Accelerating approximate matrix multiplication for near-sparse matrices on GPUs

Xiaoyan Liu1,2 · Yi Liu2 · Hailong Yang1,2 · Ming Dun3 · Bohong Yin2 · Zhongzhi Luan2 · Depei Qian2

Accepted: 22 January 2022 / Published online: 14 February 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract

Although the matrix multiplication plays a vital role in computational linear algebra, there are few efficient solutions for matrix multiplication of the near-sparse matrices. The Sparse Approximate Matrix Multiply (SpAMM) is one of the algorithms to fill the performance gap neglected by traditional optimizations for dense/sparse matrix multiplication. However, existing SpAMM algorithms fail to exploit the performance potential of GPUs for acceleration. In this paper, we present cuSpAMM, the first parallel SpAMM algorithm optimized for multiple GPUs. Several performance optimizations have been proposed, including algorithm redesign to adapt to the thread parallelism, blocking strategies for memory access optimization, and the acceleration with the tensor core. In addition, we scale cuSpAMM to run on multiple GPUs with an effective load balance scheme. We evaluate cuSpAMM on both synthesized and real-world datasets on multiple GPUs. The experiment results show that cuSpAMM achieves significant performance speedup compared to vendor optimized cuBLAS and cuSPARSE libraries.

1 Introduction

Generally, the matrices can be classified into dense matrix and sparse matrix according to the ratio of nonzero elements of the input matrices. Given a matrix $A \in \mathbb{R}^{N \times N}$, the number of nonzero elements is around $N^2$ for and $O(N)$ for dense and sparse algorithms, respectively. However, in real applications, there are a large number of matrices in the middle ground between dense and sparse matrices, a.k.a. near-sparse matrices, whose nonzero elements are between $N^2$ and $O(N)$ (or with 2% 70% nonzero elements [1]). Near-sparse matrices are widely used in the field of scientific...
computing, such as computational chemistry [2], quantum physics [3], and electronic structure calculation [4].

The near-sparse matrices are also used in emerging domains such as deep neural networks. Especially in convolutional neural networks (CNNs), the matrices of feature and weight are near-sparse [1, 5] due to weight pruning [6] and activation functions [7]. For example, the activation function of Rectified Linear Unit (ReLU) can lead to more than 50% sparsity of the feature matrices on average [5]. In CNNs, the convolution operations between feature and weight matrices are transformed to matrix multiplication using the \textit{im2col} algorithm [8]. In such case, various matrices involved in the deep neural networks are also near-sparse.

There is also a special class of matrices that are inherently near-sparse, the matrices with decay [9] (a.k.a. decay matrices), whose elements (values) decrease rapidly from diagonal to sides. The elements can be ignored if they are small enough and the corresponding matrices become near sparse. Due to the unique properties, many prior works are focusing on decay matrices, such as the decay rate [10], the left inverse [11, 12], high-dimensional statistics [13], and numerical analysis [14]. In addition, the decay matrices often appear in widely used matrix operations such as matrix inverse [10, 15], matrix exponential [16], Jacobi matrices [17], and others [18]. Moreover, decay matrices are commonly adopted in application domains such as quantum chemistry [9, 19] and quantum information theory [20–23].

On the other hand, both dense matrix multiplication (GENeral Matrix-to-matrix Multiply, GEMM) and sparse matrix multiplication (SpGEMM) are hardly efficient when applied to near-sparse matrices. The algorithm for GEMM focus on reducing the computation complexity. For example, Strassen's algorithm [24] and Williams’ algorithm [25] achieve $O(N^{2.8})$ and $O(N^{2.3727})$, respectively. In contrast, the complexity reduction is hardly useful for eliminating redundant computation of near-sparse matrices on the zero elements. On the other hand, the research for SpGEMM propose various storage formats such as CSR [26] to store the sparse matrices compactly. However, the sparse formats can hardly benefit the near-sparse GEMM due to its non-sparse nature. Therefore, both the GEMM and the SpGEMM have limited performance potential for near-sparse GEMM [1].

Fortunately, the approximation provides a good opportunity to boost the performance of near-sparse GEMM. For example, skipping the calculation of small enough elements of near-sparse matrices is a profitable way for performance acceleration. Based on such an idea, Sparse Approximate Matrix Multiply (SpAMM) [27, 28] has been proposed for accelerating the decay matrix multiplication. For matrices with exponential decay, existing research has demonstrated the absolute error of SpAMM can be controlled reliably [29].

In the meanwhile, with wide adoption in a large number of fields, GPUs have been proven with excellent speedup for matrix operations [30]. Especially with the advent of tensor core units provided by NVIDIA GPUs, mixed-precision techniques have been exploited to further accelerate matrix operations [31]. Although there are few research works optimizing SpAMM computation on CPUs [27–29], to the best of our knowledge, there is no GPU implementation available for accelerating SpAMM computation, especially exploiting the architectural features such as tensor
core and scaling to multiple GPUs. This motivates our work in this paper to redesign the SpAMM algorithm for better adaption to GPU architecture and propose corresponding optimizations to achieve superior performance compared to the state-of-the-art GEMM libraries. Specifically, this paper makes the following contributions:

- We propose $cuSpAMM^1$, a re-designed SpAMM algorithm tailored for GPU. Specifically, we adapt the calculation steps and the data access patterns of the SpAMM algorithm to the memory hierarchy and thread organization of GPU with increased parallelism and reduced memory accesses.
- We propose several optimization schemes such as blocking strategies for the calculation kernels and utilization of tensor core for accelerating the calculation. In addition, we present a scaling method to extend $cuSpAMM$ to multiple GPUs.
- We compare $cuSpAMM$ with highly optimized vendor libraries such as $cuBLAS$ [32] and $cuSPARSE$ [33] on GPUs. In addition, we evaluate on two real datasets from electronic structure calculation ($ergo$) and convolutional neural network (VGG13), to demonstrate the performance of $cuSpAMM$ on real-world applications.

The paper is organized as follows. In Sect. 2, we introduce the background of SpAMM algorithm, GPU optimizations and discusses the related works. In Sect. 3, we present our re-designed SpAMM algorithm $cuSpAMM$, and corresponding optimizations for performance acceleration on multiple GPUs. Section 4 compares our $cuSpAMM$ with the-state-of-art GEMM libraries and evaluates on two real-world datasets using $cuSpAMM$. Section 5 concludes this paper.

2 Background and related work

2.1 Decay matrix and SpAMM algorithm

A matrix is defined as the decay matrix when its elements decrease following the decay rate from the diagonal to the sides. The decay rate can be exponential or algebraical, formulated as $|A[i][j]| < c\lambda^{i-j}$ and $|A[i][j]| < c/(|i-j|^\delta + 1)$, respectively, where $A[i][j]$ is the index of the element in matrix $A$. The $|i-j|$ is the separation and can be replaced by other index-based distance function of the matrix or physical distance such as $|\mathbf{r}_i - \mathbf{r}_j|$ in non-synthetic cases [29]. By mathematical definition, the decay matrix is quite dense due to few zero elements. However, under certain conditions (e.g., elements less than the threshold), a number of elements in the decay matrix can be treated as zeros, which renders the matrix as near-sparse.

SpAMM is an approximate matrix multiplication method that can be used on decay matrices. The problem solved by SpAMM can be described as $C = \alpha AB + \beta C$, where $\alpha$ and $\beta$ are parameters, and $A$, $B$, and $C$ are the matrices with exponential decay or fast algebraical decay [27]. For convenience, the rest of the paper takes

---

$^1$ $cuSpAMM$ is open-sourced at https://github.com/buaa-hipo/cuSpAMM
\( \alpha = 1, \beta = 0, \) and the square matrices \( N \times N. \) Besides, \( \tau \) is a parameter for controlling the extent of approximation. The algorithm divides the input into quad-tree recursively, depicted in Eq. (1). The sub-matrix in lowest level is the minimal matrix size of recursion. Then, the algorithm performs the multiplication of sub-matrices recursively. The density of sub-matrices is measured by the Frobenius norm (F-norm), depicted in Eq. (2). Algorithm 1 shows the pseudo-code of SpAMM. The algorithm performs the multiplication only if the product of norms from two sub-matrices is no smaller than the parameter \( \tau \) (line 8 and line 13).

\[
A' = \begin{bmatrix}
A'_{0,0} & A'_{0,1} \\
A'_{1,0} & A'_{1,1}
\end{bmatrix}
\]

\[
||A'||_F = \sqrt{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} (A'_{i,j})^2}
\]

**Algorithm 1 SpAMM algorithm**

1: Input: Matrices \( A, B, \) parameter \( \tau \)
2: Output: Multiplication result \( C \)
3: if lowest level then
4: return \( C = AB \)
5: end if
6: for \( i=0 \) to 1 do
7: for \( j=0 \) to 1 do
8: if \( ||A_{i,0}||_F ||B_{0,j}||_F \geq \tau \) then
9: \( T_0 = SpAMM(A_{i,0}, B_{0,j}, \tau) \)
10: else
11: \( T_0 = 0 \)
12: end if
13: if \( ||A_{i,1}||_F ||B_{1,j}||_F \geq \tau \) then
14: \( T_1 = SpAMM(A_{i,1}, B_{1,j}, \tau) \)
15: else
16: \( T_1 = 0 \)
17: end if
18: \( C_{i,j} = T_0 + T_1 \)
19: end for
20: end for
21: return \( C \)
2.2 GPU architecture and optimization

2.2.1 GPU architecture

The CUDA programming paradigm provides a classic definition of NVIDIA GPU architecture [34], with the thread and memory organized hierarchically. We focus on the V100 architecture [35], CUDA 10, and CUDA 11.

**Thread hierarchy**—Threads are organized at four levels with different granularity including thread, warp, block, and grid. One warp usually contains 32 threads. The warp is the most basic unit for calculation execution and hardware scheduling if the warp contains no branch [35]. The block consists of multiple threads, and the threads within the same block can be synchronized. The grid consists of multiple blocks, and the blocks within the grid are executed in the SIMT (single instruction, multiple threads) fashion.

**Memory hierarchy**—The memory hierarchy can be divided into three levels, including register level, shared memory level, and global memory level. Each thread has private registers, which is the fastest but also the most limited storage on GPU. The second fastest memory is the shared memory for each block. The shared memory is private to the thread block. The lifetime of shared memory same as the thread block. The global memory level consists of global memory, texture memory, and constant memory that hosts the data transferred from the CPU. In V100, the texture memory and constant memory are stored in global memory [36]. The global memory can be bound to texture memory and constant memory by CUDA API.

2.2.2 GPU optimization for matrix multiplication

We briefly summarize the commonly used optimization strategies for high-performance matrix multiplication on GPU.

**Architecture targeted optimizations**—The blocking strategies [37] partition the matrices and performs calculations across GPU memory hierarchy. Memory prefetching strategies [38] utilize guiding statements for writing memory, explicitly creating buffers and calling primitives, which overlaps the data movement with computation. Register optimization strategies [30] achieve better performance by reducing the active registers and minimizing access to high-latency memory such as global memory. Other optimization approaches that avoid bank conflict and tune hyper-parameters [39] (e.g., block size) are also useful for accelerating GEMM on GPU.

**Tensor core adoption**—The tensor core [40] introduced from Nvidia Tesla V100 GPU has already been explored in many fields for further performance optimization such as linear algebra [41] and weather simulation [42] recently. In general, tensor core is a computation unit for Matrix-Multiply-Accumulate (MMA), formulated as \( D_{m \times k} = A_{m \times n} \times B_{n \times k} + C_{m \times k} \), where the maximum number of matrix elements is 256. For the input matrix in FP16, the \( A, B \) must be in FP16 precision, while matrix \( C, D \) can be in FP16 or FP32 precision. The programming of tensor core is based on a special data structure named *fragment*, which stores the computation data for the
tensor core. The *threads* in each *wrap* operate on the *fragments* to perform MMA calculation on tensor core.

### 2.3 Approximate matrix multiplication for decay matrix

The SpAMM is one of the approximate algorithms for decay matrix multiplication which first introduced by Challacombe et al. [27]. They proved that the time complexity of the algorithm is \( O(N\log N) \) at worst for matrices with a sufficient decay rate. However, Challacombe et al. [27] only presented empirical experiments on error behavior, other than a detailed analysis on numerical error. In addition, they provided a limited study on the decay matrices generated by the Heaviside matrix function from electronic structure domain [3]. Artemov et al. [29] studied the SpAMM algorithm with the matrices of exponential decay. They proved the absolute error behavior of SpAMM is \( \|E_{\text{exact}}\|_F = O(N^{1/2}) \times O(p^{p/2}) \), where \( p < 2 \). Another approximate algorithm for decay matrix multiplication is to truncate the decay matrix and use SpGEMM to calculate the product. For example, Maia et al. propose a SpGEMM algorithm based on MOZYME format and use GPU for acceleration [43]. However, when the nonzero ratio is big (or truncation threshold is small), the SpGEMM will even be slower than GEMM because the former introduces the extra overhead of index mapping and has poor locality [1]. Also, the overhead of format conversion is expensive, while SpAMM does not need to change the data format.

Until recently, the performance optimization of SpAMM mainly focuses on CPUs. Bock et al. [27] reformed the SpAMM computation recursively and implemented the algorithm in parallel. In addition, they used hashed (linkless) tree structure [44] to map the data and computation of sub-matrices. Moreover, the authors adopted hardware prefetching and optimized the performance using Intel Math Kernel Library (MKL) and AMD Core Math Library (ACML). Inspired from N-Body method [45], Bock et al. [28] and Artemov et al. [29] leveraged parallel programming models such as Charm++ [46] and Chunks/Tasks [47] to accelerate SpAMM, respectively. However, none of the existing works has optimized SpAMM on GPUs, which leaves the widely available GPU performance unexploited.

### 2.4 High performance GEMM

The high performance GEMM libraries on GPU have inspired our optimizations to SpAMM. *cuBLAS* and *cuSPARSE* are widely used high performance GEMM libraries heavily optimized by NVIDIA. Several optimization strategies have been proposed in *cuBLAS* and *cuSPARSE*. For example, the blocking strategies, instruction-level parallelism, parameter tuning, and tensor core acceleration have been successfully adopted for optimizing dense matrix multiplication in *cuBLAS*. For sparse matrix multiplication in *cuSPARSE*, various matrix storage formats such as COO, CSR, and CSC are supported to optimize both sparse-sparse and sparse-dense matrix multiplication. *spECK* [48] proposes a SpGEMM algorithm that can achieve more efficient memory utilization and better load balance for irregular sparse matrix. The authors use the GPU merge path algorithm and GPU-specific optimizations for
acceleration. ACSpGEMM [49] present a dynamic SpGEMM algorithm on GPU which adaptively re-purposes GPU threads and maximizes on-chip memory usage. Both spECK and ACSpGEMM can achieve better performance compared with cuSPARSE which proved by their experiments.

Moreover, Huang et al. [50] reviewed the blocking strategies of GEMM in cuBLAS and used tensor core to optimize the Strassen algorithm. Combined with optimizations such as software prefetching and parameter tuning, their implementation achieves 1.11× speedup on matrix multiplication compared to cuBLAS. Mukunoki et al. [51] evaluated the parallelized linear algebra kernels with multiple data precisions on GPUs. Ryoo et al. [30] summarized the general principles of matrix multiplication optimizations on GPU. Abdu et al. [52] used tensor core as matrix multiply-accumulate unit and proposed a chained reduction strategy. The above works have inspired the redesign and further optimization of SpAMM algorithm on GPU. Except the optimization of a single GEMM, batched GEMM [53] is a widely adopted technology that addresses small scale matrix multiplication and has already been integrated in vendor library such as cuBLAS. However, batched GEMM is not applicable to SpAMM due to its time-consuming reduction operations for accumulating the final result.

3 Methodology and implementation

In this section, we will first give an overview of our re-designed SpAMM algorithm tailored for GPU, cuSpAMM. Then, we introduce the design of two important kernels in cuSpAMM, which adopts several optimization strategies as well as leverages tensor core to optimize the performance. In addition, we scale our implementation to multiple GPUs for processing larger matrices. Finally, we propose load balance and accuracy searching optimizations that further improve the performance of cuSpAMM.

For the convenience of illustration, we use the following notations. The input of the algorithm is matrices $A, B \in \mathbb{R}^{N \times N}$, where $A, B$ are decay matrices, and $\tau$ is the approximation threshold. The output matrix is $C$. For optimization, we divide the input matrix into sub-matrices with size of $LoNum \times LoNum$. We use $BDIM = N / LoNum$ to denote the number of sub-matrices per row/column, where $N$ is divisible by $LoNum$ and $BDIM$ is for Block DIMension. The coordinates of the sub-matrix are represented by a square bracket. For example, $A[i, j]$ represents the sub-matrix with the starting index of $A[i \times LoNum][j \times LoNum]$. To avoid incomplete division, the matrices are padded with zeros to satisfy the above assumption. We assume that the minimal matrix can always fit in the shared memory. We use $LoNum=32$ in our implementation.

3.1 Overview of cuSpAMM

Figure 1 shows the design overview of cuSpAMM. To eliminate the GPU-unfriendly recursion in original SpAMM as well as exploit higher parallelism, we
redesign the algorithm composed of two kernels, *Get-norm kernel* and *Multiplication kernel*. The first kernel is responsible for calculating the (F-norm for all sub-matrices in $LoNum \times LoNum$), and the second kernel decides whether to multiply the matrices depending on the F-norm results from the first kernel. The array used to record the F-norm values is $normmap$, where $A_{\text{normmap}}[i][j] = \|A[i,j]\|_F$ and $B_{\text{normmap}}[i][j] = \|B[i,j]\|_F$ for matrix $A$, $B$, respectively. The re-designed cuSpAMM algorithm is equivalent to the original SpAMM algorithm, because they both perform calculations on the sub-matrices that satisfy the F-norm threshold ($\tau$). In addition, we propose several optimizations for the above two kernels, and utilize tensor core for further performance acceleration. Specifically, we apply the blocking optimization to cuSpAMM across the following memory hierarchies. At *device level*, we partition matrices $A$, $B$, $C$, $A_{\text{normmap}}$ and $B_{\text{normmap}}$ in GPU global memory. We use texture cache to load the read-only matrix $A$ and $B$ for acceleration [54]. At *block*, *warp*, and *thread level*, we partition the intermediate results in corresponding memory hierarchy. The details of blocking optimization are presented in the following sections.

### 3.2 Get-norm kernel

The *get-norm* kernel is responsible for calculating the F-norm (based on Eq. (2)) results for all sub-matrices. Each block of *get-norm* kernel calculates the F-norm of one sub-matrix. Considering the computation characteristics of F-norm, we adopt the reduction algorithm for better parallelization, as shown in Fig. 2a. Firstly, each thread takes an element from the input matrix, calculates its square value, and stores the results into shared memory. Then, the thread block performs the reduction on the shared memory. To optimize reduction, we adopt sequential addressing instead of stride addressing to avoid the bank conflict on shared memory.

To further accelerate the performance with input matrices in FP16 precision, we use tensor core as MMA unit for reduction. Equations. (3) and (4) show the
reduction of \( m^2 \) elements, where \([1]_{m \times m}\) and \([0]_{m \times m}\) represents a square matrix composed of 1 and 0, respectively, \( x_{11}, x_{12}, \ldots, x_{mm} \) are the data waiting for summation. After two MMA operations, the reduction results are stored in matrix \( D' \). This optimization can accelerate the reduction calculation compared to the traditional reduction on GPU [52]. Finally, the thread 0 writes the result back to \( \text{normmap} \).

\[
D = [1]_{m \times m} \times \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mm} \end{bmatrix} + [0]_{m \times m} \tag{3}
\]

\[
D' = \begin{bmatrix} \sum_{i=1}^{m} x_{ii} & \cdots & \sum_{i=1}^{m} x_{1i} \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{m} x_{mi} & \cdots & \sum_{i=1}^{m} x_{mi} \end{bmatrix} \times [1]_{m \times m} + [0]_{m \times m} \tag{4}
\]

Meanwhile, we apply additional optimizations to further boost the performance. Firstly, we increase the amount of data to be processed by each thread for coalescing the global memory access. Moreover, we perform loop unrolling on both algorithm level and warp level to reduce redundant jump and synchronization operations. In our implementation, one thread will load and calculate eight elements of one sub-matrix. We use \#pragma unroll for unrolling.
3.3 Multiplication kernel

The multiplication kernel is responsible for performing the actual matrix multiplication depending on the F-norm results from get-norm kernel. Figure 2b shows the blocking strategy and execution flow for multiplication kernel. Each block has LoNum×LoNum threads and is responsible for calculating one sub-matrix of matrix C. We suppose the block is responsible for C[i,j], and $C[i,j] = \sum A[i,k] \times B[k,j] \times \text{bitmap}[k]$, where k ranges from 0 to (BDIM-1) and bitmap[k] is 1 or 0, indicating whether A[i,k] and B[k,j] satisfies the F-norm threshold, respectively. The bitmap is stored in shared memory. The shared memory is enough for storing the index in int32 for the matrices used in Sect. 4. When $N=32,768$ (largest size of marix in Sect. 4), the bitmap only needs 4 KB in each block. Since each SM has 96 KB on V100, the amount of shared memory is enough. Also, the index in bitmap can be stored in registers. However, the number of index is $N/\text{LoNum}$. If $N=32,768$, one thread block will need 1,024 registers. Since other operations also need registers, the registers may spill to global memory, which leads to the loss of performance. Thus, we store index in shared memory instead of register.

Once the bitmap is calculated, all threads begin to go through the bitmap. If bitmap[k] is 1, the threads load the A[i,k] and B[k,j] into shared memory and then perform the dot product. We adopt double buffering for hiding the memory access latency during the batched sub-matrix product.
However, as shown in Fig. 3a, it is inefficient to implement double buffering naively. This is because the thread needs to go through the bitmap to identify next valid sub-matrices for multiplication. The naive implementation introduces additional instructions (e.g., jump and comparison), which even leads to performance degradation. To address the above problem, we improve the double buffering technique as shown in Fig. 3b, which transforms the access of the valid flags in the bitmap from discontinuous to continuous for better locality. Although such an approach introduces additional calculations, it improves the efficiency of data prefetching with better locality, and thus accelerates the performance of multiplication kernel.

Algorithm 2  

\textit{cuSpAMM}: multiplication kernel

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{Input:} Matrix pointer $\ast A$, $\ast B$, $\ast C$, normmap pointer $\ast A_{\text{normmap}}$, $\ast B_{\text{normmap}}$, parameter $\tau$
\State \textbf{Shared memory:} bitmap, map_offset sAW, sBW, sAR, sBR
\For {k=threadId to N/LoNum by blockDim.x do}
\State norm_mul = $A_{\text{normmap}}[i][k] \times B_{\text{normmap}}[k][j]$
\If {norm_mul $\geq \tau$}
\State bitmap[i] = 1
\Else 
\State bitmap[i] = 0
\EndIf
\EndFor
\For {i=threadId to N/LoNum by blockDim.x do}
\If {bitmap[i] == 1}
\State t = 0
\For {j=0 to i-1 do}
\State t = t + bitmap[j]
\EndFor
\State map_offset[i] = t
\EndIf
\EndFor
\State \textbf{reduce bitmap to get the amount of valid multiplication, save it in validNum}
\If {validNum is not zero, fetch data in first block}
\State \textbf{syncthreads}
\For {i=0 to validNum-1 do}
\State b = map_offset[i]
\State \textbf{syncthreads}
\State Exchange pointer between read and write
\If {the thread is in first half block}
\State if i is less than validNum-1, let next = map_offset[i+1] and perform prefetch
\Else 
\State if the thread is in second half block
\State Calculate two values (c1,c2) of sAWtimesBW
\EndIf
\EndIf
\EndFor
\State write back c1, c2 to matrix C
\end{algorithmic}
\end{algorithm}

Algorithm 2 shows the optimized multiplication kernel. The threads identify the sub-matrices that require actual multiplication and record corresponding indexes in bitmap in parallel (line 5~8). Specifically, threads calculate the F-norm condition using $A_{\text{normmap}}[i][k]$ and $B_{\text{normmap}}[k][j]$ and update bitmap[k] for each k. We use another array map_offset to store the indexes of valid sub-matrices continuously (line 9~14). During each iteration (line 19~27), the first half of warps of the block threads are responsible for matrix multiplication, and the second half is responsible for data prefetching. This strategy facilitates hiding the memory access latency.
by overlapping computation with data access on GPU. Although the GPU provides interleaved multithreading, using double buffer can explicitly instruct the scheduler to make better scheduling as well as saves hardware scheduling overhead. Besides, each thread calculates two elements of $C$, which can be stored in thread registers during dot product.

For input matrices in FP16 precision, we use the tensor core to further accelerate the matrix multiplication. Algorithm 3 shows the pseudo-code of multiplication kernel using tensor core for acceleration (with the same code in FP32 precision omitted). Each block has four warps and each warp is responsible for the sub-matrix $C[warpRow,warpCol]$. The fragment $a_{\text{frag}}$ and $b_{\text{frag}}$ stores sub-matrices of $A$ and $B$, $ab_{\text{frag}}$ is the accumulator of intermediate results. The $ab_{\text{frag}}$ uses FP32 precision for obtaining better accuracy. The $ab_{\text{frag}}$ is initialized to 0. We also apply the double buffering optimization using fragment, which is similar to the implementation in FP32 precision.

### Algorithm 3 Multiplication kernel using tensor core

1: **Input:** Matrix pointer $A, B, C$, normmap pointer $A_{\text{normmap}}, B_{\text{normmap}},$ parameter $\tau$
2: **shared memory:** bitmap
3: **fragment:** $a_{\text{frag}}, b_{\text{frag}}, ab_{\text{frag}}$
4: ......
5: for $i=0$ to $\text{validNum}-1$ do
6: $b = \text{map}_\text{offset}[i]$
7: _syncthreads
8: Exchange pointer between read and write
9: if $i$ is less than $\text{validNum}-1$, perform prefetch and let next = $\text{map}_\text{offset}[i+1]$
10: load_matrix_sync($a_{\text{frag}}, A+warpRowOff+b\times\text{LoNum}, N$)
11: load_matrix_sync($b_{\text{frag}}, B+warpColOff+b\times\text{LoNum}*N, N$)
12: $\text{mna}_\text{sync}(ab_{\text{frag}}, a_{\text{frag}}, b_{\text{frag}}, ab_{\text{frag}})$
13: _end for
14: store_matrix_sync($C+warpRowOff\times T+warpColOff, ab_{\text{frag}}$)

### 3.4 Scaling to multiple GPUs

Modern servers are usually equipped with multiple GPUs (e.g., Nvidia DGX contains up to 16 GPUs). To leverage such performance potential, we extend the blocking optimizations to enable cuSpAMM scale to multiple GPUs on a single server. Note that our multiple GPU optimizations can be further integrated with distributed matrix multiplication optimizations such as CANNON [55] and SUMMA [56]. However, due to the time constraint, we focus on describing the multiple GPU optimizations on a single server, and leave the extension for distributed GPUs in future work.

Algorithm 4 presents the pseudo-code of scaling cuSpAMM to multiple GPUs. We omit the CPU-GPU transfer in Algorithm 4 for simplicity of pseudo-code. We suppose that there are $M$ GPUs indexed from 0 to $M-1$. The calculation task is divided by row, and GPU $i$ is responsible for the rows in the range of $(i\times M/N, (i+1)\times M/N]$ of $C$. The data transfer is divided into $P$ batches and implicitly managed by the use of UM (Unified Memory) technique. Since the memory of our...
GPU device is enough for our matrices in Sect. 4, using UM (with memory direction) leads to little performance loss compared to cudaMalloc due to little page fault. We control the data transfer by ordered page faults. Firstly, several CUDA streams are created with each stream manipulating one GPU device. Then, the CPU transfers the whole matrix $B$ to each GPU in batches, and each GPU obtains the normmap of $B$ at the same time (line 4–6). After that, the CPU sends rows $[i \times N/M, (i+1)\times N/M)$ of matrix $A$ in batches to GPU $i$. In this way, the tasks are independent from each other, and thus no communication is required among GPUs. When each GPU receives the corresponding rows of $A$, it invokes get-norm kernel and waits for the kernel to finish (line 9). After that, it invokes multiplication kernel for calculating the result (line 11). The batching approach is able to hide the data transfer latency as well as reduce the number of active blocks, which in turn mitigates the scheduling overhead.

For the decay matrix, large elements in the matrix are concentrated on the diagonal, and these large values can be evenly distributed to different GPUs through horizontal tiling. Thus, the load balance of each GPU is guaranteed by the characteristics of the decay matrix itself. Ensuring load balancing of SpAMM over arbitrary matrices is a challenging problem, and we will consider it in future work.

### 3.5 Additional optimizations

#### 3.5.1 Improving load balance

The load imbalance could occur in multiplication kernel as shown in Fig. 4a. This is because each block calculates the bitmap dynamically to determine how many operations it needs to perform, which leads to blocks with less load staying idle and wasting resources. To measure the workload of each block, we propose the concept of valid multiplication $v$. For block responsible for calculating sub-matrix $C[i, j]$, its $v$ equals to $\sum_{i=0}^{BDIM} bitmap[i]$. We organize the $v$ values of all blocks into a matrix $V$, where $V[i][j]$ is the $v$ value of the sub-matrix $C[i, j]$. We observe that in matrix $V$,Fig. 4 The illustration of load balance strategy. The size of decay matrix is 1024×1024, the sub-matrix is 32×32 ($LoNum=32$ and $BDIM=32$) and each multiplication block is responsible for 16×16 sub-matrix.
the closer to the diagonal, the greater the \( v \) is, which is determined by the properties of decay matrix.

Based on the above observation, we propose the following load balance strategy. Each block of the multiplication kernel is responsible for the calculation of \( s \) (tunable parameter) sub-matrices with equal stride. For example, as shown in Fig. 4b, one block is responsible for sub-matrices \( C[0, 0], C[0, \text{BDIM}/2], C[\text{BDIM}/2, 0] \) and \( C[\text{BDIM}/2, \text{BDIM}/2] \) with \( s=2 \). The multiplication block can easily adopt the above strategy by adding a loop to change the index of its corresponding sub-matrices in order to achieve better load balance.

**Algorithm 4 Scaling to multiple GPUs**

1: **Input**: matrix A, B, \( \tau \)
2: **Output**: matrix C
3: create CUDA stream for devices
4: for \( i=0 \) to \( P-1 \) do
5: launch get-norm kernel for B
6: end for
7: synchronize at stream level
8: for \( i=0 \) to \( P-1 \) do
9: launch get-norm kernel for A
10: synchronize at stream level
11: launch multiplication kernel
12: end for
13: synchronize at host level
14: output C

### 3.5.2 Searching for customized accuracy

For users using SpAMM algorithm to accelerate non-scientific applications such as deep neural networks (DNNs), adjusting \( \tau \) to control the extent of approximation is not intuitive. For example, the users of DNNs are more concerned about the accuracy of the entire network, other than the numerical accuracy of a single GEMM. In such case, we provide a tuning parameter \( \text{valid ratio} \), formulated as:

\[
\text{valid ratio} = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \frac{V[i,j]}{\text{BDIM}^3}}{\text{BDIM}^3} = \frac{\sum_{i=0}^{N-1} \sum_{j=0}^{N-1} V[i,j]}{\text{BDIM}^3},
\]

to control the actual multiplication of sub-matrices, which ensures that the sub-matrices with large and dense elements participate in calculation with higher priority. This tuning parameter can better adapt to the accuracy requirements of non-scientific applications.

Specifically, after the normmaps of \( A \) and \( B \) are obtained, we use a tuning kernel to calculate the average result (\( \text{ave} \)) of the norm products of all sub-matrices. The kernel then iterates to find the suitable value of norm \( \tau \) that satisfies the \( \text{valid ratio} \) given by the user. Binary search is applied during iterations with search space \([0, k \times \text{ave}]\), where the initial value of the norm is \( \text{ave} \), \( k \) is the expansion coefficient, and the upper bound of the search space is dynamically extended. The initial value of \( k \) is one and will increase to \( k+1 \) whenever the existing upper bound cannot satisfy the search demand. Besides, users can specify the number of iterations and tolerable error of \( \text{valid ratio} \) to balance the time cost and accuracy for searching. Since the searching algorithm is independent of the computation...
kernels, users can develop customized searching algorithms according to their application characteristics.

4 Evaluation

4.1 Experiment setup

Experiment platform—The experiments are conducted on a CPU-GPU server, with two Intel Xeon E5-2680v4 processors and eight NVIDIA Volta V100 GPUs (connected by PCIe). Each GPU contains 32 GB memory. We use nvcc compiler with -O3 option for our implementations.

Matrix multiplication libraries—We compare with GEMM libraries (cuBLAS [32] and cuBLASXT [57]) that treat the decay matrix as dense matrix, and SpGEMM library (cuSPARSE, ACSpGEMM [49], and spECK [48]) that treats the decay matrix as sparse matrix through truncation. Note that, with truncation, the elements smaller than the threshold are treated as zero. The input matrix without truncation is dense matrix. We adopt CSR format for SpGEMM. For cuBLAS, we use cublasSgemm and cublasHgemm API in CUDA 10 and CUDA 11 for matrix multiplication. For cuSPARSE, we use cusparseScsrmm API in CUDA 10 and cusparseSpGEMM API in CUDA 11. It’s worth noting that cublasHgemm in CUDA 10 and GEMM in CUDA 11 use as much tensor core as possible for acceleration. We run ACSpGEMM in CUDA 10 and spECK in CUDA 11, following configurations of their open-source code. Unless otherwise specified, we use the performance in CUDA 10 to calculate the speedup of SpAMM. Using UM in libraries above may result in degradation of performance, so we use cudaMalloc to alloc the memory in device for the implementations of compared libraries.

Evaluation criteria—For matrix multiplication, we use cudaEvent to record the execution time of the program, and the execution time ignores the overhead of input and output transfer (including format conversion) as well as warmup time. We also use Nsight Compute toolkit [58] to measure the performance of kernel. As for accuracy criteria, we use the F-norm of error matrix $E_{n \times n}$ in Eq. 5. For cuSPARSE, the decay matrix is truncated by setting the elements smaller than the threshold $TRUN$ to zeros, which introduces error for SpGEMM. The nz ratio represents the ratio of nonzero elements in the matrix. We use valid ratio to exhibit computation and memory patterns for cuSpAMM. The speedup of cuSpAMM is compared to the library with the same precision.

$$E_{n \times n} = A_{n \times n}B_{n \times n} - SpAMM(A_{n \times n}, B_{n \times n}, \tau)$$

Synthesized matrix dataset—For performance analysis, we synthesize the matrices with algebraical decay where $a_{ij} = b_{ij} = 0.1/|i-j|^{0.1} + 1$, and we control the valid ratio of the matrix indirectly by tuning the norm threshold $\tau$. cuSpAMM is slower than cuBLAS if the valid ratio is bigger than 25% for FP32 and 30% for
Specifically, we use the tuning method in Sect. 3.5.2 to select the threshold and constrain the number of iterations to 20. The errors between actual and expected valid ratios are less than 1%.

### 4.2 Comparison with matrix multiplication libraries

In this section, we use synthesized matrices with algebraically decay listed in Table 1 for comparing with vendor optimized GEMM libraries and SpGEMM libraries including cuBLAS, cuBLAS, and cuSPARSE.

#### 4.2.1 Comparison with GEMM libraries

Comparison with cuBLAS—Table 2 presents the speedup of cuSpAMM on a single GPU compared to cuBLAS. The maximum speedup under each valid ratio is highlighted in red and blue for FP32 and FP16, respectively. Although cuSpAMM
introduces extra memory for normmap, using cuSpAMM for near-sparse matrix multiplication is faster than GEMM. User needs to tradeoff between memory and performance to decide use which algorithm. However, in our implementation, one normmaps only takes 1/1024 of the storage space of the input matrix, which can be ignored if compared to input matrix.

When the valid ratio is 5%, the highest speedup is achieved with 13.4× (FP32) and 16.1× (FP16). It can be seen that cuSpAMM can accelerate matrix multiplication across all matrix sizes when the valid ratio is below a certain threshold. The reason is that when the valid ratio is below the threshold (25% for FP32 and 30% for FP16), our optimizations adopted in cuSpAMM can leverage the property of decay matrix multiplication for better parallelism compared to dense matrix multiplication adopted in cuBLAS.

Comparison with cuBLAS in CUDA 11—Table 3 presents the speedup of cuSpAMM on a single GPU compared to cuBLAS in CUDA 11. We use the CUB-LAS_DEFAULT_MATH, which is the highest-performance mode and uses tensor core whenever possible. The maximum speedup under each valid ratio is highlighted in red and blue for FP32 and FP16, respectively. Compared with CUDA 10, The small matrix multiplication (N=1,024 and 2,048) of cuBLAS in CUDA 11 is obviously accelerated and cuSpAMM is hard to beat the performance of the latter. However, cuSpAMM can still accelerate matrix multiplication when valid

| Valid ratio \ N | Precision | 1024 | 2048 | 4096 | 8192 | 16,384 | 32,768 |
|-----------------|-----------|------|------|------|------|--------|--------|
| ≈30%            | FP32      | 0.8  | 0.9  | 0.8  | 0.8  | 0.8    | 0.8    |
|                 | FP16      | 0.5  | 0.4  | 0.3  | 0.3  | 0.5    | 0.6    |
| ≈25%            | FP32      | 0.9  | 1.0  | 1.0  | 1.0  | 0.9    | 1.0    |
|                 | FP16      | 0.6  | 0.5  | 0.4  | 0.4  | 0.5    | 0.7    |
| ≈20%            | FP32      | 1.1  | 1.2  | 1.3  | 1.2  | 1.2    | 1.2    |
|                 | FP16      | 0.7  | 0.6  | 0.5  | 0.5  | 0.7    | 0.9    |
| ≈15%            | FP32      | 1.2  | 1.6  | 1.6  | 1.6  | 1.6    | 1.6    |
|                 | FP16      | 0.7  | 0.7  | 0.7  | 0.6  | 0.9    | 1.2    |
| ≈10%            | FP32      | 1.5  | 2.2  | 2.2  | 2.2  | 2.5    | 2.5    |
|                 | FP16      | 0.8  | 1.0  | 1.0  | 0.8  | 1.3    | 1.8    |
| ≈5%             | FP32      | 1.9  | 3.3  | 4.3  | 4.3  | 4.4    | 4.5    |
|                 | FP16      | 1.0  | 1.4  | 1.7  | 1.6  | 2.3    | 3.0    |

Table 4 The speedup to cuBLASXT on multiple GPUs when N=32,768

| Valid ratio | 30% | 25% | 20% | 15% | 10% | 5% |
|-------------|-----|-----|-----|-----|-----|----|
| 2 GPUs      | 1.0 | 1.0 | 1.1 | 1.2 | 1.2 | 1.2|
| 4 GPUs      | 1.3 | 1.4 | 1.4 | 1.4 | 1.4 | 1.5|
| 8 GPUs      | 1.3 | 1.3 | 1.4 | 1.4 | 1.3 | 1.4|
Accelerating approximate matrix multiplication for…

The valid ratio is under 20% for FP32 and 15% for FP16. When the valid ratio is 5%, the highest speedup is achieved with 4.5× (FP32) and 3.0× (FP16).

Comparison with cuBLASXT in CUDA 11—We compare cuSpAMM with cuBLASXT on multiple GPUs, using N=32,768. Since cuBLASXT does not support FP16 precision, so we only compare cuSpAMM with cuBLASXT in FP32. In cuBLASXT, the cublasXtgemm API contains the process of copying input data to each device. Thus, we also count the overhead of data copy from host to device when measure the performance of cuSpAMM. Table 4 shows the speedup of cuSpAMM to cuBLASXT. When the valid ratio is 5% and GPU number is 4, the highest speedup is achieved with 1.5×.

4.2.2 Comparison with SpGEMM library

We choose the appropriate settings for $\tau$ so that both implementations reach the same level of error. Since determining the appropriate settings ($\tau$ and TRUN) is time consuming, we choose matrices matrix size of 1,024 and 8,192 for illustration (larger matrix causes out-of-memory error with cuSPARSE on a single GPU).

Comparison with cuSparse—cuSparse does not provide matrix multiplication in FP16 precision in CUDA 10. Thus, we only compare cuSpAMM and cuSparse in FP32 precision. We use CSR format in this experiment. From Table 5, it is clear that at the same error level, cuSpAMM is much faster than cuSparse, and the highest speedup reaches more than 601.0×. With big nz ratio, SpGEMM of cuSparse is even much slower than GEMM of cuBLAS, which is also pointed by prior work [1]. In addition, the speedup of cuSpAMM becomes larger as the nz ratio increases. Especially as the number of GPUs increases, the performance advantage of cuSpAMM becomes even larger (e.g., 3985.1x speedup on eight GPUs). The phenomenon further demonstrates the incapability of cuSparse for handling near-sparse matrices with large nonzero ratio (e.g., more than 24%). Note that, the execution
time of \textit{cuSPARSE} does not include the time of format conversion. Thus, the performance speedup of our \textit{cuSpAMM} will be higher than the results in Table\textsuperscript{5} when compared in real-world application.

**Comparison with \textit{cuSparse} in CUDA 11**— Table\textsuperscript{6} shows the speedup of \textit{cuSpAMM} and \textit{cuBLAS} when compared with \textit{cuSPARSE} in CSR format. We use the same matrices and configurations in Table\textsuperscript{5} in this experiment. In CUDA 11, the \textit{cuSparse} can not perform SpGEMM when\( N=8192\) and \(nz\) \textit{ratio} bigger than 7.5\% because the \textit{cusparseSpGEMM\_workEstimation} API needed by \textit{cusparseSpGEMM\_compute} needs too much memory\[59\]. The phenomenon in CUDA 11 also demonstrates the incapability of \textit{cuSPARSE} for handling near-sparse matrices. The highest speedup of \textit{cuSpAMM} is \(140.6\times/229.6\times/355.0\times/529.8\times\) on 1 GPU/2 GPUs/4 GPUs/8 GPUs for FP32, and \(283.2\times/439.7\times/555.8\times/589.1\times\) on 1 GPU/2 GPUs/4 GPUs/8 GPUs for FP16.

**Comparison with other state-of-the-art SpGEMM libraries**— Table\textsuperscript{7} shows the speedup of \textit{cuSpAMM} compared with ACSpGEMM\[49\] and spECK\[48\] in FP32 precision. We use the same matrices and configurations in Table\textsuperscript{5} in this experiment. Same with \textit{cuSPARSE}, Table\textsuperscript{7} shows that the two libraries also are limited in using SpGEMM for near-sparse matrix. The highest speedup of \textit{cuSpAMM} is

\begin{table}[h]
\centering
\begin{tabular}{lllll|llll}
\hline
No. & Speedup for \(k\) GPUs in FP16\textsuperscript{*} & Speedup for \(k\) GPUs in FP32\textsuperscript{*} & FP16\textsuperscript{†} & FP32\textsuperscript{†} \\
\hline
\hline
1 & 283.2 & 439.7 & 555.8 & 589.1 & 140.6 & 229.6 & 355.0 & 529.8 & 445.3 & 147.5 \\
2 & 73.9 & 108.3 & 122.5 & 129.3 & 41.7 & 64.1 & 86.2 & 106.4 & 70.7 & 22.5 \\
   & 15.1 & 20.5 & 22.9 & 23.6 & 9.0 & 11.9 & 18.7 & 17.4 & 12.11 & 3.5 \\
\hline
2 & 10.7 & 18.1 & 28.1 & 34.1 & 4.7 & 8.7 & 14.7 & 22.4 & 1.47 & 0.22 \\
\hline
\end{tabular}
\caption{The performance comparison with \textit{cuSPARSE} in CSR format in FP16 precision and FP32 precision in CUDA 11.}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{lllll|llll}
\hline
No. & Speedup to ACSpGEMM & Speedup to spECK \\
\hline
\hline
1 & 75.7 & 123.7 & 191.3 & 285.4 & 20.7 & 33.9 & 52.4 & 78.2 \\
2 & 36.7 & 56.4 & 75.9 & 93.6 & 8.4 & 13.0 & 17.4 & 21.5 \\
   & 8.7 & 11.4 & 18.0 & 16.8 & 3.4 & 4.5 & 7.0 & 6.5 \\
\hline
2 & 1.6 & 3.2 & 5.8 & 10.9 & 161.2 & 320.2 & 581.5 & 1089.2 \\
4 & 4.5 & 8.7 & 14.3 & 23.4 & 172.9 & 330.9 & 546.7 & 890.7 \\
   & 3.7 & 6.7 & 11.3 & 17.3 & 0.9 & 1.6 & 2.8 & 4.2 \\
\hline
\end{tabular}
\caption{The speedup of \textit{cuSpAMM} for \(k\) GPUs compared with ACSpGEMM and spECK in FP32 precision.}
\end{table}
Accelerating approximate matrix multiplication for...

compared with ACSpGEMM is 75.7×/ 123.7×/ 191.3×/ 285.4× on 1 GPU/2 GPUs/4 GPUs/ 8 GPUs. The highest speedup of cuSpAMM to spECK is 172.9×/ 330.9×/ 581.5×/ 1089.2× on 1 GPU/2 GPUs/4 GPUs/ 8 GPUs.

Fig. 5 Performance comparison with cuBLAS on matrices with algebraical decay. We change the valid ratio from 30% to 5% with matrix size increasing from 1,024 to 32,768. In addition, we evaluate cuSpAMM scaling from one to eight GPUs.
4.3 Performance analysis

**Scalability of cuSpAMM** —Figure 5 shows the performance comparison when scaling our optimized cuSpAMM to multiple GPUs. We use the performance of cuBLAS in CUDA 10 to calculate the speedup. cuSpAMM achieves better performance speedup when scaling to multiple GPUs across all matrix sizes. For example, when valid ratio = 5%, cuSpAMM achieves the highest speedup of 51.4× with matrix size 4096 in FP16 running on eight GPUs, compared to cuBLAS. FP32 is sometimes faster than FP16 because the latter is unfriendly for memory coarsening. Prior works such as [60] also point that FP16 is not always faster than FP32. For the decay matrix, the smaller the valid ratio, the closer the multiplication task is to the diagonal. When the tasks are completely centered on the diagonal, each block will process only one multiplication and the cuSpAMM achieves perfect load balancing. Thus, the valid ratio of 5% shows better scalability because it has better load balance.

**Ablation experiment** —Table 8 and Table 9 show the performance of implementations with/without optimizations with N=4096 and valid ratio=25% on single GPU. It can be seen that all the proposed optimizations are effective. Specifically, in Get-norm kernel, we first add memory optimizations and unrolling to naive implementation, which accelerates the performance by 1.3× for FP32 and 1.4× for FP16. We then use the tensor core for FP16 precision, which can further speedup the kernel by 3.5×. In the Multiplication kernel, we first adopt memory optimizations (including using shared memory and double buffer) and achieve 2.4× speedup for FP32 and 2.3× speedup for FP16 compared to the naive implementation. Using tensor core for matrix multiplication can further boost the performance by 2.2× for FP16 precision.

| Table 8 | The performance (speedup to naive implementation) of Get-norm kernel with/without optimizations |
|---------|--------------------------------------------------------------------------------------------------|
| Optimizations | FP32 performance | FP16 performance |
| None | 1.0 | 1.0 |
| + memory optimizations, unrolling | 1.3 | 1.4 |
| + tensor core for FP16 | – | 4.7 |

| Table 9 | The performance (speedup to naive implementation) of the Multiplication kernel with/without optimizations |
|---------|--------------------------------------------------------------------------------------------------|
| Optimizations | FP32 performance | FP16 performance |
| None | 1.0 | 1.0 |
| + memory optimizations | 2.4 | 2.3 |
| + tensor core for FP16 | – | 4.4 |
4.4 Case study

We choose two applications widely used in scientific computing and deep neural network to further demonstrate the performance speedup. In ergo application, we use real-world decay matrix and compare cuSpAMM with both cuBLAS and cuSPARSE. We only compare with cuBLAS in VGG13 application since the experiment results in Sect. 4.2 indicate cuBLAS achieves better performance with near-sparse matrices compared to cuSPARSE, which is also pointed by the prior work [1]. The precision of the applications are both FP32 and the environment is CUDA 10. For
ergo application, we select \( \tau \) of different orders of magnitude, resulting in different orders of magnitude of error. For VGG13, we try the values of \( \tau \) for VGG that do not lead to too much accuracy loss.

### 4.4.1 ergo application

**ergo** [4] is an electronic structure computing program widely used in a range of scientific disciplines. We use **ergo** and the water cluster XYZ file [61] to derive the decay matrices directly. The program generates four decay matrices with exponential rate, and the size of each matrix is 13,656×13,656. We use **cuSpAMM** to calculate the power of these matrices, and we use parameter \( \tau \) to control the error \( \left( \frac{||E||_F}{||C||_F} \right) \) of the results. Table 10 and Fig. 6 present the F-norm of the matrices, the error of **cuSpAMM** with different \( \tau \), and the performance speedup. **cuSpAMM** achieves increasing speedup when \( \tau \) becomes larger across all matrices. The speedup of **cuSpAMM** on multi-GPUs comes from the multiplication kernel. When scaling to \( P \) GPUs, the computation complexity of multiplication kernel is \( \text{valid ratio}\times N^3/P \). When \( P=2 \), the performance speedup is \( \text{valid ratio}\times N^3/2 \) theoretically. However, when considering the overhead including organization of data streams and repeated norm calculation of matrix \( B \), the performance speedup is not large enough to offset the overhead. Therefore, the actual performance of **cuSpAMM** on two 2 GPUs is similar to that of a single GPU. As scaling to more GPUs, the performance speedup becomes significant (larger \( P \)), and thus compensates the scaling overhead. Therefore, **cuSpAMM** can achieve higher performance speedup when scaling to 4 and 8 GPUs. For matrices with large F-norm \( (||C||_F \geq 1e^7) \) and \( \tau=1e^{-2} \), the average speedup ranges from 3.0× to 9.8× when scaling to multiple GPUs. In the meanwhile, the error introduced by **cuSpAMM** is much smaller than the data involved in the calculation \( (||E||_F/||C||_F < 8.9e^{-7}) \). For the matrix with small F-norm such as the matrix no.1 and no.2, acceptable error \( (||E||_F/||C||_F < 1.6e^{-5} \text{ when } \tau = 1e^{-4}) \) can be achieved with average speedup of 1.7×/2.9×/3.4×/6.5× (1/2/4/8 GPUs). The F-norm of matrix no.1 is the smallest, which means the valid ratio is also the smallest. As we explained in Sect. 4.3, a lower valid ratio leads to better load balance and scalability. Thus, matrix no.1 shows better scalability than other matrices. In the extreme case with no errors introduced (when \( \tau = 1e^{-10} \)), **cuSpAMM** can still provide average speedup of 1.3×/1.5×/2.3×/4.0× across all matrices.

| Truncation | Nz ratio | \( ||E||_F^* \) | Speedup for \( k \) GPUs |
|-----------|----------|------------------|---------------------|
| \( 1.00E-10 \) | 7.34%    | 0.00              | 2.7 3.2 4.8 8.3       |
| \( 1.00E-08 \) | 5.57%    | 2.00e-5           | 1.8 2.0 3.1 5.4       |
| \( 1.00E-06 \) | 3.85%    | 2.10e-3           | 1.1 1.2 1.9 3.3       |
| \( 1.00E-04 \) | 2.21%    | 0.22              | 0.5 0.6 0.9 1.6       |
| \( 1.00E-02 \) | 0.72%    | 23.78             | 0.1 0.2 0.2 0.4       |

**Table 11** The performance comparison with **cuSPARSE** on matrix no. 1
Table 12 The accuracy and speedup of cuSpAMM on VGG13 application

| Layer   | Valid ratio | Acc loss | \(\tau\) | Speedup for \(k\) GPUs* |
|---------|-------------|----------|--------|--------------------------|
|         |             |          |        | \(k = 1\) | \(k = 2\) | \(k = 4\) |
| conv21  | 97.47\%     | 0        | 0.1    | 2.8 | 5.0 | 8.4 |
|         | 96.84\%     | 0%       | 0.05   | 2.8 | 5.1 | 8.5 |
|         | 85.00\%     | −0.1%    | 2.5    | 3.1 | 5.6 | 9.3 |
|         | 82.90\%     | −0.1%    | 3.0    | 3.2 | 5.7 | 9.4 |
|         | 63.41\%     | −0.9%    | 4.5    | 3.9 | 6.8 | 10.8 |
| conv31  | 97.92\%     | 0        | 2.5    | 2.6 | 4.8 | 7.6 |
|         | 94.21\%     | 0        | 3.5    | 2.6 | 4.8 | 7.8 |
|         | 87.44\%     | −0.1%    | 4.5    | 2.7 | 5.0 | 8.1 |
|         | 74.36\%     | −0.1%    | 5.5    | 3.1 | 5.6 | 9.0 |
|         | 43.38\%     | −1.1%    | 7.5    | 4.8 | 8.1 | 12.4 |

The acc loss measures the difference of prediction accuracy between the cuSpAMM optimized and original models, where negative results indicate accuracy loss.

Table 11 shows the speedup of cuSpAMM compared with cuSPARSE on matrix no. 1. We use the same values of \(\tau\) as the truncation thresholds for cuSPARSE. The matrix no. 1 has the smallest element value in the 4 real-world matrices, which means it has the highest sparsity after truncation, which is beneficial to cuSPARSE. The error of cuSpAMM is smaller than cuSPARSE when \(\tau\) is equal to the truncation threshold. From Table 11, cuSpAMM has better performance if the nonzero is no smaller than 3.85%. The highest speedup is 2.7×/3.1×/4.8×/8.3× for 1 GPU/2 GPUs/4 GPUs/8 GPUs. When truncation threshold is \(1e^{-4}\) and \(1e^{-2}\), the matrix has a high sparsity and cuSPARSE has better performance but the error is much bigger than other configurations.

4.4.2 VGG13 application

We use VGG13 model on dataset MNIST [62]. We use the im2col algorithm to convert the trained weights and input data into matrices. Applying cuSpAMM to implicit GEMM convolution in cutlass requires additional format conversion, which we leave for the future work. We use 80% of the dataset for training and the rest for validation. The size of the input figure is 32×32, and the number of channels is three. The batch size is set to 100 in both training and testing. The prediction accuracy of the original model is 96.6%.

Due to time constraint, we only choose the two largest convolution layers conv21 and conv31 from VGG13 for detailed evaluation with cuSpAMM. The matrices of other layers are dense, thus we omit the results for these layers since they are not the optimization targets for cuSpAMM. After applying im2col operation, the scale of the matrix multiplication is 128×576×25,600 and 256×1,152×6,400 for these two layers, respectively. We apply cuSpAMM to accelerate the two layers and use the same dataset to test the performance and
accuracy. Since the size of matrices is not large enough to occupy more than four GPUs, we only perform performance evaluation on four GPUs at the largest scale for VGG13.

Table 12 shows the evaluation results on both accuracy and performance when using cuSpAMM. In general, cuSpAMM is more effective for improving the performance of matrices multiplication in convolution neural network due to its insensitivity to matrix approximation. The distribution of values in VGG13 matrix is more balance than decay matrix. Thus, cuSpAMM can accelerate matrix from VGG13 with higher valid ratio than decay matrix since the former matrix leads to better load balance of cuSpAMM. For both conv21 and conv31, we can observe significant performance speedup under different settings of $\tau$ with negligible accuracy loss (less than 1.1%). The performance speedup also scales well when increasing from one to four GPUs. Particularly for layer conv31, cuSpAMM achieves 2.6× to 7.8× performance speedup with the prediction accuracy unaffected. With 0.1% accuracy loss, cuSpAMM achieves 2.7× to 9.0× performance speedup. The highest performance speedup 12.4× is achieved when scaling to four GPUs with 1.1% accuracy loss.

5 Conclusion

In this paper, we proposed the first SpAMM algorithm cuSpAMM, tailored for acceleration on multiple GPUs. We re-designed the SpAMM algorithm to parallelize the get-norm kernel and multiplication kernel. In addition, we applied several optimizations to improve the performance and accuracy of cuSpAMM, including blocking strategy, tensor core acceleration, double buffering, load balance and parameter searching. Moreover, we scaled cuSpAMM to multiple GPUs for handling ever-increasing large near-sparse matrices. Our experiment resulted on both synthesized and real-world datasets showed that cuSpAMM achieved significant speedup compared to vendor optimized cuBLAS and cuSPARSE libraries.

Acknowledgements This work was supported by National Key Research and Development Program of China (No. 2020YFB1506703), National Natural Science Foundation of China (No. 62072018) and State Key Laboratory of Software Development Environment (No. SKLSDE-2021ZX-06). Hailong Yang is the corresponding author.

References

1. Gale T, Zaharia M, Young C, Elsen E (2020) Sparse GPU kernels for deep learning. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–14. IEEE
2. Odashima MM, Prado BG, Vernek E (2017) Pedagogical introduction to equilibrium Green’s functions: condensed-matter examples with numerical implementations. Revista Brasileira de Ensino de FÃ­sica, 39: 00
3. Grimme S, Antony J, Ehrlich S, Krieg H (2010) A consistent and accurate ab initio parametrization of density functional dispersion correction (dft-d) for the 94 elements h-pu. J Chem Phys 132(15):154104

4. Rudberg E, Rubensson EH, Salek P, Kruchinina A (2018) Ergo: An open-source program for linear-scaling electronic structure calculations. vol 7, pp 107–111. Elsevier

5. Cao S, Ma L, Xiao W, Zhang C, Liu Y, Zhang L, Nie L, Yang Z (2019) Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 11216–11225

6. Ioannou Y, Robertson D, Cipolla R, Criminisi A, Roots D (2017) Improving CNN efficiency with hierarchical filter groups. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 1231–1240

7. Sharma S, Sharma S, Athaiya A (2017) Activation functions in neural networks. Towards Data Sci 6(12):310–316

8. Vasudevan A, Anderson A, Gregg D (2017) Parallel multi channel convolution using general matrix multiplication. In: 2017 IEEE 28th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp 19–24. IEEE

9. Benzi M, Boito P, Razouk N (2013) Decay properties of spectral projectors with applications to electronic structure. SIAM Rev 55(1):3–64

10. Stephen D, Moss William F, Smith Philip W (1984) Decay rates for inverses of band matrices. Math Comput 43(168):491–499

11. Eijkhout V, Polman B (1988) Decay rates of inverses of banded m-matrices that are near to toeplitz matrices. Linear Algebra Appl 109:247–277

12. Tessler R (2010) Left inverses of matrices with polynomial decay. J Funct Anal 259(11):2793–2813

13. Aune E (2012) Computation and modeling for high dimensional gaussian distributions. Doctoral dissertation, Norwegian University of Science and Technology

14. Ye Q (2013) Error bounds for the lanczos methods for approximating matrix exponentials. SIAM J Numer Anal 51(1):68–87

15. Benzi M, Tuma M (1999) Orderings for factorized sparse approximate inverse preconditioners. SIAM J Sci Comput 21(5):1851–1868

16. Iserles A (2000) How large is the exponential of a banded matrix. New Zealand Math 29(177192):56

17. Simon B (1982) Some jacobi matrices with decaying potential and dense point spectrum. Commun Math Phys 87(2):253–258

18. Michele B, Golub Gene H (1999) Bounds for the entries of matrix functions with applications to preconditioning. BIT Numer Math 39(3):417–438

19. Bowler David R, Tsuyoshi M (2012) Methods in electronic structure calculations. Rep Prog Phys 75(3):036503

20. Cramer M, Eisert J (2006) Correlations, spectral gap, and entanglement in harmonic quantum systems on generic lattices. New J Phys 8(5):71

21. Cramer M, Jens E, Plenio Martin B, Dreissig J (2006) Entanglement-area law for general bosonic harmonic lattice systems. Phys Rev A 73(1):012309

22. Jens E, Cramer M, Plenio Martin B (2010) Area laws for the entanglement entropy—a review. Rev Mod Phys 82(1):277–306

23. Norbert S, Ignacio CJ, Wolf Michael M (2006) Quantum states on harmonic lattices. Commun Math Phys 267(1):65–92

24. Li J, Ranka S, Sahni S (2011) Strassen’s matrix multiplication on GPUs. In: 2011 IEEE 17th international Conference on Parallel and Distributed Systems, pp 157–164. IEEE

25. Coppersmith D, Winograd S (1987) Matrix multiplication via arithmetic progressions. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing, pp 1–6

26. Greathouse JL, Daga M (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, pp 769–780. IEEE

27. Bock N, Challacombe M (2013) An optimized sparse approximate matrix multiply for matrices with decay. SIAM J Sci Comput 35(1):C72–C98

28. Bock N, Challacombe M, Kalé LV (2016) Solvers for O(N) electronic structure in the strong scaling limit. SIAM J Sci Comput 38(1):C1–C21
29. Artemov AG (2019) Sparse approximate matrix multiplication in a fully recursive distributed task-based parallel framework. CoRR, arXiv: abs/1906.08148
30. Ryoo S, Rodrigues CI, Baghsorkhi SS, Stone SS, Kirk DB, Hwu WW (2008) Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. pp 73–82
31. Roberto O, Watson Mark A, Edgar Richard G, Leslie V, Yihan S, Alan A (2010) Accelerating correlated quantum chemistry calculations using graphical processing units and a mixed precision matrix multiplication library. J Chem Theory Comput 6(1):135–144
32. NVIDIA Corporation. cublas. https://developer.nvidia.com/cublas, (2021)
33. NVIDIA Corporation. Nvidia cusparse. https://docs.nvidia.com/cuda cusparse, (2021)
34. Khairy M, Shen Z, Aamodt TM, Rogers TG (2020) Accel-Sim: an extensible simulation framework for validated GPU modeling. In: 2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA), pp 473–486. IEEE
35. NVIDIA Corporation. Nvidia tesla v100 GPU architecture. https://images.nvidia.cn/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, (2021)
36. Bari M, Stoltzfus L, Lin P, Liao C, Emani M, Chapman B (2018) Is data placement optimization still relevant on newer GPUs? Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States)
37. Bischof CH, Lacroute PG (1990) An adaptive blocking strategy for matrix factorizations. In: Helmar B (ed) CONPAR 90 — VAPP IV, pp 210–221, Berlin, Heidelberg. Springer Berlin Heidelberg
38. Lakshminarayana NB, Kim H (2014) Sparse register aware prefetching for graph algorithms on GPUs. In: 2014 IEEE 20th international symposium on high performance computer architecture (HPCA), pp 614–625. IEEE
39. Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: SC’08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp 1–11. IEEE
40. Markidis S, Der Chien SW, Laure E, Peng IB, Vetter JS (2018) Nvidia tensor core programmability, performance & precision. In: 2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW), pp 522–531. IEEE
41. Haidar A, Tomov S, Dongarra J, Higham NJ (2018) Harnessing GPU tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 603–613. IEEE
42. Hatfield S, Chantry M, Düben P, Palmer T (2019) Accelerating high-resolution weather models with deep-learning hardware. In: Proceedings of the Platform for Advanced Scientific Computing conference, pp 1–11
43. Maia Julio DC, dos Anjos Formiga CL, Rocha GB (2020) GPU algorithms for density matrix methods on mopac: linear scaling electronic structure calculations for large molecular systems. J Molecular Modeling 26(11):1–12
44. Gargantini I (1982) An effective way to represent quadtrees. Commun ACM 25(12):905–910
45. Alexander G (2000) Gray and Andrew W. Moore. “N-Body” problems in statistical learning. In NIPS
46. Kalé Laxmikant V, Krishnan Sanjeev (1993) CHARM++: a portable concurrent object oriented system based on c++. In: OOPSLA ’93, pp 91–108
47. Rubensson EH, Rudberg E (2014) Chunks and tasks: a programming model for parallelization of dynamic algorithms. Parallel Comput 40(7):328–343
48. Liu W, Vinter B (2014) An efficient GPU general sparse matrix-matrix multiplication for irregular data. In: 2014 IEEE 28th international parallel and distributed processing symposium, pp 370–381. IEEE
49. Winter M, Mlakar D, Zayer R, Seidel H-P, Steinberger M (2019) Adaptive sparse matrix-matrix multiplication on the GPU. In: Proceedings of the 24th symposium on principles and practice of parallel programming, pp 68–81
50. Huang J, Yu CD, van de Geijn RA (2018) Implementing strassen’s algorithm with CUTLASS on NVIDIA volta GPUs. arXiv:abs/1808.07984
51. Mukunoki D, Ogita T (2020) Performance and energy consumption of accurate and mixed-precision linear algebra kernels on GPUs. J Comput Appl Math 372:112701
52. Navarro CA, Carrasco R, Barrientos RJ, Riquelme JA, Vega R (2020) GPU tensor cores for fast arithmetic reductions. IEEE Trans Parallel Distrib Syst 32(1):72–84
53. Li X, Liang Y, Yan S, Jia L, Li Y (2019) A coordinated tiling and batching framework for efficient GEMM on GPUs. In: Proceedings of the 24th symposium on principles and practice of parallel programming, pp 229–241
54. Kurzak J, Tomov S, Dongarra J (2012) Autotuning gemm kernels for the fermi GPU. IEEE Trans Parallel Distrib Syst 23(11):2045–2057
55. Gupta A, Kumar V (1993) Scalability of parallel algorithms for matrix multiplication. In: 1993 International Conference on Parallel Processing-ICPP’93, vol3, pp 115–123. IEEE
56. Van Den Geijn RA, Watt J (1997) SUMMA: scalable universal matrix multiplication algorithm. Concurrency: Pract Exp 9(4):255–274
57. NVIDIA Corporation. Nvidia cublasxt. https://docs.nvidia.com/cuda/nvblas/index.html, (2021)
58. NVIDIA Corporation. Nvidia nsight. (2021)
59. NVIDIA Corporation. Nvidia cusparse github issue. https://github.com/NVIDIA/CUDALibrarySamples/issues/38, (2021)
60. Ho N-M, Wong W-F (2017) Exploiting half precision arithmetic in nvidia GPUs. In 2017 IEEE High Performance Extreme Computing Conference (HPEC), pp 1–7. IEEE
61. Ergoscf. http://www.ergoscf.org/xyz/h2o.php, (2020)
62. Deng L (2012) The mnist database of handwritten digit images for machine learning research. IEEE Signal Process Mag 29(6):141–142

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Xiaoyan Liu¹,² · Yi Liu² · Hailong Yang¹,² · Ming Dun³ · Bohong Yin² · Zhongzhi Luan² · Depei Qian²

Xiaoyan Liu
liuxiaoyan@buaa.edu.cn

Yi Liu
yi.liu@buaa.edu.cn

Ming Dun
duming0301@buaa.edu.cn

Bohong Yin
yinbohong@buaa.edu.cn

Zhongzhi Luan
zhongzhi.luan@buaa.edu.cn

Depei Qian
deperiq@buaa.edu.cn

¹ State Key Laboratory of Software Development Environment, Beijing 100191, China
² School of Computer Science and Engineering, Beihang University, Beijing 100191, China
³ School of Cyber Science and Technology, Beihang University, Beijing 100191, China