. In contrast, the right-hand subplots employ box plots to offer a statistical overview.

Notably, BDMP-PBP and BDMP-TCKI consistently outperform their competitors on both GPU platforms. On the NVIDIA RTX 2080Ti, BDMP-PBP registers an average speedup of 2.04×, 2.64×, 1.14×, and 2.23 × over SELLP, MergeBase, BalanceCSR, and cuSPARSE CSR, respectively. Similarly, BDMP-TCKI exhibits speedups of 2.26×, 2.91×, 1.26×, and 2.46×. Switching attention to the NVIDIA A100 GPU, BDMP-PBP exceeds the performance of its aforementioned competitors with speedups of 1.57×, 2.29×, 1.38×, and 2.99× respectively, while BDMP-TCKI boasts speedups of 1.69×, 2.47×, 1.48×, and 3.22×.

For matrices with fewer non-zeros, the performance of all methods approximates that of CSR, primarily because of prevailing resource availability and launch overheads. Nevertheless, for matrices with many non-zeros, the performance of BDMP distinctly eclipses that of the other methods.

Figure 7 shows the speedups of SELLP, MergeBase, BalanceCSR, BDMP-PBP, and BDMP-TCKI over the cuSPARSE CSR, specifically focusing on a subset of matrices that are prevalent in performance evaluations in related literature. On average, the BDMP-PBP algorithm achieves speedups of 1.02×, 1.54×, 1.03×, and 1.21× over SELLP, MergeBase, BalanceCSR and cuSPARSE CSR, respectively on the NVIDIA RTX 2080Ti, and 1.29×, 1.94×, 1.31×, and 1.53× on the NVIDIA A100. The BDMP-TCKI algorithm achieves speedups of 1.31×, 1.67×,

Fig. 6 Speedup of SELLP, MergeBase, BalanceCSR, BDMP-PBP, and BDMP-TCKI versus cuSPARSE CSR, as run on Turing and Ampere GPUs

1.27×, and 2.53× on the NVIDIA RTX 2080Ti, and 1.53×, 1.95×, 1.47×, and 2.95× on the NVIDIA A100.

In Fig. 6, we can observe that, on both machines, BDMP-PBP shows only a slight performance improvement compared to other methods. This modest improvement can be attributed to BDMP-PBP's use on the CSR format for sub-matrices, rather than leveraging a more efficient compression format. The primary factor contributing to the enhanced performance of BDMP-PBP is its mixed-precision strategy. We anticipate that integrating more effective compression formats could potentially amplify BDMP-PBP's performance even further.

From both Figs. 6 and 7, it is evident that the BDMP algorithm performs better on the A100 compared to the 2080Ti. This enhanced performance can be attributed to the A100's doubled shared memory capacity and its 2.5x greater memory bandwidth, which reduce memory bound.

It is important to note the influence of the X-vector on speedups in our experiments. We maintained a consistent random seed for generating the values of the X-vector across all tests. Our examination included various random seeds to assess their impact on performance, and the results showed no significant

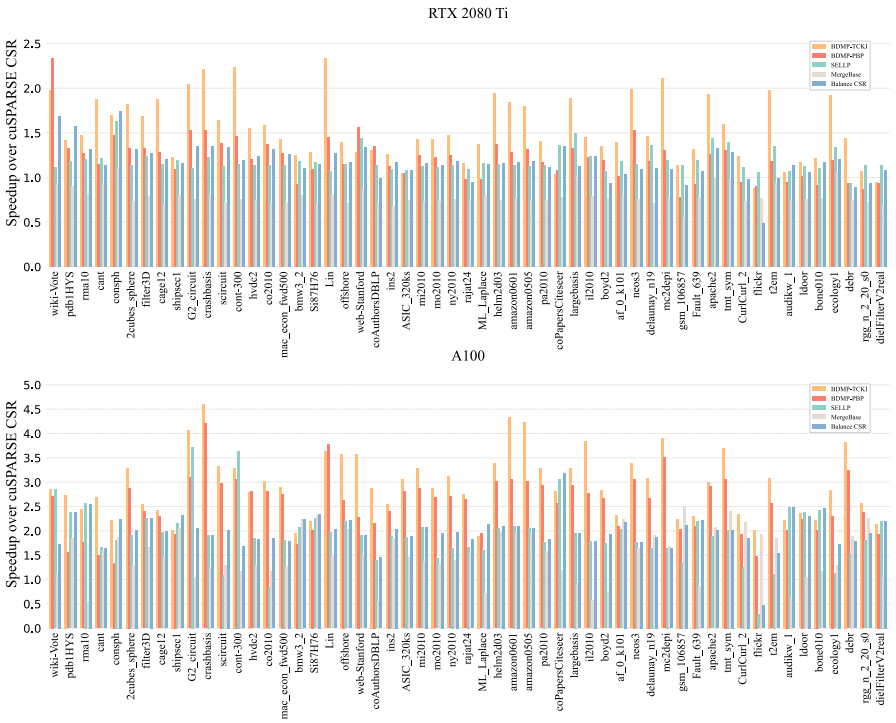

Fig. 7 Speedup of SELLP, MergeBase, BalanceCSR, BDMP-PBP, and BDMP-TCKI versus cuSPARSE CSR for subset of matrices

variances in speedups. Additionally, we explored the effect of escalating the magnitude of random values within the X-vector, utilizing ranges such as [1, 10, 100]. We observed that changes in the magnitude of these values did not notably alter the speedups. Similarly, we considered the role of the Y-vector in our experiments. Although the impact of the Y-vector on the performance of sparse matrix-vector multiplication is generally less pronounced compared to the X-vector and the matrix A, we conducted tests to ensure a comprehensive analysis. For the Y-vector, we generated its initial values using a consistent approach across all tests, opting for random numbers within an appropriate range that reflects typical use cases. It is noteworthy that, like the X-vector, variations in the Y-vector, including changes in its magnitude and distribution, showed minimal influence on the overall speedup.

From the aforementioned performance comparisons, we can discern the distinct advantages of BDMP-PBP and BDMP-TCKI in different aspects. When selecting between these algorithms, several key factors should be considered. BDMP-PBP offers significant flexibility and can be integrated with various matrix optimization methods, making it an invaluable tool for ongoing research in optimization strategies. In contrast, BDMP-TCKI excels in memory efficiency

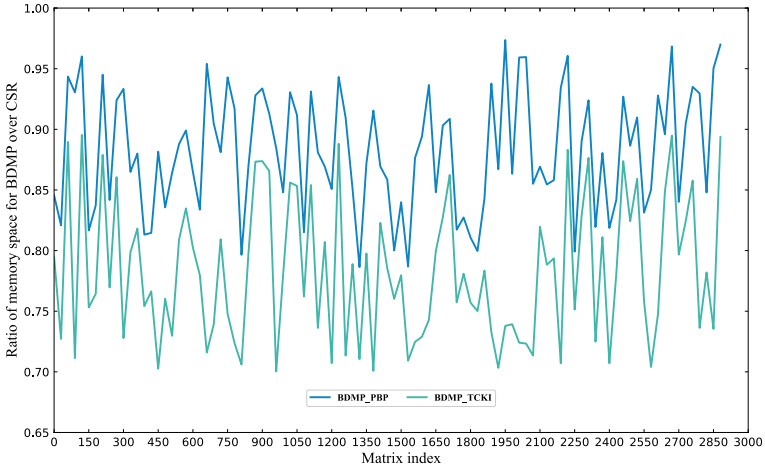

Fig. 8 Relative memory space costs for BMDP over CSR across the entire benchmark suite

and processing speed, making it a more suitable choice for applications where performance and efficiency are paramount. Despite their differences, both algorithms, as integral components of the BDMP framework, maintain comparable precision in their outputs. These findings suggest that for scenarios emphasizing efficiency, BDMP-TCKI is recommended, whereas the potential of BDMP-PBP for integration with other techniques makes it particularly suitable for research purposes in the field of matrix optimization.

4.5 Storage cost analysis

In Fig. 8, the memory space costs of the standard CSR format are compared with our BDMP-PBP and BDMP-TCKI. Compared to the standard CSR format, our BDMP-PBP (depicted by the blue line) consistently utilizes slightly less memory space. The space reduction is up to 22%, with an average of 12%. This outcome is anticipated, given that single-precision storage demands are less than those of double precision. The BDMP-PBP requires additional storage due to the necessity of a complete row pointer array for each block, bringing its memory consumption closer to that of the standard CSR format. It is worth noting that while BDMP-PBP uses the CSR format for both its single-precision and double-precision sub-matrices, this approach may not be the most efficient for each sub-matrix. Indeed, there exists an opportunity for enhanced memory optimization by selecting a storage format specifically adapted to the unique characteristics of each sub-matrix.

On the other hand, BDMP-TCKI (represented by the green line) consumes less memory than the standard double-precision CSR format. The space reduction is up to 31%, with an average of 22%. Although it requires additional storage for the block row pointer array and other supplementary arrays for each block, it effectively economizes memory. This optimization is achieved by judiciously selecting the most efficient storage format for each block and utilizing bit compression for both row and

column indices. In conclusion, both BDMP-PBP and BDMP-TCKI offer solutions that are more memory-efficient compared to the conventional CSR format.

4.6 Overhead analysis

Figure 9 shows the preprocessing overhead of both BDMP-PBP and BDMP-TCKI algorithms relative to a single SpMV operation. Comparative analysis indicates that BDMP-PBP consistently outperforms BDMP-TCKI in terms of preprocessing overhead across the entire dataset range. This disparity is attributed to the simpler preprocessing steps in BDMP-PBP, which involve matrix partitioning and precision selection. In contrast, BDMP-TCKI additionally requires selecting the optimal compression format for each block. To quantify, the preprocessing overhead of BDMP-PBP is on average equivalent to the time taken for 5 SpMV operations, while BDMP-TCKI's overhead is on average equivalent to 15 SpMV operations.

5 Conclusion

SpMV remains an essential operation in various scientific and engineering fields, especially in the domain of linear algebra computations. However, when executed on GPUs, SpMV encounters obstacles that hinder its computational efficiency. These challenges include irregular memory access patterns, extensive memory usage, high bandwidth requirements, and underutilization of parallelism.

In this paper, we introduce an innovative approach called BDMP, which aims to enhance the efficiency of SpMV on GPUs. Our methodology involves partitioning the original matrix into uniformly sized blocks, taking into account architectural characteristics and accuracy requirements. Additionally, we dynamically assign

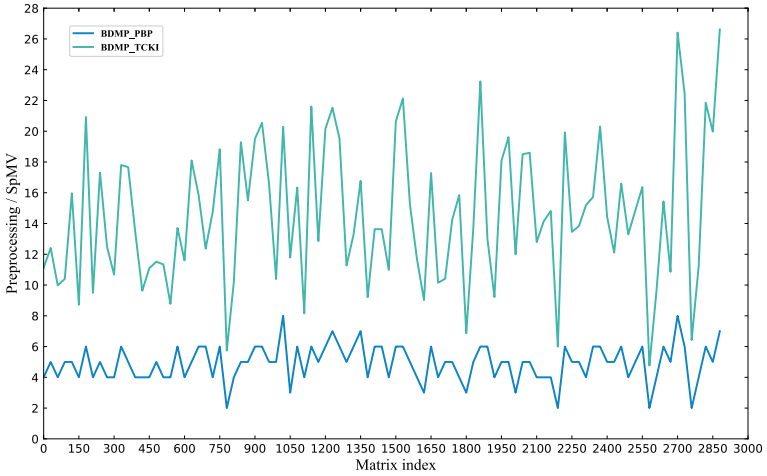

Fig. 9 Comparative preprocessing overhead of BDMP-PBP and BDMP-TCKI relative to SpMV operation

precision to each block using a strategy that considers the value distribution of the underlying sparse matrix. To implement BDMP, we develop two distinct SpMV computation algorithms: BDMP-PBP and BDMP-TCKI. BDMP-PBP focuses on partitioning the matrix into two independent matrices for separate computations based on block precision, while BDMP-TCKI optimizes thread-level parallelism and memory bandwidth utilization by tailoring a compressed storage format and kernel implementation specific to each block. We compare our BDMP method against cuSPARSE CSR and three other state-of-the-art SpMV implementations on both a Turing RTX 2080Ti and an Ampere A100, using matrices from the University of Florida's SuiteSparse. Our evaluation shows that, on Turing RTX 2080Ti, BDMP-PBP and BDMP-TCKI achieve average speedups ranging from 1.14× to 2.64× and 1.26× to 2.91×, respectively. On Ampere A100, the corresponding ranges are 1.38× to 2.99× for BDMP-PBP and 1.48× to 3.22× for BDMP-TCKI. Additionally, our BDMP-PBP and BDMP-TCKI algorithms can reduce memory consumption by up to 22% and 31%, respectively, compared to the standard CSR format. Our BDMP method maintains an acceptable level of precision slightly below double precision. These results validate the effectiveness of our BDMP techniques in improving computation speed without significantly compromising the necessary precision for reliable outcomes.

In our future endeavors, we aim to not only further refine our precision selection strategy and extend our methodology to accommodate lower precisions, but also to conduct comprehensive studies to accurately determine how different precision storage configurations impact computational accuracy. Enhancing and optimizing the BDMP-PBP algorithm remain a key objective. Its suitability for integration with various optimization techniques opens new avenues in matrix optimization research. We anticipate that advancements in BDMP-PBP will particularly arise from synergies with other strategies and a deeper understanding of matrix characteristics. Additionally, we plan to explore the potential of BDMP in other numerical algorithms involving sparse matrices, such as matrix–matrix multiplication and triangular solve.

References

1. Bell N, Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. ACM, Portland Oregon, pp 1–11. https://doi.org/10.1145/1654059.1654078
2. Nisa I, Siegel C, Rajam AS, Vishnu A, Sadayappan P (2018) Effective machine learning based format selection and performance modeling for spmv on gpus. In: 2018 IEEE International Parallel and

Distributed Processing Symposium Workshops (IPDPSW). IEEE, Vancouver, BC, pp 1056–1065. https://doi.org/10.1109/IPDPSW.2018.00164
3. Filippone S, Cardellini V, Barbieri D, Fanfarillo A (2017) Sparse matrix-vector multiplication on gpgpus. ACM Trans Math Softw 43(4):1–49. https://doi.org/10.1145/3017994
4. Tang WT, Tan WJ, Ray R, Wong YW, Chen W, Kuo S-h, Goh RSM, Turner SJ, Wong W-F (2013) Accelerating sparse matrix-vector multiplication on gpus using bit-representation-optimized schemes. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. ACM, Denver Colorado, pp 1–12. https://doi.org/10.1145/2503210.2503234
5. Kreutzer M, Hager G, Wellein G, Fehske H, Bishop AR (2014) A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. SIAM J Sci Comput 36(5):401–423. https://doi.org/10.1137/130930352
6. Zheng C, Gu S, Gu T-X, Yang B, Liu X-P (2014) Biell: a bisection ellpack-based storage format for optimizing spmv on gpus. J Parallel Distrib Comput 74(7):2639–2647. https://doi.org/10.1016/j.jpdc.2014.03.002
7. Tang WT, Tan WJ, Goh RSM, Turner SJ, Wong W-F (2015) A family of bit-representation-optimized formats for fast sparse matrix-vector multiplication on the gpu. IEEE Trans Parallel Distrib Syst 26(9):2373–2385. https://doi.org/10.1109/TPDS.2014.2357437
8. Yan CC, Yu H, Xu W, Zhang Y, Chen B, Tian Z, Wang Y, Yin J (2015) Memory bandwidth optimization of spmv on gpgpus. Front Comput Sci 9(3):431–441. https://doi.org/10.1007/s11704-014-4127-1
9. Li K, Yang W, Li K (2015) Performance analysis and optimization for spmv on gpu using probabilistic modeling. IEEE Trans Parallel Distrib Syst 26(1):196–205. https://doi.org/10.1109/TPDS.2014.2308221
10. Maggioni M, Berger-Wolf T (2016) Optimization techniques for sparse matrix-vector multiplication on gpus. J Parallel Distrib Comput 93–94:66–86. https://doi.org/10.1016/j.jpdc.2016.03.011
11. Godwin J, Holewinski J, Sadayappan P (2012) 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, London United Kingdom, pp 47–56. https://doi.org/10.1145/2159430.2159436
12. Yang W, Li K, Li K (2017) A hybrid computing method of spmv on cpu-gpu heterogeneous computing systems. J Parallel Distrib Comput 104:49–60. https://doi.org/10.1016/j.jpdc.2016.12.023
13. Elafrou A, Karakasis V, Gkountouvas T, Kourtis K, Goumas G, Koziris N (2018) Sparsex: a library for high-performance sparse matrix-vector multiplication on multicore platforms. ACM Trans Math Softw 44(3):1–32. https://doi.org/10.1145/3134442
14. AlAhmadi S, Mohammed T, Albeshri A, Katib I, Mehmood R (2020) Performance analysis of sparse matrix-vector multiplication (spmv) on graphics processing units (gpus). Electronics 9(10):1675. https://doi.org/10.3390/electronics9101675
15. Chen Y, Xiao G, Wu F, Tang Z, Li K (2020) tpspmv: a two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures. Inf Sci 523:279–295. https://doi.org/10.1016/j.ins.2020.03.020
16. Gao J, Xia Y, Yin R, He G (2021) Adaptive diagonal sparse matrix-vector multiplication on gpu. J Parallel Distrib Comput 157:287–302. https://doi.org/10.1016/j.jpdc.2021.07.007
17. Karimi E, Agostini NB, Dong S, Kaeli D (2022) Vcsr: an efficient gpu memory-aware sparse format. IEEE Trans Parallel Distrib Syst 33(12):3977–3989. https://doi.org/10.1109/tpds.2022.3177291
18. Ahmad K, Sundar H, Hall M (2019) Data-driven mixed precision sparse matrix vector multiplication for gpus. ACM Trans Archit Code Optim 16(4):1–24. https://doi.org/10.1145/3371275
19. Blanchard P, Higham NJ, Lopez F, Mary T, Pranesh S (2020) Mixed precision block fused multiply-add: error analysis and application to gpu tensor cores. SIAM J Sci Comput 42(3):124–141
20. Liu J (2022) Accuracy controllable spmv optimization on gpu. J Phys Conf Ser 2363(1):012008. https://doi.org/10.1088/1742-6596/2363/1/012008
21. Erhan Tezcan Torun T, Kosar F, Kaya K, Unat D (2022) Mixed and multi-precision spmv for gpus with row-wise precision selection. In: 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). IEEE, Bordeaux, France, pp 31–40. https://doi.org/10.1109/SBAC-PAD55451.2022.00014

22. Gao J, Ji W, Tan Z, Wang Y, Shi F (2022) Taichi: a hybrid compression format for binary sparse matrix-vector multiplication on gpu. IEEE Trans Parallel Distrib Syst 33(12):3732–3745. https://doi.org/10.1109/TPDS.2022.3170501
23. Isupov K (2022) Multiple-precision sparse matrix-vector multiplication on gpus. J Comput Sci 61:101609. https://doi.org/10.1016/j.jocs.2022.101609
24. Simecek I (2009) Sparse matrix computations using the quadtree storage format. In: 2009 11th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, Timisoara, Romania, pp 168–173. https://doi.org/10.1109/SYNASC.2009.55
25. Simecek I, Langr D, Tvrdik P (2012) Minimal quadtree format for compression of sparse matrices storage. In: 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE, Timisoara, Romania, pp 359–364. https://doi.org/10.1109/SYNASC.2012.30
26. Zhang J, Wan J, Li F, Mao J, Zhuang L, Yuan J, Liu E, Yu Z (2016) Efficient sparse matrix-vector multiplication using cache oblivious extension quadtree storage format. Future Gener Comput Syst 54:490–500. https://doi.org/10.1016/j.future.2015.03.005
27. Verschoor M, Jalba AC (2012) Analysis and performance estimation of the conjugate gradient method on multiple gpus. Parallel Comput 38(10–11):552–575. https://doi.org/10.1016/j.parco.2012.07.002
28. Choi JW, Singh A, Vuduc RW (2010) Model-driven autotuning of sparse matrix-vector multiply on gpus. ACM Sigplan Notices 45(5):115–126
29. Buatois L, Caumon G, Lévy B (2009) Concurrent number cruncher: a gpu implementation of a general sparse linear solver. Int J Parallel Emerg Distrib Syst 24(3):205–223. https://doi.org/10.1080/17445760802337010
30. Epperson JF (2021) An introduction to numerical methods and analysis, 3rd edn. Wiley, Hoboken
31. NVIDIA: Volta Architecture Whitepaper (2017). https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
32. NVIDIA: Turing Architecture Whitepaper (2018). https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
33. NVIDIA: Ampere Architecture Whitepaper (2021). https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf
34. Monakov A, Lokhmotov A, Avetisyan A (2010) Automatically tuning sparse matrix-vector multiplication for gpu architectures. In: Hutchison D, Kanade T, Kittler J, Kleinberg JM, Mattern F, Mitchell JC, Naor M, Nierstrasz O, Pandu Rangan C, Steffen B, Sudan M, Terzopoulos D, Tygar D, Vardi MY, Weikum G, Patt YN, Foglia P, Duesterwald E, Faraboschi P, Martorell X (eds) High performance embedded architectures and compilers, vol 5952. Springer, Berlin, pp 111–125. https://doi.org/10.1007/978-3-642-11515-8_10
35. Anzt H, Tomov S, Dongarra J (2014) Implementing a sparse matrix vector product for the sell-c/sell-c-σ formats on nvidia gpus. University of Tennessee, Tech. Rep. ut-eecs-14-727
36. Yan S, Li C, Zhang Y, Zhou H (2014) yaspmv: yet another spmv framework on gpus. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, Orlando Florida USA, pp 107–118. https://doi.org/10.1145/2555243.2555255
37. Merrill D, Garland M (2016) Merge-based parallel sparse matrix-vector multiplication. In: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, Salt Lake City, UT, USA, pp 678–689. https://doi.org/10.1109/SC.2016.57
38. Flegar G, Quintana-Ortí ES (2017) Balanced csr sparse matrix-vector product on graphics processors. In: Rivera FF, Pena TF, Cabaleiro JC (eds) Euro-Par 2017: parallel processing, vol 10417. Springer, Cham, pp 697–709. https://doi.org/10.1007/978-3-319-64203-1_50
39. Xia Y, Gao J, He G (2019) A parallel solving algorithm on gpu for the time-domain linear system with diagonal sparse matrices. In: Ren R, Zheng C, Zhan J (eds) Big scientific data benchmarks, architecture, and systems, vol 911. Springer, Singapore, pp 73–84. https://doi.org/10.1007/978-981-13-5910-1_7
40. He G, Chen Q, Gao J (2021) A new diagonal storage for efficient implementation of sparse matrix-vector multiplication on graphics processing unit. Concurr Comput Pract Exp. https://doi.org/10.1002/cpe.6230

41. Yang W, Li K, Liu Y, Shi L, Wan L (2014) Optimization of quasi-diagonal matrix–vector multiplication on gpu. Int J High Perform Comput Appl 28(2):183–195. https://doi.org/10.1177/1094342013501126
42. Yang W, Li K, Li K (2018) A parallel computing method using blocked format with optimal partitioning for spmv on gpu. J Comput Syst Sci 92:152–170. https://doi.org/10.1016/j.jcss.2017.09.010
43. Niu Y, Lu Z, Dong M, Jin Z, Liu W, Tan G (2021) Tilespmv: a tiled algorithm for sparse matrix-vector multiplication on gpus. In: 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, Portland, OR, USA, pp 68–78. https://doi.org/10.1109/ipdps49936.2021.00016
44. Willcock J, Lumsdaine A (2006) Accelerating sparse matrix computations via data compression. In: Proceedings of the 20th Annual International Conference on Supercomputing. ACM, Cairns Queensland Australia, pp 307–316. https://doi.org/10.1145/1183401.1183444
45. Kourtis K, Goumas G, Koziris N (2008) Optimizing sparse matrix-vector multiplication using index and value compression. In: Proceedings of the 5th Conference on Computing Frontiers. ACM, Ischia Italy, pp 87–96. https://doi.org/10.1145/1366230.1366244
46. Aliaga JI, Anzt H, Quintana-Ortí ES, Tomás AE, Tsai YM (2021) Balanced and compressed coordinate layout for the sparse matrix-vector product on gpus. In: Balis B, Heras DB, Antonelli L, Bracciali A, Gruber T, Hyun-Wook J, Kuhn M, Scott SL, Unat D, Wyrzykowski R (eds) Euro-Par 2020: parallel processing workshops, vol 12480. Springer, Cham, pp 83–95. https://doi.org/10.1007/978-3-030-71593-9_7
47. Zachariadis O, Satpute N, Gómez-Luna J, Olivares J (2020) Accelerating sparse matrix-matrix multiplication with gpu tensor cores. Comput Electr Eng 88:106848
48. ...Abdelfattah A, Anzt H, Boman EG, Carson E, Cojean T, Dongarra J, Fox A, Gates M, Higham NJ, Li XS, Loe J, Luszczek P, Pranesh S, Rajamanickam S, Ribizel T, Smith BF, Swirydowicz K, Thomas S, Tomov S, Tsai YM, Yang UM (2021) A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. Int J High Perform Comput Appl 35(4):344–369. https://doi.org/10.1177/10943420211003313
49. Higham NJ, Mary T (2022) Mixed precision algorithms in numerical linear algebra. Acta Numer 31:347–414. https://doi.org/10.1017/S0962492922000022
50. NVIDIA: cuSPARSE Library (2021). https://docs.nvidia.com/cuda/archive/11.2.1/cusparse/index.html
51. Aliaga JI, Anzt H, Grützmacher T, Quintana-Ortí ES, Tomás AE (2022) Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units. Concurr Comput Pract Exp. https://doi.org/10.1002/cpe.6515
52. Davis TA, Hu Y (2011) The University of Florida sparse matrix collection. ACM Trans Math Softw 38(1):1–25. https://doi.org/10.1145/2049662.2049663

Authors and Affiliations

Zhixiang Zhao[1] · Guoyin Zhang[1] · Yanxia Wu[1] · Ruize Hong[1] · Yiqing Yang[1] · Yan Fu[1]

✉ Yan Fu
fuyan@hrbeu.edu.cn

Zhixiang Zhao
641zzx@hrbeu.edu.cn

Guoyin Zhang
zhangguoyin@hrbeu.edu.cn

Yanxia Wu
wuyanxia@hrbeu.edu.cn

Ruize Hong
hongruize@hrbeu.edu.cn

Yiqing Yang
yangyiqing@hrbeu.edu.cn

[1] College of Computer Science and Technology, Harbin Engineering University, Harbin, China