Master's Thesis

# An Efficient Sparse Matrix Storage Format for Sparse Matrix-Vector Multiplication and Sparse Matrix-Transpose-Vector Multiplication on GPUs

Ryohei Izawa

Supervisor Yasushi Inoguchi

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

Conferment December, 2023

**Abstract**

The utilization of sparse matrix storage formats is widespread across various fields, including scientific computing, machine learning, and statistics. Within these domains, there is a need to perform Sparse Matrix-Vector Multiplication (SpMV) and Sparse Matrix-Transpose-Vector Multiplication (SpMVT) iteratively within a single application. However, executing SpMV and SpMVT on GPUs using existing sparse matrix storage formats presents challenges related to memory usage, load balancing, and memory access efficiency.

In our research, we propose a novel sparse matrix storage format named GCSB, specifically designed for efficient SpMV and SpMVT operations on GPUs, leveraging high memory compression. Initially, we adapt CSB, a sparse matrix storage format compatible with CPU-based SpMV and SpMVT, for GPU use in a straightforward manner, referred to as CSB-baseline. Subsequently, we extend the CSB-baseline to propose GCSB, which enables faster execution of SpMV and SpMVT than CSR through load balancing and efficient utilization of L1 cache, while maintaining theoretical memory usage equivalent to that of CSR.

Through experiments, we demonstrate that GCSB achieves SpMV and SpMVT with theoretical memory usage equivalent to CSR while outperforming CSR in terms of execution speed on several matrices from the University of Florida Sparse Matrix Collection. GCSB achieves up to 1.47× speedup on TITAN RTX and 2.75× on A100. Additionally, we show that GCSB reduces L1 cache miss counts compared to CSB-baseline. Furthermore, we qualitatively evaluate that GCSB demonstrates its superior performance under conditions where non-zero elements are broadly distributed throughout the matrix, the matrix size is considerable, and the proportion of non-zero elements within the matrix is relatively high.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem setting

Matrix and vector multiplication is a fundamental operation widely employed in various fields, including data analysis, graph analytics, machine learning, and scientific computing. Among these matrix and vector multiplications, matrices with a substantial number of zero elements are referred to as sparse matrices. The multiplication of a sparse matrix by a dense vector is termed Sparse Matrix-Vector Multiplication (SpMV). Given a sparse matrix $A$ of dimensions $m \times n$ and a dense vector $x$ of size $n$, SpMV computes a dense vector $y$ of size $m$ as $y = Ax$.

When performing SpMV on a large sparse matrix $A$ of dimensions $m \times n$ with $nnz$ non-zero elements, the memory usage required for reading matrix $A$ can become a limiting factor in the performance of SpMV. Consequently, many algorithms store the sparse matrix $A$ in a compressed format before computing SpMV. In order to minimize the memory usage of SpMV, several compressed formats have been developed, including the Coordinate (COO), Compressed Sparse Rows (CSR), and ELLPACK (ELL)[21]. Utilizing these compression techniques, numerous approaches have been investigated to efficiently and rapidly compute SpMV while optimizing memory usage [15, 24] on CPUs.

Graphic Processing Units (GPUs) are extensively employed in a wide range of high-performance applications, providing remarkably elevated throughput. GPUs are engineered with a substantial quantity of compact processing units, facilitating a notable level of parallelism. Additionally, GPUs incorporate an extensive memory hierarchy and possess a high memory bandwidth. Leveraging these distinctive attributes of GPUs, recent works have been conducted to optimize SpMV on GPUs [2, 9, 10, 13, 25].

There are applications that involve not only SpMV but also Sparse Matrix-Transpose-Vector Multiplication (SpMVT), which computes $y = A^T x$, where $A^T$ denotes the transpose of a sparse matrix $A$. Algorithms such as the Bi-Conjugate Gradient Algorithm (BCG) and Quasi-Minimal Residual Algorithm [21] iterate over SpMV and SpMVT computations. Deep Neural Networks (DNN), which have been achieving results in various fields such as image processing and natural language processing, are also dealing with sparse matrices to improve processing speed or accuracy [11, 8, 17]. In DNN, both sparse matrices and their transposes are handled within an application [9, 12, 26]. Compression techniques such as CSR and ELL are oriented towards compression in the row direction, thus not inherently suited for transposed matrices. While using the Compressed Sparse Columns (CSC), which compresses in the column direction, is an option, it results in storing the matrix in two different formats, demanding more memory usage and thereby becoming less efficient.

**Algorithm 1** Bi-Conjugate Gradient Algorithm [21]

Compute $r_0 := b - Ax_0$. Choose $r_0^*$ such that $(r_0, r_0^*) \neq 0$.
Set $p_0 := r_0$, $p_0^* := r_0^*$;
**for** $j = 0, 1, \ldots$ until convergence **do**
    $\alpha_j := (r_j, r_j^*)/(Ap_j, p_j^*)$;
    $x_{j+1} := x_j + \alpha_j p_j$;
    $r_{j+1} := r_j - \alpha_j A p_j$; // SpMV
    $r_j^* := r_j^* - \alpha_j A^T p_j^*$; // SpMVT
    $\beta_j := (r_{j+1}, r_{j+1}^*)/(r_j, r_j^*)$;
    $p_{j+1} := r_{j+1} + \beta_j p_j$;
    $p_{j+1}^* := r_{j+1}^* + \beta_j p_j^*$;
**end for**

To address these challenges, Compressed Sparse Blocks (CSB) [4] is proposed as a format suitable for such operations. However, CSB was originally designed for CPUs, and directly porting it to GPUs presents challenges related to achieving coalesced memory access and load balancing.

An adapted GPU version of CSB called eCSB [23] has been proposed. However, eCSB suffers from the following limitations:

- The theoretical memory usage required by eCSB is either equal to or larger than that of the COO, failing to achieve the same level of memory compression as the CSR.
- The criteria used to select containment methods in eCSB are determined heuristically, leading to suboptimal load balancing.

## 1.2 Contributions

In this study, we present our contributions through the proposal of a sparse matrix format named Grouped Compressed Sparse Blocks (GCSB), designed specifically for GPUs. GCSB is compatible with SpMV and SpMVT operations in an application, offering exceptional load balancing and efficient L1 cache usage, all while maintaining a theoretical memory usage equivalent to that of the CSR. Our main contributions are as follows:

- We redefine CSB for GPU, introducing a single and straightforward format that achieves theoretical memory usage equivalence to CSR.
- We introduce a novel load balancing technique named *block swizzle load balancing*, which assigns blocks to Streaming Multiprocessors (SMs) based on the distribution of non-zero elements.
- We present a method named *grouped block element reordering*, which rearranges the non-zero elements within a group of blocks assigned to a SMs, thereby achieving efficient L1 cache usage.

## 1.3 Outline

The organization of this thesis is as follows: Chapter 2 provides an overview by examining related works to establish the foundational knowledge required for this paper, including an exploration of GPUs and sparse matrix storage formats. In Chapter 3, we provide a detailed explanation of our proposed method, which handles both SpMV and SpMVT execution on GPUs. Chapter 4 presents the setup and results of our experiments, and Chapter 5 offers concluding remarks.

# Chapter 2

# Background

## 2.1 Introduction

In this chapter, we begin by introducing fundamental sparse matrix storage formats that are designed to support SpMV or SpMVT operations. We provide an explanation of these formats, namely COO, CSR, ELL, and CSB, focusing on the arrays or vectors used for storage and the associated memory usage required for storing matrices.

Additionally, we provide insights into the fundamental architecture of GPUs, including their memory structures and considerations when performing computations on these devices.

Subsequently, we discuss the challenges associated with executing both SpMV and SpMVT within the same program on a GPU. This discussion encompasses the difficulties of implementing these operations using existing sparse matrix storage formats such as COO, CSR, and ELL. We also address the complexities involved in adapting CPU-based CSB implementations for GPU execution and highlight remaining issues in eCSB, a proposed method for executing SpMV and SpMVT within the same GPU program.

Following that, we describe the kernels for SpMV and SpMVT on GPUs using each of the sparse matrix storage formats, including COO, CSR, and ELL.

Finally, we present existing research on optimization techniques for SpMV using existing sparse matrix storage formats executed on GPUs.

## 2.2 Sparse matrix storage formats

A sparse matrix refers to a matrix of size $m \times n$ that contains a relatively small number of non-zero elements ($nnz$). Various methods of representing

sparse matrices exist, each with distinct storage requirements, computational attributes, and techniques for accessing and manipulating matrix elements.

### 2.2.1 COO

COO is a straightforward sparse matrix storage format. Figure 2.1 illustrates the representation of COO for a $4 \times 4$ matrix example. COO consists of arrays: *values* storing the values of non-zero elements, *row_indices* storing row indices of non-zero elements within the matrix, and *column_indices* storing column indices of non-zero elements.

| | | | | | |
|---|---|---|---|---|---|
| *a* | *b* | *0* | *0* | *c* | *d* |
| *e* | *0* | *f* | *0* | *g* | *0* |
| *0* | *h* | *0* | *0* | *0* | *0* |
| *0* | *i* | *0* | *j* | *0* | *0* |
| *0* | *0* | *0* | *0* | *0* | *0* |
| *0* | *k* | *0* | *0* | *0* | *l* |

*a* ~ *z* : non-zero element

$$values = [a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l]$$
$$row\_indices = [0\ 0\ 0\ 0\ 1\ 1\ 1\ 2\ 3\ 3\ 5\ 5]$$
$$column\_indices = [0\ 1\ 4\ 5\ 0\ 2\ 4\ 1\ 1\ 3\ 1\ 5]$$

Figure 2.1: COO format.

*values*, *row_indices* and *column_indices* each consist of *nnz* elements. Therefore, required memory usage for COO is as follows:

$$\begin{aligned} MemoryUsage_{COO} = {} & nnz \cdot sizeof(value) \\ & + 2nnz \cdot sizeof(index) \end{aligned} \tag{2.1}$$

### 2.2.2 CSR

CSR is a widely employed sparse matrix representation across various applications. Figure 2.2 illustrates the CSR representation using a $4 \times 4$ matrix example. CSR consists of arrays such as *values* for storing the values of non-zero elements and *columns_indices* for holding column indices of non-zero

elements within the matrix. Moreover, CSR is characterized by the presence of the $row\_ptr$ array. The value of $row\_ptr[i]$ represents the offset, indicating the position of the first non-zero element in the $i$-th row among all non-zero elements.

| | | | | | |
|---|---|---|---|---|---|
| $a$ | $b$ | 0 | 0 | $c$ | $d$ |
| $e$ | 0 | $f$ | 0 | $g$ | 0 |
| 0 | $h$ | 0 | 0 | 0 | 0 |
| 0 | $i$ | 0 | $j$ | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | $k$ | 0 | 0 | 0 | $l$ |

$a \sim z$ : non-zero element

$$
\begin{aligned}
values &= [a\ b\ c\ d\ e\ f\ g\ h\ i\ j\ k\ l] \\
column\_indices &= [0\ 1\ 4\ 5\ 0\ 2\ 4\ 1\ 1\ 3\ 1\ 5] \\
row\_ptr &= [0\ 4\ 7\ 8\ 10\ 10\ 12]
\end{aligned}
$$

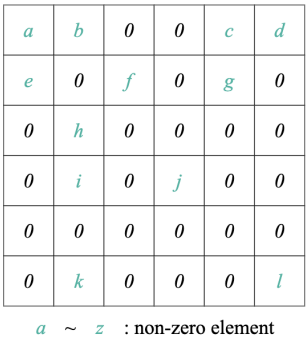
Figure 2.2: CSR format.

Considering the size of $row\_ptr$ is $m+1$, the memory usage required for CSR is as follows:

$$
\begin{aligned}
MemoryUsage_{CSR} = & \ nnz \cdot sizeof(value) \\
& + nnz \cdot sizeof(index) \\
& + (m+1) \cdot sizeof(index)
\end{aligned}
\tag{2.2}
$$

### 2.2.3 ELL

The ELL stores only the non-zero elements of each row in a data array, along with their corresponding column indices. Figure 2.3 provides an illustrative example of the ELL representation. The length of each row in the data array is determined by the maximum number of non-zero elements in a row, denoted as $k$. Rows with fewer non-zero elements than $k$ are zero-padded accordingly.




$$
\begin{aligned}
values = [&a\ b\ c\ d\ e\ f\ g\ *\ h\ *\ *\ * \\
&i\ j\ *\ *\ *\ *\ *\ *\ k\ l\ *\ *] \\
column\_indices = [&0\ 1\ 4\ 5\ 0\ 2\ 4\ *\ 1\ *\ *\ * \\
&1\ 3\ *\ *\ *\ *\ *\ *\ 1\ 5\ *\ *]
\end{aligned}
$$

Figure 2.3: ELL format.

The memory usage required for ELL is as follows:

$$
\begin{aligned}
MemoryUsage_{ELL} = &mk \cdot sizeof(value) \\
&+ mk \cdot sizeof(index)
\end{aligned}
\tag{2.3}
$$

Storing matrices in the ELL becomes efficient when there is low variance in the number of non-zero elements per row, as this leads to reduced zero-padding.

### 2.2.4 CSB

The previously mentioned COO, CSR, and ELL are typically compressed using row-major order. Consequently, when performing SpMVT computations, discontinuous accesses arise, leading to inefficiencies. To address this issue, CSB [4] has been proposed as a compression method for handling both SpMV and SpMVT within the same application.

CSB divides matrix $A$ into blocks, denoted as $A_{ij}$, with a block size of $\beta \times \beta$. A matrix $A$ of size $m \times n$ is represented as follows.

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n\_block\_cols} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n\_block\_cols} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n\_block\_rows,0} & A_{n\_block\_rows,1} & \cdots & A_{n\_block\_rows,n\_block\_cols} \end{bmatrix} \tag{2.4}$$

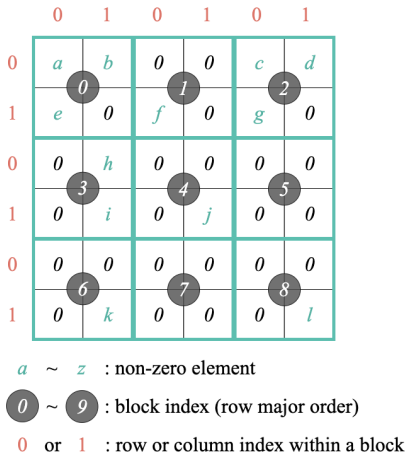
where $A_{ij}$ represents the partitioned blocks of $A$, $n\_block\_rows$ denotes the number of blocks in the row direction, which is equal to $(m + \beta - 1)/\beta$, and $n\_block\_cols$ represents the number of blocks in the column direction, which is equal to $(n + \beta - 1)/\beta$.

A Block $A_{ij}$ can be represented as follows:

$$A_{ij} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,\beta-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,\beta-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{\beta-1,0} & a_{\beta-1,1} & \cdots & a_{\beta-1,\beta-1} \end{bmatrix} \tag{2.5}$$

where $a$ denotes the values of each non-zero element within the block $A_{ij}$.

Figure 2.4 shows the representation of CSB with a matrix example. CSB includes an array *values* to store the non-zero elements, an array *comb_row_col_indices* to store the row and column indices of non-zero elements within block $A_{ij}$, and an array $block\_offset$ to hold the offset indicating the position of the first non-zero element within block $A_{ij}$ relative to all non-zero elements. The value of $comb\_row\_col\_indices[i \times m + j]$ encodes the row index in the upper bits and the column index in the lower bits for the $(i \times m + j)$-th block.




$$values = [a\ b\ e\ f\ c\ d\ g\ h\ i\ j\ k\ l]$$

$$comb\_row\_col\_indices = [00\ 01\ 10\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 11\ 11]$$

$$block\_offset = [0\ 3\ 4\ 7\ 9\ 10\ 10\ 11\ 11\ 12]$$

Figure 2.4: CSB format.

The block size $\beta$ is determined, for instance, as $\beta = \min(\sqrt{n}, \sqrt{m})$. The number of blocks is $n\_blocks = (m+\beta-1)/\beta \times (n+\beta-1)/\beta$. The order of non-zero elements within a block follows either Z-ordering or Morton order [19]. As shown in Figure 2.5, in Morton order, the initial elements located in the upper-left quadrant are stored first, followed by the elements in the upper-right, lower-left, and finally lower-right quadrants. This recursive pattern is consistently utilized for the layout.

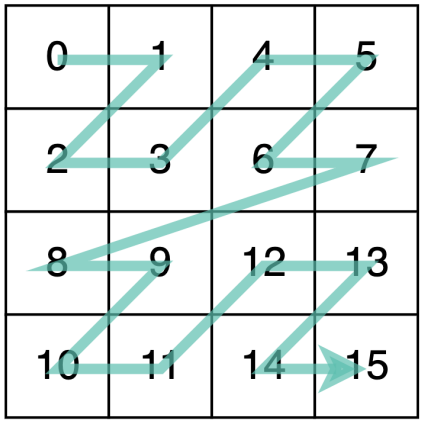



Figure 2.5: Morton order

The required memory usage for CSB is as follows:

$$\begin{aligned} MemoryUsage_{CSB} &= nnz \cdot sizeof(value) \\ &+ nnz \cdot sizeof(index) \\ &+ (n\_blocks + 1) \cdot sizeof(index) \end{aligned} \tag{2.6}$$

When $n\_blocks = m$, $MemoryUsage_{CSB}$ is equivalent to $MemoryUsage_{CSR}$.

## 2.3 GPUs

GPUs are originally designed to efficiently handle graphics processing tasks within computer hardware. They excel at tasks involving visual data, like rendering 2D and 3D images, processing videos, and delivering real-time visuals in games.

In the past few years, the strong ability of GPUs to process tasks in parallel has been used for a wider range of computing needs. They've been put to use in many different areas like scientific calculations, machine learning, deep learning, virtual simulations, and even mining cryptocurrencies.

In our study, we employ NVIDIA GPUs and CUDA programming model, which is the integrated development environment of NVIDIA GPUs.

### 2.3.1 GPU architecture

GPUs are equipped with multiple Streaming Multiprocessors (SMs), with each SM containing numerous CUDA cores. These CUDA cores manage

individual tasks and have the capability to execute multiple threads concurrently, facilitating parallel processing. Threads serve as the smallest units of execution within a program and are organized into groups known as warps. Typically, a warp consists of 32 threads running the same program simultaneously, executing the same instructions at the same time. Warps operate on CUDA cores within an SM and adhere to the Single Instruction Multiple Data (SIMD) architecture.

Figure 2.6 provides an overview of data transfer between the GPU and CPU, as well as memory allocation. GPUs feature global memory, characterized by its high bandwidth but also significant latency. During computations, data is initially read from CPU memory into global memory, processed, and then written back to global memory before being transferred to CPU memory. Additionally, GPUs are equipped with texture memory and constant memory. The GPU memory incorporates two cache levels: the L2 cache, which is shared among SMs, and caches data during read and write operations to global memory, and each SM has its dedicated L1 cache, shared among the threads within the SM.

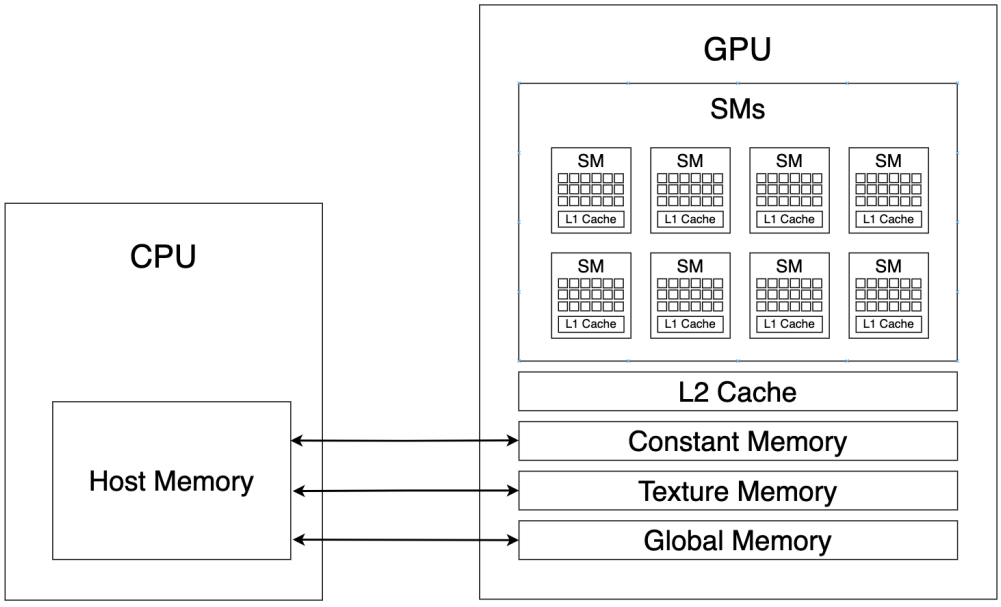
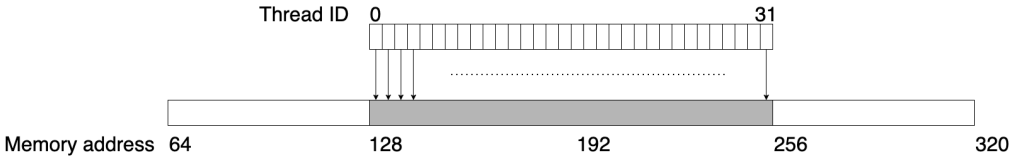



Figure 2.6: GPU architecture, data transfer and memory allocation.

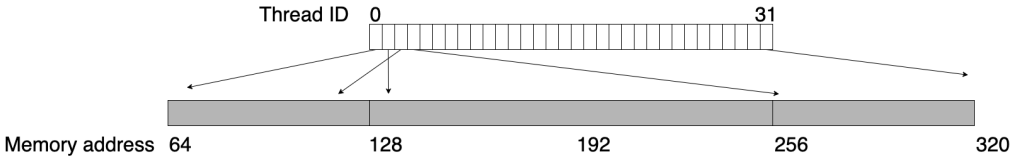
To achieve high computational performance on GPUs, various optimizations are required. In this context, we discuss the reduction of memory usage, the achievement of high cache hit rates, the avoidance of warp divergence, the maintenance of high occupancy, and load balancing.

When memory usage becomes problematic, the exchange of data between CPU memory and GPU global memory becomes imperative, resulting in frequent data transfers. Reducing memory usage is essential for optimization and acceleration.

Efficient utilization of the memory hierarchy, including L1 and L2 caches, and ensuring aligned memory access and coalesced memory access are necessary. Aligned memory access is achieved when the initial address of a memory transaction is a multiple of the cache line granularity, which is 128 bytes for L1 cache and 32 bytes for L2 cache. Coalesced memory access is achieved when all 32 threads within a warp access contiguous memory chunks. Figure 2.7 illustrates cases within the L1 cache where aligned and coalesced memory access is achieved and cases where it is not. In case (a), the requested addresses by the threads within the warp fit within a 128-byte cache line, achieving both aligned and coalesced memory access, resulting in a single transaction. In case (b), the threads within the warp request addresses randomly scattered in the global memory, spanning multiple cache lines, potentially leading to up to 32 transactions in the worst case.




(a) Align and coalesced memory access.




(b) Miss-align and un-coalesce memory access.

Figure 2.7: GPU L1 cache access [5].

Warp divergence happens when conditional branches or data accesses differ. While threads following one branch path remain active, threads not taking that path become inactive, leading to a loss of parallelism and resulting in decreased performance.

Occupancy refers to the ratio of active warps to the maximum number of warps per SM. Increasing occupancy allows each SM to maintain a state in which as many warps as possible are consistently executed, reducing idle periods and minimizing wasteful consumption of computational resources.

Additionally, load imbalance can occur not only between different SMs but also among warps or threads. When tasks assigned to a particular SM are smaller than those in other SMs, that SM might become idle in comparison to active SMs. Similarly, when tasks allocated to warps or threads within an SM are limited, certain warps or threads might remain inactive and in a waiting state.

GPUs exhibit significant computational performance, but optimizing memory access and thread management is pivotal. When utilizing the CUDA programming model for parallel computing, paying attention to these factors is essential.

### 2.3.2 CUDA programming model

CUDA programming is a framework developed by NVIDIA that enables developers to harness the immense computational power of GPUs for parallel computing tasks. It provides a programming model and API for writing high-performance code that can execute efficiently on GPUs.

In CUDA programming, the concepts of grid, block, and thread are employed to control the computation on the GPU. Figure 2.8 illustrates the configuration of parallel threads. The grid represents the entire computational domain and contains multiple blocks. A block represents a smaller region within the grid and is executed on a single SM within the GPU. Each block comprises multiple threads, with each thread executing its own task. Grid and Block can be specified with dimensions of up to three dimensions each.

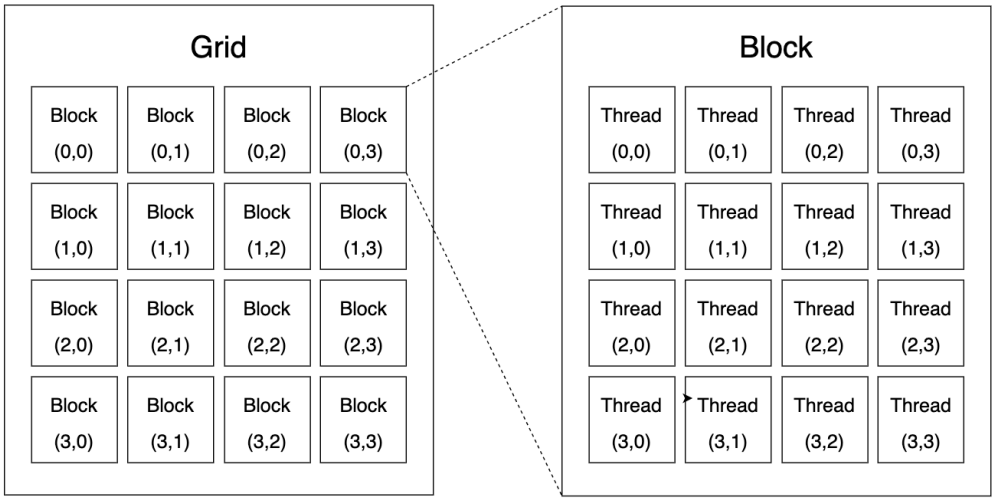



Figure 2.8: Grid, Blocks and Threads.

The central computational unit in CUDA programming is the kernel. A kernel is a function written in C/C++ with specific CUDA syntax that is executed by multiple threads in parallel. Kernels are launched from the host (CPU), and executed on the device (GPU). As shown in Figure 2.9, the kernel labeled as $SomeKernel$ with the $\_\_global\_\_$ attribute is invoked from the host side. $<<< GridDim, BlockDim >>>$ notation specifies the dimensions of the grid and block dimensions, determining the hierarchical structure for thread execution.

```
1 __global__ void SomeKernel(float *x, float *y)
2 {
3     ...
4 }
5
6 int main() {
7     ...
8     SomeKernel<<<GridDim, BlockDim>>>(x, y);
9 }
```

Figure 2.9: Sample CUDA Kernel

## 2.4 The challenges of executing SpMV and SpMVT within an application on GPUs

In this section, we discuss the challenges related to executing SpMV and SpMVT on GPUs. There are primarily three approaches for performing SpMV and SpMVT using the existing sparse matrix storage formats. The first approach involves utilizing sparse matrix storage formats such as COO, CSR, and ELL. The second approach involves implementing the CSB format specifically designed for GPUs. The third approach extends the CSB format to the eCSB format [23].

### 2.4.1 SpMV and SpMVT with existing sparse matrix storage formats

Conventional sparse matrix storage formats, such as COO, CSR, and ELL, as shown in Figures 2.1, 2.2, and 2.3, store values along the row directions. Consequently, achieving efficient memory access in both row and column directions is challenging. Although the Compressed Sparse Columns (CSC) storage format exists for sparse matrices, where values are stored along the column direction of CSR, using two storage formats results in redundant memory usage.

In the following section, we present CUDA Kernels for SpMV and SpMVT utilizing COO, CSR, and ELL. In the case of SpMVT, computations are conducted by accessing data in the row direction while disregarding memory access in the column direction. However, due to frequent memory accesses to the output vector $y$ for each row, efficient cache utilization cannot be guaranteed.

### 2.4.2 SpMV and SpMVT with CSB

CSB [4] is a sparse matrix storage format originally designed for performing SpMV and SpMVT on multi-core CPUs. CSB maintains a memory usage equivalent to CSR while enabling efficient computation of SpMV and SpMVT. However, directly applying this this methodology directly on GPUs presents challenges.

CSB includes a process involving recursive block partitioning based on the number of non-zero elements within blocks. GPUs are built on the principle of SIMD execution, where all threads must execute the same program. Consequently, adapting processing based on the number of non-zero elements within blocks introduces branching and results in warp divergence, which is

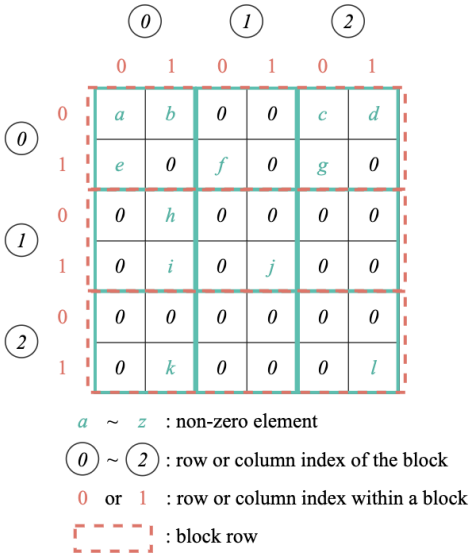
problematic in SIMD execution. Replicating the intricate recursive process becomes challenging.

When porting CSB directly to GPUs, a naive implementation arises where each thread handles the multiplication of non-zero elements within a single block. In this scenario, each thread accesses non-adjacent addresses, potentially spanning multiple cache lines. In such cases, achieving coalesced memory access becomes difficult, hindering efficient memory transactions.

Furthermore, since CSB operates on a block level, blocks with a low number of non-zero elements lead to inefficient utilization of the allocated computational resources. Thus, careful attention is required for thread scheduling to achieve load balancing.

### 2.4.3 SpMV and SpMVT with eCSB

There is a technique known as Expanded Compressed Sparse Blocks (eCSB) [23], which redefines CSB for GPU implementation. The storage format of eCSB is illustrated in Figure 2.10. In eCSB, in addition to the array of non-zero elements *values* and the array of row indices and column indices of non-zero elements within the blocks referred to as *comb_row_col_indices*, it also includes a block index array for each non-zero element labeled as *block_indices* and the array of the pointer to the first non-zero element in each block row known as *block_row_ptr*.




$$values = [a\ b\ e\ f\ c\ d\ g\ h\ i\ j\ k\ l]$$

$$comb\_row\_col\_indices = [00\ 01\ 10\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 11\ 11]$$

$$block\_indices = [00\ 00\ 00\ 01\ 02\ 02\ 02\ 10\ 10\ 11\ 20\ 22]$$

$$block\_row\_ptr = [0\ 7\ 10\ 12]$$

Figure 2.10: eCSB format.

The memory usage of eCSB is given by $nnz \cdot sizeof(value) + 2nnz \cdot sizeof(index) + n\_block\_rows \cdot sizeof(index)$, where $n\_block\_rows$ represents the number of block rows. Depending on the distribution of non-zero elements in the matrix, eCSB dynamically switches between sparse matrix formats such as ELL, COO, or a combination of the two (HYBRID).

When conducting SpMV and SpMVT using eCSB, the initial step involves utilizing the Bell and Garland procedure [3] to determine two parameters: the number of non-zero elements within each block row and the remaining number of block rows after thinning out those block rows with fewer non-zero elements. Based on these parameters, it is heuristically decided whether to store data in ELL, COO, or a HYBRID of ELL and COO. The fundamental concept underlying this approach is to use COO when non-zero elements are distributed in a highly random manner, and to resort to HYBRID or ELL otherwise.

However, eCSB encounters challenges in the following aspects. Initially, in both ELL and COO formats, the memory usage required for SpMV and

SpMVT is equal to or greater than that of COO, which results in an insufficient memory compression effect. Additionally, the choice of storage methods is heuristic, and the distribution of loads among SMs or warps/threads is not optimized.

## 2.5 CUDA kernels for SpMV and SpMVT

In this section, we outline the implementation of SpMV and SpMVT on GPUs through CUDA kernels for each sparse matrix storage format. It is important to note that eCSB, a CUDA-based CSB, varies from the CPU-based CSB discussed earlier. In Chapter 3, we delve into the details of the baseline CSB implementation on GPUs, which is the focus of our research. In this section, we provide CUDA kernels for COO, CSR, and ELL.

### 2.5.1 COO kernel

The CUDA kernel for COO involves a straightforward computation where each thread handles a single non-zero element of the matrix. Figure 2.11 illustrates the CUDA kernel for COO-based SpMV. Each thread is responsible for multiplying a single non-zero element. In line 11, we ensure that the thread id does not exceed the number of non-zero elements. In line 12, we perform the multiplication of non-zero elements and store the values in vector $y$. When storing the results in the output vector $y$, `atomicAdd` is employed to accumulate the individual computations. `atomicAdd` is an instruction designed to prevent race conditions when multiple threads concurrently access a specific memory location to perform addition operations. This ensures that simultaneous writes to the same memory location are avoided, thereby enabling accurate results. Figure 2.12 shows the CUDA kernel for SpMVT. The distinction from the SpMV CUDA kernel lies in the swapping of $columns[i]$ and $rows[i]$ when performing `atomicAdd`.

```
__global__ void KernelSpMVCOO(const int nnz,
                              const float* values,
                              const int* rows,
                              const int* columns,
                              const float* x,
                              float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    if (i < nnz) {
      atomicAdd(&y[rows[i]], values[i] * x[columns[i]]);
    }
}
```

Figure 2.11: SpMV CUDA kernel for COO.

```
__global__ void KernelSpMVTCOO(const int nnz,
                              const float* values,
                              const int* rows,
                              const int* columns,
                              const float* x,
                              float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    if (i < nnz) {
      atomicAdd(&y[columns[i]], values[i] * x[rows[i]]);
    }
}
```

Figure 2.12: SpMVT CUDA kernel for COO.

### 2.5.2 CSR kernel

In the CSR CUDA Kernel, a notable characteristic is that each thread is assigned to a single row. Figure 2.13 represents the CUDA Kernel for CSR-based SpMV. Since each thread handles non-zero elements in a row, we ensure that the thread ID does not exceed the number of non-zero elements in line 12. In lines 13 and 14, the $i$-th thread extracts the starting and ending indices of non-zero elements for row $i$ from *row_ptr*. In lines 15 and 16, we perform value multiplication for each non-zero element and sum up those results in the temporary variable *temp_sum*. We store the result in the output vector

$y[i]$. As a result, each thread accesses a specific address in the output vector $y$.

In contrast, in the case of SpMVT, each thread is responsible for a single row of the original matrix, leading to access of multiple addresses in the output vector $y$. Figure 2.14 illustrates the CUDA Kernel for SpMVT. The use of 'atomicAdd`atomicAdd` is necessary to circumvent memory conflicts in line 15. In cases where multiple threads access the same memory in `atomicAdd`, each thread needs to wait for the others to complete their operations. This can lead to reduced parallelism and performance bottlenecks, which contradicts the expected high parallelism capabilities of GPUs.

```
1  __global__ void KernelSpMVCSR(const int n_rows,
2                                const float* values,
3                                const int* row_ptr,
4                                const int* columns,
5                                const float* x,
6                                float* y) {
7      int idx = blockDim.x * blockIdx.x + threadIdx.x;
8      int idy = blockDim.y * blockIdx.y + threadIdx.y;
9      int offset = gridDim.x * blockDim.x;
10     int i = idx + idy * offset;
11     float temp_sum = 0.0;
12     if (i < n_rows) {
13         int row_start = row_ptr[i];
14         int row_end = row_ptr[i+1];
15         for (int j = row_start; j < row_end; j++) {
16             temp_sum += values[j] * x[columns[j]];
17         }
18         y[i] = temp_sum;
19     }
20 }
```

Figure 2.13: SpMV CUDA kernel for CSR.

```
1  __global__ void KernelSpMVTCSR(const int n_rows,
2                                 const float* values,
3                                 const int* row_ptr,
4                                 const int* columns,
5                                 const float* x,
6                                 float* y) {
7      int idx = blockDim.x * blockIdx.x + threadIdx.x;
8      int idy = blockDim.y * blockIdx.y + threadIdx.y;
9      int offset = gridDim.x * blockDim.x;
10     int i = idx + idy * offset;
11     if (i < n_rows) {
12         int row_start = row_ptr[i];
13         int row_end = row_ptr[i+1];
14         for (int j = row_start; j < row_end; j++) {
15             atomicAdd(&y[columns[j]], values[j] * x[i]);
16         }
17     }
18 }
```

Figure 2.14: SpMVT CUDA kernel for CSR.

### 2.5.3 ELL kernel

Similar to CSR, the CUDA Kernel for ELL also assigns each thread to handle one row. Figure 2.15 shows the CUDA Kernel for ELL-based SpMV. In line 13, we calculate the index of the first non-zero element in the row. In ELL, the multiplication is performed *max_columns* times, representing the maximum number of non-zero elements per row, as shown in line 15. As a result, the number of multiplications assigned to threads is consistent. However, unnecessary multiplications may arise due to zero-padding.

Figure 2.16 illustrates the CUDA Kernel for ELL-based SpMVT. In line 15, `atomicAdd` is used to accumulate the multiplication results of the transposed sparse matrix, and the final result is stored in the output vector $y$.

```
__global__ void KernelSpMVELL(const int n_rows,
                           const int max_columns,
                           const float* values,
                           const int* columns,
                           const float* x,
                           float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    float temp_sum = 0.0;
    if (i < n_rows) {
        int row_offset = max_columns * i;
        for (int j = 0; j < max_columns; j++) {
            temp_sum += values[row_offset + j] * x[columns
                [row_offset + j]];
        }
        y[i] = temp_sum;
    }
}
```

Figure 2.15: SpMV CUDA kernel for ELL.

```
__global__ void KernelSpMVTELL(const int n_rows,
                           const int max_columns,
                           const float* values,
                           const int* columns,
                           const float* x,
                           float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    if (i < n_rows) {
        int row_offset = max_columns * i;
        for (j = 0; j < max_columns; j++) {
            atomicAdd(&y[col[row_offset + j]], values[
                row_offset + j] * x[i];
        }
    }
}
```

Figure 2.16: SpMVT CUDA kernel for ELL.

## 2.6 Related works

Implementing existing sparse matrix storage formats straightforwardly poses challenges in terms of memory access and load balancing. For instance, in the case of CSR, when each thread processes all non-zero elements within a single row, memory accesses for each thread do not achieve coalesced access. Additionally, load imbalance can occur due to variations in non-zero elements per row. Many studies have been conducted to enhance existing sparse matrix storage formats in these aspects.

In the research by Anzt et al., they address these issues specifically for COO [1]. They achieve load balancing by partitioning non-zero elements into chunks of the same size and assigning these chunks to warps. Furthermore, they ensure full coalesced access in all memory accesses by processing the assigned non-zero elements for each thread using a warp-sized stride.

Koza et al. proposed Compressed Multi-Row Storage (CMRS), which achieves load balancing and coalesced memory access by assigning a fixed number of rows, referred to as a 'strip', to a single thread block [14]. However, CMRS compromises load balancing when the number of non-zero elements per row varies significantly, due to its fixed-size row to thread block partitioning. CSR-Adaptive, proposed by Greathouse et al., addresses this issue by statically fixing the number of non-zero elements per thread block and dynamically calculating the number of rows each thread block handles [10]. This approach allows multiple rows with a small number of non-zero elements to be assigned to a single thread block, while rows with a large number of non-zero elements are assigned to their respective thread blocks, ensuring load balance even when the number of non-zero elements per row varies significantly. However, it does not handle load balancing for matrices with rows containing an extremely large number of non-zero elements. Daga et al. extended CSR-Adaptive to address matrices with rows containing an extremely large number of non-zero elements by introducing a strategy to process such rows using multiple thread blocks [6].

CSR5 uniformly divides non-zero elements into predetermined-sized 2D tiles and assigns them to all threads, achieving load balance regardless of the sparse or dense structure of the matrix [16]. In HOLA, each thread block is assigned the same number of non-zero elements [22]. The number of non-zero elements to be processed per thread block is predetermined. To facilitate this, a buffer is prepared in advance to store row index in which non-zero elements processed by the thread block exist. Additionally, the number of non-zero elements to be processed by each thread within a thread block is also made equal. To handle cases where there are no non-zero elements within a row, a flag indicating the absence of non-zero elements in that row

is maintained to avoid thread divergence. Merrill et al. propose a method for load balancing using the merge-path technique [18]. With merge-path, each thread is assigned work in such a way that they have the same number of non-zero elements plus *row_ptr* elements.

VCSR is an approach primarily focused on achieving L1 cache coalesced access [13]. It rearranges rows in order of the number of non-zero elements and bundles a L1 cache line sized set of rows together. By vertically aligning the non-zero elements, it ensures that a single memory access accommodates all non-zero elements within a L1 cache line.

Gale et al. reorganize rows in descending order of the number of non-zero elements to achieve load balance both among SMs and among threads [9]. They reverse-engineered the thread block scheduler using an approach which Pai proposes [20], ensuring that blocks are assigned to SMs in the order of their block indices and, consequently, that rows with a higher number of non-zero elements are allocated to SMs in a sequential manner.

## 2.7 Summary

In this chapter, we introduced various fundamental sparse matrix storage formats such as COO, CSR, and ELL, as well as the CSB, which is a storage format for sparse matrices compatible with SpMV and SpMVT. We also explained GPU architecture and considerations during computation.

We discussed the challenges of running SpMV and SpMVT within an application on GPUs, including the difficulties in executing them with basic sparse matrix storage formats, applying CSB to the GPU, and the remaining issues with eCSB. Basic sparse matrix storage formats often result in inefficient memory access to the output vector y when transposing because they primarily store values and indices in row-major order. Applying CSB directly to the GPU involves recursive processing based on the number of non-zero elements within a block, leading to warp divergence. It also suffers from non-contiguous memory access during warp execution and load imbalance due to differences in the number of non-zero elements between blocks. Furthermore, existing techniques like eCSB, which redefine CSB on the GPU, require more memory than CSR or COO and may not achieve sufficient load balancing.

In the next chapter, considering these challenges, we propose a sparse matrix storage format for efficient execution of SpMV and SpMVT within an application on GPUs.

# Chapter 3

# Proposed Method

## 3.1 Introduction

In this chapter, we introduce CSB implemented naively on GPUs as CSB-baseline, as well as GCSB, which is defined to optimize CSB-baseline in terms of load balancing and reducing L1 cache miss count.

eCSB, a redefined version of CSB for GPUs, requires more memory compared to CSR, resulting in less compression efficiency compared to the original CSB. Therefore, CSB-baseline is introduced to redefine CSB on GPUs in a way that it requires memory usage almost equivalent to CSR, similar to the original implementation. In the introduction of CSB-baseline, we explain the vectors used for storage, the required memory usage, and the CUDA Kernels for both SpMV and SpMVT. Additionally, we describe the trade-off between memory access efficiency and the required memory usage based on the block size $\beta \times \beta$.

Regarding the introduction of GCSB, we first explain the arrays used for storage and the required memory usage. Next, we introduce techniques that constitute GCSB, such as block-swizzle load balancing and grouped block element reordering. Finally, we describe the CUDA Kernels for SpMV and SpMVT when using GCSB.

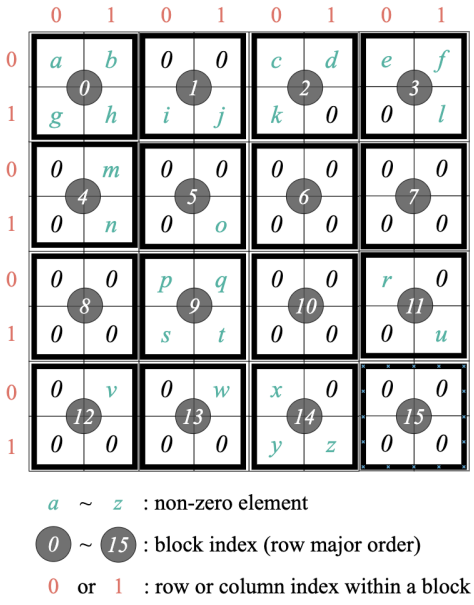
## 3.2 CSB on GPU

As demonstrated in Chapter 2, CSB is originally developed for CPU implementations. While eCSB efficiently computes SpMV and SpMVT on GPUs, it incurs a larger memory overhead compared to COO and does not achieve compression efficiency equivalent to CSR.

Hence, we first redefine CSB for GPU environments with compression efficiency equivalent to CSR, referring to it as CSB-baseline.

### 3.2.1 CSB-baseline

Similar to CPU-based CSB, CSB-baseline divides the matrix into blocks and reorders the non-zero elements in the order of blocks. Figure 3.1 illustrates the storage method of CSB-baseline for an $8 \times 8$ matrix example. In this example, the block size is $2 \times 2$, resulting in a total of 16 blocks arranged in row-major order. CSB-baseline employs arrays *values*, *comb_row_col_indices*, and $block\_offset$. *values* contains the values of non-zero elements. *comb_row_col_indices* holds the row and column indices within the block for non-zero elements, with the upper and lower bits representing the row and column indices, respectively. $block\_offset$ holds the indices within *values* or *comb_row_col_indices* of the first non-zero element in each block. The order of non-zero elements within a block follows Morton order, while the order of blocks follows row-major order. Alternatively, other orders of blocks are possible, but in that case, an array *block_indices* indicating the position of each block in row-major order is required.




(a) The matrix divided into $2 \times 2$ blocks.

$$values = [a\ b\ g\ h\ |\ i\ j\ |\ c\ d\ k\ |\ e\ f\ l\ |\ m\ n\ |\ o\ |\ p\ q\ s\ t\ |\ r\ u\ |\ v\ |\ w\ |\ x\ y\ z]$$

$$comb\_row\_col\_indices = [00\ 01\ 10\ 11\ |\ 10\ 11\ |\ 00\ 01\ 10\ |\ 00\ 01\ 11\ |\ 01\ 11\ |\ 11\ |\ 00\ 01\ 10\ 11\ |\ 00\ 11\ |\ 01\ |\ 01\ |\ 00\ 10\ 11]$$

$$block\_offset = [0\ 4\ 6\ 9\ 12\ 14\ 15\ 15\ 15\ 15\ 19\ 19\ 21\ 22\ 23\ 26\ 26]$$

$$(block\_indices = [0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15])$$

(b) Arrays for CSB-baseline. The black-bordered boxes represent each block.

Figure 3.1: CSB-baseline format.

The required memory usage for CSB-baseline is as follows:

$$\begin{aligned} MemoryUsage_{CSB-baseline} &= nnz \cdot sizeof(value) \\ &+ nnz \cdot sizeof(index) \\ &+ (n\_blocks + 1) \cdot sizeof(index) \end{aligned} \tag{3.1}$$

Next, we explain the CUDA Kernels for SpMV and SpMVT in the CSB-baseline. Figure 3.2 represents the SpMV kernel, and Figure 3.3 shows the SpMVT kernel. We provide an explanation assuming that blocks are arranged in row-major order. In the GPU Kernel for SpMV with CSB-baseline,

shown in Figure 3.2, each thread, indexed as thread $i$, processes non-zero elements in the block $i$. In line 17, the condition is set to ensure that the thread index does not exceed the number of blocks. In lines 18 and 19, the indices within *values* or *comb_row_col_indices* for the first and last+1 non-zero elements within each block are obtained. Line 20 contains a loop that iterates to process non-zero elements within the block sequentially. Lines 21 to 23 are responsible for calculating the row and column indices of the non-zero element within the block. The higher-order bits of *digit_column_per_block* within *comb_row_column_indices* represent the row index, while the lower bits represent the column index. In lines 24 and 25, the row and column indices for the block are computed. In line 26, the multiplication of the non-zero element with the corresponding value from vector $x$ is calculated, and the result is stored in the output vector $y$ using `atomicAdd`.

```
1  __global__ void KernelSpMVCSB(const int n_blocks,
2                        const int n_block_rows,
3                        const int n_block_columns,
4                        const int n_rows_per_block,
5                        const int n_columns_per_block,
6                        const int digit_column_per_block,
7                        const float* values,
8                        const int* comb_row_col_indices,
9                        const int* block_offset,
10                       const int* block_indices,
11                       const float* x,
12                       float* y) {
13     int idx = blockDim.x * blockIdx.x + threadIdx.x;
14     int idy = blockDim.y * blockIdx.y + threadIdx.y;
15     int offset = gridDim.x * blockDim.x;
16     int i = idx + idy * offset;
17     if (i < n_blocks) {
18         int start_block = block_offset[i];
19         int end_block = block_offset[i + 1];
20         for (int j = start_block; j < end_block; j++) {
21             int comb_row_col_index = comb_row_col_indices[
                   j];
22             int row_index_in_block = comb_row_col_index >>
                    digit_column_per_block;
23             int column_index_in_block = comb_row_col_index
                    & ((1 << digit_column_per_block) - 1);
24             int block_row_index = i / n_block_columns;
25             int block_column_index = i % n_block_rows;
26             atomicAdd(&y[block_row_index *
                   n_rows_per_block + row_index_in_block],
                   values[j] * x[block_column_index *
                   n_columns_per_block + column_index_in_block
                   ]);
27         }
28     }
29 }
```

Figure 3.2: SpMV CUDA kernel for CSB-baseline.

The difference between SpMV and SpMVT lies solely in the indexing during the write operation to the output vector $y$, where the indices of vectors $y$ and $x$ are reversed in line 26 in Figure 3.3.

```
__global__ void KernelSpMVTCSB(const int n_blocks,
                       const int n_block_rows,
                       const int n_block_columns,
                       const int n_rows_per_block,
                       const int n_columns_per_block,
                       const int digit_column_per_block,
                       const float* values,
                       const int* comb_row_col_indices,
                       const int* block_offset,
                       const int* block_indices,
                       const float* x,
                       float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    if (i < n_blocks) {
        int start_block = block_offset[i];
        int end_block = block_offset[i + 1];
        for (int j = start_block; j < end_block; j++) {
            int comb_row_col_index = comb_row_col_indices[
                j];
            int row_index_in_block = comb_row_col_index >>
                digit_column_per_block;
            int column_index_in_block = comb_row_col_index
                & ((1 << digit_column_per_block) - 1);
            int block_row_index = i / n_block_columns;
            int block_column_index = i % n_block_rows;
            atomicAdd(&y[block_column_index *
               n_columns_per_block + column_index_in_block
               ], value[j] * x[block_row_index *
               n_rows_per_block + row_index_in_block]);
        }
    }
}
```

Figure 3.3: SpMVT CUDA kernel for CSB-baseline.

By setting the the block size $\beta = \min(\sqrt{m}, \sqrt{n})$, it is possible to achieve memory usage nearly equivalent to CSR. However, caution is advised when $\beta$ becomes excessively large. A larger $\beta$ can result in increased address locations of the output vector $y$ accessed by each thread, leading to cache contention among threads and resulting in a bottleneck. On the other hand, reducing $\beta$ increases the number of blocks $n\_blocks$, thereby expanding the required memory capacity. In the extreme case, when $\beta = 1$, $n\_blocks = m \times n$.

## 3.3 GCSB

Through CSB-baseline, sparse matrices can be compressed with memory usage equivalent to CSR when *n_blocks* equals to *n_rows*. However, CSB-baseline faces challenges, as observed in Section 2.4, including the inability to achieve coalesced memory access due to each thread being assigned a single block, and the issue of load imbalance among SMs and Warp/Thread units stemming from varying numbers of non-zero elements per block. To address these challenges, we introduce Grouped Compressed Sparse Blocks (GCSB) as our proposed solution. The features of GCSB include: (1) Compression of sparse matrices with memory usage equivalent to CSR and CSB-baseline, (2) simultaneous achievement of load balancing among SMs and within warps using *block-swizzle load balancing*, (3) attainment of coalesced memory access through *grouped block element reordering*, achieved by grouping several blocks and rearranging non-zero elements within the group based on the number of elements that fit in an L1 cache line. The storage procedure for sparse matrices using GCSB is described in detail in Section 3.3.1. Block-swizzle load balancing is explained in Section 3.3.3. Grouped block element reordering is covered in Section 3.3.4. GPU Kernels for SpMV and SpMVT using GCSB are explained in Section 3.3.5.

### 3.3.1 GCSB format

The sparse matrix storage procedure of GCSB consists of the following four steps:

(1) Divide the matrix into blocks of $\beta \times \beta$.

(2) Reorder the blocks in descending order of the number of non-zero elements within each block.

(3) Group the blocks in *group_size*, starting with those having the most non-zero elements.

(4) Reorder the non-zero elements within each group and applying zero-padding.

These steps are illustrated in Figure 3.4 using an example of an $8 \times 8$ matrix. Steps (1) to (4) correspond to (a) to (d) in Figure 3.4. The block size $\beta \times \beta$ is a variable parameter. In this example, it is set to $2 \times 2$. Additionally, *group_size* is set to 4 in this example, while it is also a variable parameter.

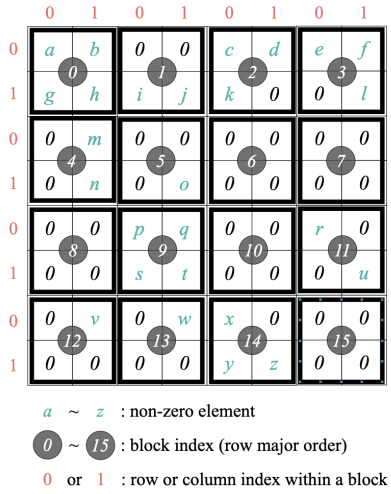
First, as illustrated in Figure 3.4 (a), the matrix is divided into $2 \times 2$ blocks. In this step, each block is assigned a block index in row-major

order, and the row and column indices of each element within the blocks are provided.

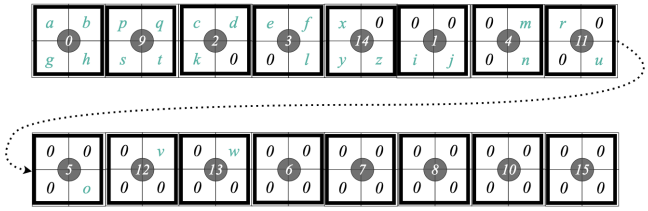
Next, as shown in Figure 3.4 (b), the blocks are reordered in the order of having a higher number of non-zero elements within each block. This technique is referred to as *block-swizzle load balancing*, and its detailed effects is discussed in Section 3.3.3.

Subsequently, as presented in Figure 3.4 (c), the blocks with a higher number of non-zero elements are grouped in chunks of $group\_size$. In this example, one group consists of four blocks, and a total of four groups, labeled as groupA to groupD, are generated.

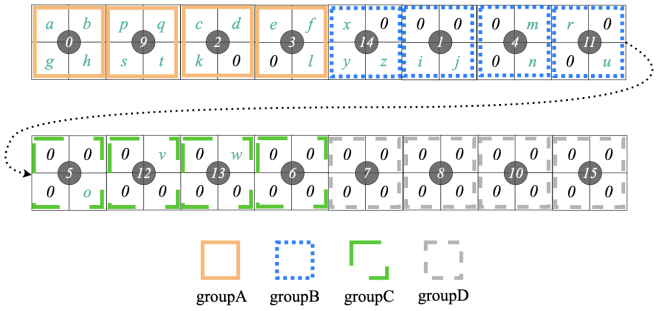
Finally, as depicted in Figure 3.4 (d), the non-zero elements within each group are rearranged. Non-zero elements are sequentially stored in the array with non-zero elements, starting from the group consisting of blocks with a larger number of non-zero elements. Within each group, non-zero elements are stored in the order of the first non-zero element in each block, followed by the second non-zero element in each block, and so on until all non-zero elements from all blocks are stored. The non-zero elements within each block in GCSB are stored in Morton order. In cases where there is variability in the number of non-zero elements within blocks in a group, zero-padding is applied, as indicated by $*$ in Figure 3.4 (d), to align the storage order. This reordering method is referred to as *grouped block element reordering*, and further details are explained in Section 3.3.4.



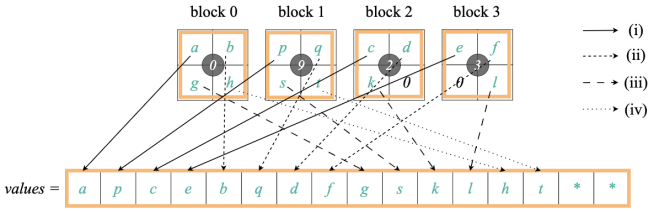

(a) Divide the matrix into blocks of $\beta \times \beta$.



(b) Reorder the blocks in descending order of the number of non-zero elements within each block.




(c) Group the blocks in *group_size*, starting with those having the most non-zero elements.




(d) Reorder the non-zero elements within each group and applying zero-padding.

Figure 3.4: The procedure for creating the GCSB format.

Figure 3.5 illustrates the arrays that compose GCSB. *values* is an array that stores the values of non-zero elements by repeating the procedure shown in Figure 3.4 (d) in order, starting from groups with a larger number of non-zero elements. *comb_row_col_indices* is an array that stores the indices of rows and columns of the non-zero elements within the blocks, following the same procedure as *values*. The row and column indices within the blocks are composed similarly to CSB-baseline, with the upper bits representing the row indices within a block and the lower bits representing the column indices within a block. The elements of *values* and *comb_row_col_indices* also include the zero-padding introduced in Figure 3.4 (c). *block_indices* is an array designed to keep track of where each block, reordered in the order of a larger number of non-zero elements, is positioned within the original matrix. The values stored in *block_indices* represent the indices of the blocks in row major order within the matrix. The original row index within the matrix is calculated as $block\_indices[i]$ / $n\_block\_rows$, while the column index is determined as $block\_indices[i]$ % $n\_block\_columns$. $group\_offset$ serves as a pointer indicating the location of the first non-zero element within a group in either *values* or *comb_row_col_indices*.

$$
\begin{aligned}
values = [&a\ p\ c\ e\ b\ q\ d\ f\ g\ s\ k\ l\ h\ t\ *\ *\\
&x\ i\ m\ r\ y\ j\ n\ u\ z\ *\ **\\
&o\ v\ w\ *]\\
comb\_row\_col\_indices = [&00\ 00\ 00\ 00\ 01\ 01\ 01\ 01\ 10\ 10\ 10\ 11\ 11\ 11\ *\ *\\
&00\ 10\ 01\ 00\ 10\ 11\ 11\ 11\ 11\ *\ **\\
&11\ 01\ 01\ *]\\
block\_indices = [&0\ 9\ 2\ 3\ 14\ 1\ 4\ 11\ 5\ 12\ 13\ 6\ 7\ 8\ 10\ 15]\\
group\_offset = [&0\ 16\ 28\ 32\ 32]
\end{aligned}
$$

Figure 3.5: GCSB format in the example of Figure 3.4. The boxes distinguished by color and line type are corresponding to the groups in Figure 3.4 (c). The $*$ symbol indicates locations that have been zero-padded in the process shown in Figure 3.4 (d). The upper bits of *comb_row_col_indices* value represent row indices of the non-zero elements within the block, while the lower bits represent column indices of the non-zero elements within the block. $group\_offset$ maintains pointers to the first index of each group within the array containing both non-zero elements and zero-padding.

Denoting the number of groups as $n\_groups$ and supposing that there

is no zero-padding, the theoretical memory usage required for GCSB is as follows:

$$\begin{aligned} MemoryUsage_{GCSB} = &(nnz) \cdot sizeof(value) \\ &+ (nnz) \cdot sizeof(index) \\ &+ n\_blocks \cdot sizeof(index) \\ &+ (n\_groups + 1) \cdot sizeof(index) \end{aligned} \tag{3.2}$$

Considering that $nnz >> n\_blocks$ and $n\_blocks > n\_groups$, GCSB achieves higher memory compression efficiency compared to eCSB, allowing for the compression of sparse matrices with memory usage nearly equivalent to CSR or CSB-baseline.

### 3.3.2 Determining $\beta$ and group_size

The block size $\beta \times \beta$ of GCSB is an arbitrary parameter. Considering that each thread performs multiplication for one block in SpMV and SpMVT with GCSB, we decide the following to optimize the utilization of L1 cache during memory access to the output vector y for each thread:

$$\beta \leq \frac{L1CacheLineSize}{sizeof(value)} \tag{3.3}$$

where $L1CacheLineSize$ is in bytes, and $sizeof(value)$ is also in bytes. In the examples seen in Figure 3.4 and 3.5, $L1CacheLineSize$ is set to 16 bytes, and $sizeof(value)$ is set to 4 bytes, resulting in $\beta = 2 \leq 16/4 = 4$. It is important to note the trade-off of increased memory usage in *block_indices* due to the increased number of blocks as the $\beta$ decreases, despite the increased parallelism.

The *group_size* is also an arbitrary parameter; however, to optimize the utilization of the L1 cache during memory access to values and *comb_row_column_indices*, we decide the following:

$$group\_size = \frac{L1CacheLineSize}{sizeof(value)}. \tag{3.4}$$

In the examples seen in Figure 3.4 and 3.5, with $L1CacheLineSize$ set to 16 bytes and $sizeof(value)$ set to 4 bytes, we have $group\_size = 4 = 16/4$. The efficiency of L1 cache utilization by setting *group_size* to the number of elements that fit in an L1 cache line is explained in Section 3.3.4.

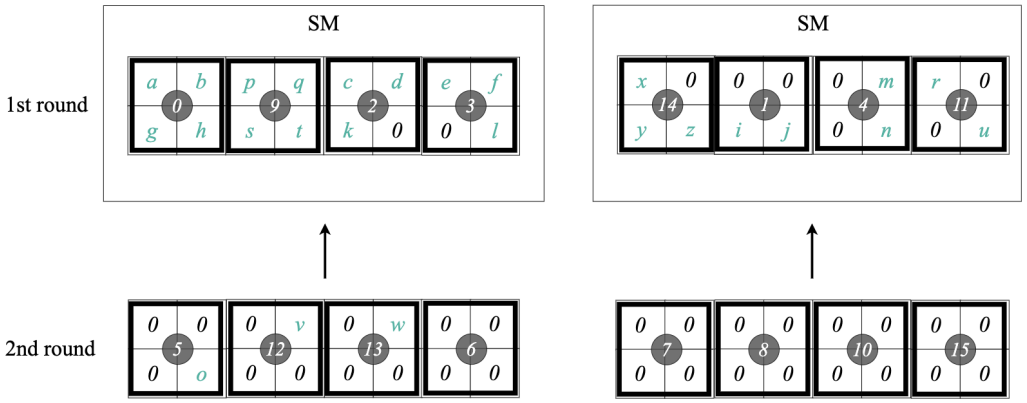
### 3.3.3 Block-swizzle load balancing

As discussed in Section 2.4, achieving load balancing is a key technique for improving the efficiency of GPU computations. Load imbalance can primarily occur in two ways [9]:

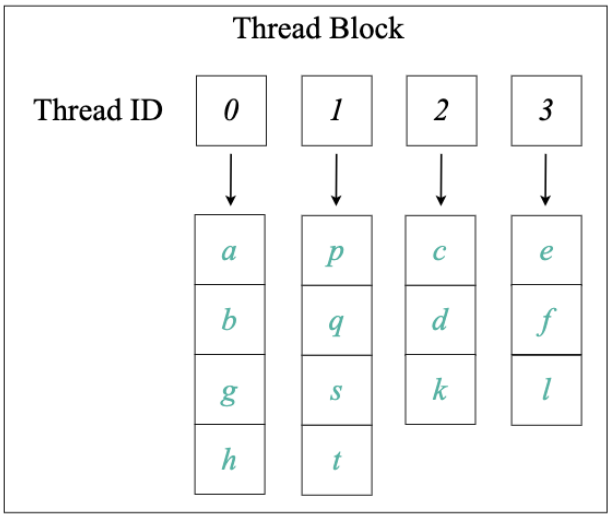
- Load imbalance among SMs.
- Load imbalance within a warp.

Addressing these issues is challenging since GPU scheduling algorithms are not publicly disclosed, making the implementation of load balancing complex.

To tackle this problem, Gale et al. [9] reverse-engineered [20] and gained insights into the mechanism of load imbalance through a heuristic experiment. From this experiment, it is determined that the mapping of thread blocks to SMs is conducted in a round-robin manner. After the initial wave, thread blocks are scheduled on SMs in the order of *block_index* as resources become available. Based on this heuristic experiment, Gale et al. propose *row-swizzle load balancing* as a solution to address the imbalance in load distribution. This approach involves reordering rows based on the number of non-zero elements within each row, arranging them in descending order of non-zero elements. In the context of GCSB, we extend this technique to blocks, introducing *block-swizzle load balancing*. Block-swizzle load balancing is a technique that achieves load balancing by reordering blocks based on the number of non-zero elements within each block in descending order. Figures 3.6 (a) and (b) respectively illustrate the mechanisms of load balancing among SMs and load balancing among threads through block-swizzle load balancing. Concerning load balancing among SMs, rearranging blocks in order of a higher number of non-zero elements leads to a round-robin assignment of thread blocks with more non-zero elements, ensuring load balance among SMs. In the example in Figure 3.6 (a), it is assumed that one group corresponds to one thread block. In the first round, thread blocks with a higher number of non-zero elements, such as groupA and groupB, are assigned to each SM, and in the next round, the next blocks with a higher number of non-zero elements, such as groupC and groupD, are assigned. Within the same thread block, blocks with a similar number of non-zero elements are processed by individual threads, achieving load balancing among threads. In the example in Figure 3.6 (b), blocks with a similar number of non-zero elements are assigned to threads with thread IDs ranging from 0 to 3, and computations are performed accordingly.




(a) SM level load balancing.




(b) Thread level load balancing.

Figure 3.6: Block-swizzle load balancing.

### 3.3.4 Grouped block element reordering

As seen in Section 2.3, minimizing L1 cache misses is one of the methods to optimize computations on the GPU.

Memory operations on the GPU are performed per warp. The GPU cache line size is 128 bytes, and when 32 threads within a warp each request a 4-byte value, 128 bytes of data are allocated for each request.

In CSB-baseline, each thread is assigned one block, and the computation of the first element within each block is executed simultaneously. Therefore,

when each thread accesses the memory of the first element, the addresses of these elements are non-contiguous, resulting in one transaction per thread. Subsequently, when each thread accesses the memory of the second element, if the number of threads exceeds the number of L1 cache lines, another set of transactions occurs. Repeating this worst case scenario for each non-zero element results in a transaction for each non-zero element, becoming a bottleneck in SpMV and SpMVT.

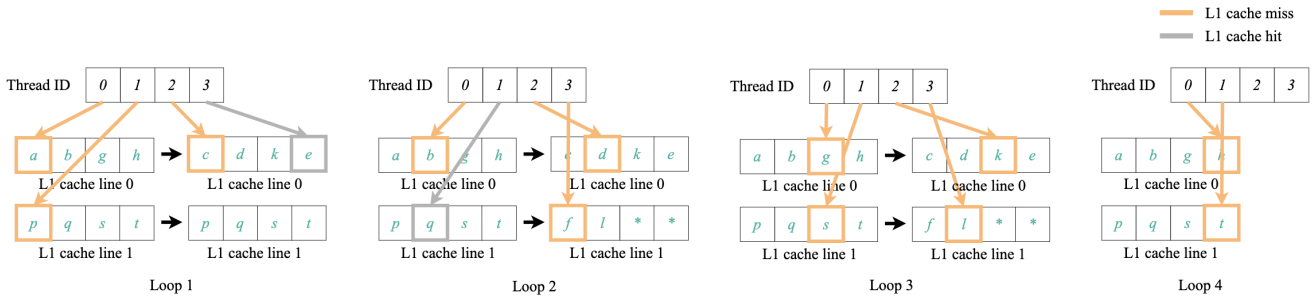
To address a similar issue in SpMV with CSR, VCSR, which is a method to prevent cache misses by rearranging the values of CSR vertically, has been proposed [13]. We have redefined this method for CSB and propose it as a technique to enhance CSB-baseline, referred to as *grouped block element reordering*.

Grouped block element reordering is a technique that involves grouping multiple blocks together and rearranging the non-zero elements within the blocks, as illustrated in Figure 3.4 (b), to enable coalesced memory access within the SM. The size of a group of blocks is designed to match the number of element that fit in an L1 cache line. By accessing memory in units of an L1 cache line, coalesced memory access is achieved.

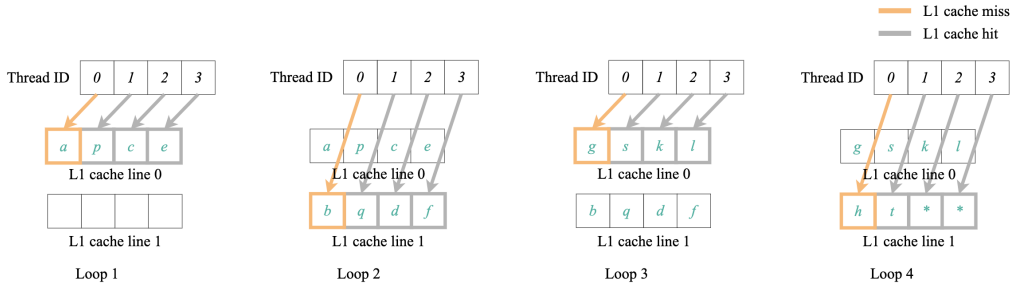
Figures 3.7 (a) and (b) respectively illustrate examples of L1 cache misses before and after the application of grouped block element reordering. In this example, the L1 cache has 2 cache lines, each with a size of 4, and the group size is set to 4. It is assumed that no zero-padding is applied in the pattern before the application of grouped block element reordering. The Loop represents the loop in the kernel where each thread accesses non-zero elements within a block. Orange arrows represent L1 cache misses, while gray arrows represent L1 cache hits.

As shown in Figure 3.7 (a), before the application of grouped block element reordering, in the first loop, each thread incurs a total of three L1 cache misses when accessing the initial element of the block. Thread ID 3 experiences an L1 cache hit as there are already non-zero elements in L1 cache line 0. In the second loop, each thread incurs a total of three cache misses when accessing the second element of the block. In Loop 3, a total of four cache misses occur, and in Loop 4, a total of two cache misses occur. Before the application of grouped block element reordering, a total of 12 L1 cache misses are incurred.

On the other hand, as shown in Figure 3.7 (b), after the application of grouped block element reordering, there is only one L1 cache miss in each loop, resulting in a total of four L1 cache misses. This represents a reduction in L1 cache misses compared to the scenario before the application of grouped block element reordering.




(a) L1 cache misses without grouped block element reordering.




(b) L1 cache misses with grouped block element reordering.

Figure 3.7: Comparison of L1 cache misses between before and after application of grouped block element reordering.

More generally, the naive CSB approach, which could potentially result in $nnz$ L1 cache misses in the worst case scenario, i.e., when the number of L1 cache lines is much lesser than the number of blocks allocated to a single SM. On the other hand, our method significantly reduces L1 cache misses to $\sum_{i=1}^{\frac{n}{l}} \max(b_i)$, where $nnz$ (number of non-zero elements in the matrix), $n\_blocks$ (number of blocks), $l$ (the number of elements that fit in an L1 cache line), $nl$ (number of cache lines), and $max(bi)$ (maximum number of non-zero elements in a block within the group).

One important consideration in grouped block element reordering is the potential for a significant increase in memory usage due to zero-padding. When there is variation in the number of non-zero elements in each block within a group, the other blocks are zero-padded to match the block with the maximum number of non-zero elements. Assuming nnz $\gg$ blocks and memory usage for $block\_indices$ and $group\_offset$ are negligible, the required memory usage for GCSB is represented by the following equation:

$$MemoryUsage_{GCSB} \approx \sum_{i=1}^{n\_groups} max(b_i)l \cdot (sizeof(value) + sizeof(index)) \tag{3.5}$$

where $max(b'_i)$ represents the maximum number of non-zero elements in each block within the group. If there is a substantial difference between the maximum number of non-zero elements in the blocks within the group and the number of zero elements in other blocks, the required memory usage can significantly increase.

### 3.3.5 CUDA kernel for GCSB

Figures 3.8 and 3.9 show the CUDA Kernels for SpMV and SpMVT in GCSB, respectively.

In GPU Kernel for SpMV, each thread performs multiplication within a single block. To ensure that thread IDs do not exceed the number of blocks, the thread ID remains within the block count in line 18. The original position of thread $i$ within the processing block in line 19. This position is used to calculate the row and column indices of that block within the original matrix in lines 29 and 30, respectively. In Lines 20 and 21, the index of the group to which the block belongs and its position within that group are calculated. In lines 22 and 23, the start and the end + 1 index of non-zero elements of that group are obtained. In line 24, there's a loop that iterates over non-zero elements to be processed by thread $i$ within the block. In line 25, the index of the non-zero element to be processed in that loop is calculated. In lines 26 to 28, we extract the row and column indices of the non-zero element within the block. In line 31, we perform the multiplication with an element of vector $x$ and store the result in the output vector $y$ using `atomicAdd`.

```
__global__ void KernelSpMVGCSB(const int n_blocks,
                           const int n_block_rows,
                           const int n_block_columns,
                           const int n_rows_per_block,
                           const int n_columns_per_block,
                           const int
                              digit_column_per_block,
                           const int group_size,
                           const float* values,
                           const int* comb_row_col_indices
                              ,
                           const int* group_offset,
                           const int* block_indices,
                           const float* x,
                           float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    if (i < n_blocks) {
        int block_index = block_indices[i];
        int group_index = i / group_size;
        int offset_in_group = i % group_size;
        int group_start = group_offset[group_index];
        int group_end = group_offset[group_index + 1];
        for (int j = group_start; j < group_end; j++) {
            int k = j * group_size + offset_in_group
            int comb_row_col_index = comb_row_col_indices[
               k];
            int row_index_in_block = comb_row_col_index >>
                digit_column_per_block;
            int column_index_in_block = comb_row_col_index
                & ((1 << digit_column_per_block) - 1);
            int block_row_index = block_index /
               n_block_columns;
            int block_column_index = block_index %
               n_block_rows;
            atomicAdd(&y[block_row_index *
               n_rows_per_block + row_index_in_block],
               values[k] * x[block_column_index *
               n_columns_per_block + column_index_in_block
               ]);
        }
    }
}
```

Figure 3.8: SpMV CUDA kernel for GCSB.

Regarding SpMVT, apart from the exchange of row index and column index at line 31, it undergoes similar execution.

```
__global__ void KernelSpMVTGCSB(const int n_blocks,
                           const int n_block_rows,
                           const int n_block_columns,
                           const int n_rows_per_block,
                           const int n_columns_per_block,
                           const int
                              digit_column_per_block,
                           const int group_size,
                           const float* values,
                           const int* comb_row_col_indices
                              ,
                           const int* group_offset,
                           const int* block_indices,
                           const float* x,
                           float* y) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offset = gridDim.x * blockDim.x;
    int i = idx + idy * offset;
    if (i < n_blocks) {
        int block_index = block_indices[i];
        int group_index = i / group_size;
        int offset_in_group = i % group_size;
        int group_start = group_offset[group_index];
        int group_end = group_offset[group_index + 1];
        for (int j = group_start; j < group_end; j++) {
            int k = j * group_size + offset_in_group
            int comb_row_col_index = comb_row_col_indices[
               k];
            int row_index_in_block = comb_row_col_index >>
                digit_column_per_block;
            int column_index_in_block = comb_row_col_index
                & ((1 << digit_column_per_block) - 1);
            int block_row_index = block_index /
               n_block_columns;
            int block_column_index = block_index %
               n_block_rows;
            atomicAdd(&y[block_column_index *
               n_columns_per_block + column_index_in_block
               ], values[k] * x[block_row_index *
               n_rows_per_block + row_index_in_block]);
        }
    }
}
```

Figure 3.9: SpMVT CUDA kernel for GCSB.

## 3.4 Summary

In this chapter, we introduced our proposed methods: CSB-baseline, which redefines CSB for GPUs, and GCSB, which further optimizes CSB-baseline in terms of load balancing and L1 cache miss count.

Both CSB-baseline and GCSB are expected to achieve memory usage almost equivalent to CSR, compared to COO or eCSB. Furthermore, GCSB is expected to outperform CSB-baseline in SpMV or SpMVT due to the improvements in load balancing through block-swizzle load balancing and cache miss count reduction through grouped block element reordering.

However, both CSB-baseline and GCSB involve a trade-off between memory access and memory usage depending on the block size $\beta \times \beta$. Therefore, it is necessary to configure an appropriate $\beta$ based on the platform and matrix size. In the case of GCSB, there is a challenge when there is significant variability in the number of non-zero elements within a group, as it can lead to excessive memory usage allocation due to increased zero-padding.

# Chapter 4

# Evaluation

## 4.1 Introduction

In this chapter, we evaluate the performance of GCSB in SpMV and SpMVT through experiments. Firstly, we provide an overview of the platforms, sparse matrices used in the experiments, and the sparse matrix storage formats compared with GCSB. Next, we explain several metrics used to evaluate the performance of GCSB. We then describe the results of the GCSB experiments. The experiments include evaluations with matrices considered ideal for GCSB, evaluations using the University of Florida Sparse Matrix Collection, and evaluations with matrices where non-zero elements are randomly distributed. All matrices used in the experiments are square matrices with equal row and column sizes, i.e., $m = n$.

In the experiments with matrices considered ideal for GCSB, we experimentally determine the block size for GCSB and evaluate its performance in terms of execution time, theoretical memory usage, and L1 cache miss counts to evaluate if GCSB is functioning as intended.

For the experiments using the University of Florida Sparse Matrix Collection, we evaluate whether GCSB can achieve theoretical memory usage equivalent to CSR, if it is faster than CSR and CSB-baseline, and identify specific characteristics of sparse matrices that allow GCSB to excel.

In the experiments with matrices where non-zero elements are randomly distributed, we investigate how the experimental results vary based on the proportion of non-zero elements and discuss the conditions under which GCSB performs well.

## 4.2 Evaluation conditions

We begin by describing the experimental setup and evaluation criteria. Next, we conduct experiments using ideal matrices for GCSB to assess whether each technique of GCSB functions effectively. Finally, we compare the performance of SpMV and SpMVT with GCSB and other matrix storage formats using various sparse matrices.

### 4.2.1 Experimental setup

All our experiments are conducted on NVIDIA TITAN RTX and NVIDIA A100 GPUs. Table 4.1 presents the system specifications for these two GPUs. The matrices used for SpMV and SpMVT calculations include intentionally created sparse matrix that is ideal for GCSB, as well as several sparse matrices selected from the University of Florida Sparse Matrix Collection, and sparse matrices with uniformly random distributions of non-zero elements. Table 4.2, 4.3 and 4.4 respectively provide the row and column counts, the number of non-zero elements, and the percentage of non-zero elements in each of these sparse matrices. Figure 4.1 shows the plot of space matrices from the University of Florida Sparse Matrix Collection [7]. The performance of GCSB is compared to COO, CSR, eCSB, and CSB-baseline. It is worth noting that the results of SpMV and SpMVT calculations with eCSB are based on COO-based methods.

Table 4.1: System Specifications.

| Parameter | TITAN RTX | A100 |
|---|---|---|
| Architecture | NVIDIA Turing | NVIDIA Ampere |
| SMs | 72 | 108 |
| FP32 Cores / SM | 64 | 64 |
| FP64 Cores / SM | - | 32 |
| FP32 Cores / GPU | 4,608 | 6,912 |
| FP64 Cores / GPU | - | 3,456 |
| Compute capability | 7.5 | 8.0 |
| Number of cores | 4,608 | 6,912 |
| Peak FP32 | 16.3 TFLOPS | 19.5 TFLOPS |
| Peak FP64 | - | 9.7 TFLOPS |
| Memory size | 24GB | 40 GB |
| L1 size | 64 KB | 192 KB |
| L1 line size | 128 B | 128 B |
| L2 size | 6,144 KB | 40,960 KB |
| L2 line size | 32 B | 64 B |

Table 4.2: Ideal sparse matrix for GCSB.

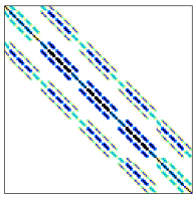
| Name | Row/column | Non-zeros | Ratio of non-zeros (%) |
|---|---|---|---|
| ideal_matrix | 8.19K/8.19K | 3,406.20K | 5.00 |

Table 4.3: Benchmark matrices from the University of Florida Sparse Matrix Collection.

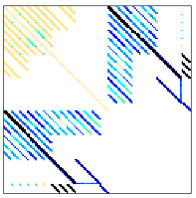
| Name | Row/column | Non-zeros | Ratio of non-zeros (%) |
|---|---|---|---|
| msc00726 | 0.73K/0.73K | 17.62K | 3.34 |
| spaceStation_4 | 0.95K/0.95K | 7.1K | 0.78 |
| CAG_mat1916 | 1.92K/1.92K | 195.99K | 5.34 |
| heart2 | 2.34K/2.34K | 682.80K | 12.48 |
| psmigr_3 | 3.14K/3.14K | 543.16K | 5.51 |
| raefsky6 | 3.40K/3.40K | 137.85K | 1.19 |
| heart1 | 3.56K/3.56K | 1,387.77K | 10.97 |
| exdata_1 | 6.00K/6.00K | 1,137.75K | 3.16 |
| TSC_OPF_1047 | 8.14K/8.14K | 1,012.52K | 1.53 |
| nemeth26 | 9.51K/9.51K | 760.63K | 0.84 |
| sme3Da | 12.50K/12.50K | 874.89K | 0.56 |
| appu | 14.00K/14.00K | 1,853.10K | 0.95 |
| human_gene2 | 14.34K/14.34K | 9,041.36K | 4.40 |
| olafu | 16.15K/16.15K | 515.65K | 0.20 |
| nd6k | 18.00K/18.00K | 3,457.66K | 1.07 |
| human_gene1 | 22.28K/22.28K | 12,345.96K | 2.49 |
| nd12k | 36.00K/36.00K | 7,128.47K | 0.55 |
| mouse_gene | 45.10K/45.10K | 14,506.20K | 0.71 |

Table 4.4: Uniformly random sparse matrices.

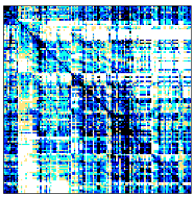
| Name | Row/column | Non-zeros | Ratio of non-zeros (%) |
|---|---|---|---|
| random_0.0001 | 8.19K/8.19K | 6.71K | 0.01 |
| random_0.0005 | 8.19K/8.19K | 33.55K | 0.05 |
| random_0.001 | 8.19K/8.19K | 67.11K | 0.10 |
| random_0.005 | 8.19K/8.19K | 335.55K | 0.50 |
| random_0.01 | 8.19K/8.19K | 671.09K | 1.00 |
| random_0.05 | 8.19K/8.19K | 3,355.44K | 5.00 |
| random_0.1 | 8.19K/8.19K | 6,710.89K | 10.00 |
| random_0.2 | 8.19K/8.19K | 13,421.77K | 20.00 |
| random_0.3 | 8.19K/8.19K | 20,132.66K | 30.00 |
| random_0.4 | 8.19K/8.19K | 26,843.55K | 40.00 |
| random_0.5 | 8.19K/8.19K | 33,554.43K | 50.00 |



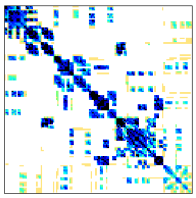
(a) msc00726



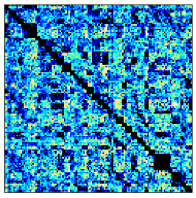
(b) spaceStation_4



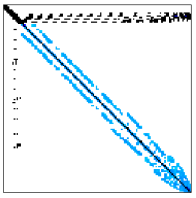
(c) CAG_mat1916



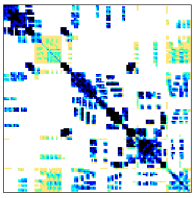
(d) heart2



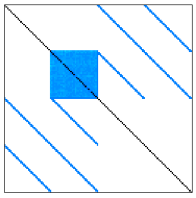
(e) psmigr_3



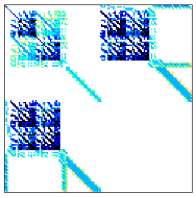
(f) raefsky6



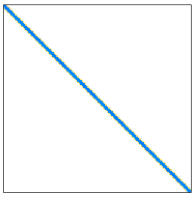
(g) heart1



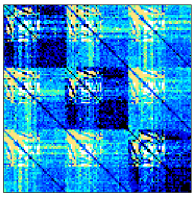
(h) exdata_1



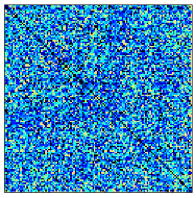
(i) TSC_OPF_1047



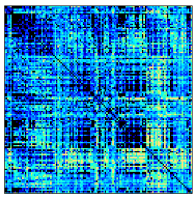
(j) nemeth26



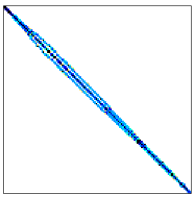
(k) sme3Da



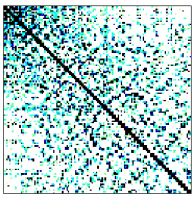
(l) appu



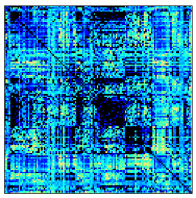
(m) human_gene2



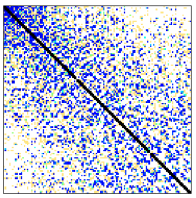
(n) olafu



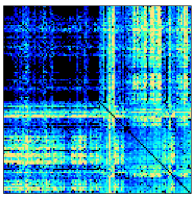
(o) nd6k



(p) human_gene1



(q) nd12k



(r) mouse_gene

Figure 4.1: Plot of sparce matrices from the University of Florida sparse matrices [7].

For both TITAN RTX and A100, $\beta$ and $group\_size$ in GCSB are set as follows, using (3.3), (3.4) and L1 cache line size of both platforms in Table 4.1:

$$\begin{aligned} \beta &\leq \frac{L1CacheLineSize}{sizeof(value)} = \frac{128\text{B}}{4\text{B}} = 32 \\ group\_size &= \frac{L1CacheLineSize}{sizeof(value)} = \frac{128\text{B}}{4\text{B}} = 32 \end{aligned} \tag{4.1}$$

Regarding $\beta$, there is a trade-off between the level of parallelism and the theoretical memory usage depending on its size. Therefore, based on the results of experimenting with $\beta$ in various patterns in Section 4.3, we seek an appropriate $\beta$.

### 4.2.2 Evaluation aspects

We evaluate GCSB in terms of the execution times of SpMV and SpMVT, the required theoretical memory usage, and L1 cache miss counts.

First, we perform a comparison between GCSB and other sparse matrix storage formats using what we consider an ideal sparse matrix for GCSB. In this comparison, we assess GCSB in terms of the execution times of SpMV and SpMVT, required theoretical memory usage, and L1 cache miss counts. Initially, we vary $\beta$ of GCSB and CSB-baseline, comparing the execution times and required theoretical memory usage with those of other sparse matrix storage formats to determine suitable $\beta$ for GCSB and CSB-baseline. Subsequently, we compare the L1 cache miss counts between GCSB and CSB-baseline to verify the effectiveness of GCSB.

Next, we compare GCSB with other sparse matrix storage formats using the sparce matrices from the University of Florida Sparse Matrix Collection for SpMV and SpMVT calculations. In this comparison, we evaluate them based on the total speedup and the memory usage relative to CSR. Speedup is calculated using Eq.(4.2) [13]. The execution time is defined as the sum of the execution times for SpMV and SpMVT. The theoretical memory usage ratio is calculated using Eq.(4.3).

$$SpeedUp = \frac{ExecutionTime_{CSR}}{ExecutionTime_{SparseMatrixStorageFormat}} \tag{4.2}$$

$$MemoryUsageRatio = \frac{MemoryUsage_{SparseMatrixStorageFormat}}{MemoryUsage_{CSR}} \tag{4.3}$$

Furthermore, we vary the proportion of non-zero elements within matrices with uniformly random distributions of non-zero elements and evaluate

GCSB. We compare the speedup and the memory usage of SpMV and SpMVT relative to CSR. This allows us to observe how the performance of GCSB changes depending on the ratio of non-zero elements.

## 4.3 GCSB evaluation

In this section, we evaluate the effectiveness of GCSB using a matrix that is ideal for GCSB through a comparison with COO, eCSB, CSR and CSB-baseline. Figure 4.2 illustrates an ideal matrix for GCSB. This matrix has dimensions of $8192 \times 8192$ and contains 3,406,208 non-zero elements, which is approximately 5% of total non-zero elements in the entire matrix. An ideal matrix for GCSB is one in which the number of non-zero elements within each block is the same across blocks in the row direction of the matrix. Each color represents the magnitude of non-zero elements. Warm colors represent a higher number of non-zero elements, while cool colors represent a lower number of non-zero elements. Each row represents a block row, indicating that blocks within the same block row have the same number of non-zero elements. When such a matrix is transformed into GCSB, SpMV and SpMVT calculations achieve thread-level load balance as the number of non-zero elements in each block within a group becomes equal. Additionally, the addresses for the dense vector $x$ and output vector $y$, accessed by a single thread block, are contiguous. This results in fewer cache misses, minimizing the bottleneck associated with accessing dense vector $x$ and output vector $y$.

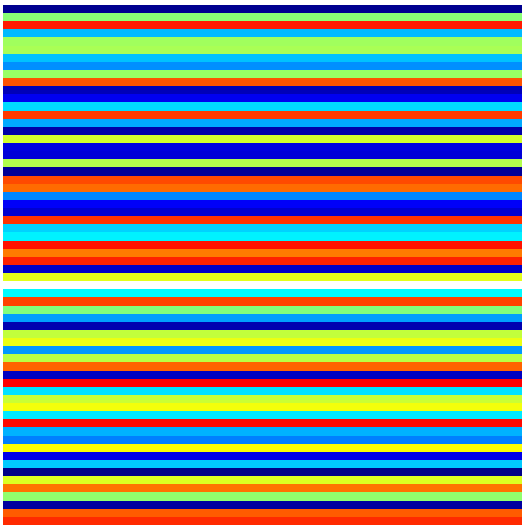


Figure 4.2: Ideal matrix for GCSB.

### 4.3.1 Evaluation on an ideal sparse matrix

We compare the results when the block sizes of CSB-baseline and GCSB are varied. CSB-baseline and GCSB assign each thread to handle one block, resulting in access to the output vector $y$ for a number of rows in the case of SpMV and a number of columns in the case of SpMVT. If the block size is too large, the number of addresses accessed within the output vector $y$ increases. Conversely, if the block size is too small, the number of blocks increases, leading to increased memory usage for *block_indices*. This trade-off necessitates setting an appropriate block size. In this comparison, the group size of GCSB is set to 32. This is because, in single-precision floating-point, one value is 4 bits, and the L1 cache line size of TITAN RTX is 128 bits, as is the L1 cache line size of NVIDIA A100, allowing for the storage of 16 elements in each cache line on either platform.

Table 4.5 and 4.6 show the kernel execution times for SpMV and SpMVT, respectively, for COO, eCSB, CSR, CSB-baseline, and GCSB when the block size is varied from 8 to 128 in powers of 2 for both TITAN RTX and A100. The execution times for COO and CSR remain constant as it is independent of the block size. The execution time of eCSB also remains unchanged, as $\beta$ of eCSB is fixed at $\beta = \min(\sqrt{n}, \sqrt{m})$. The results highlight that reducing the block size from 128 to 16 leads to shorter execution times for both SpMV and SpMVT in both CSB-baseline and GCSB. Both CSB-baseline and GCSB achieve a smaller total execution time for SpMV and SpMVT than CSR when reducing the block size to 32.

Table 4.5: Execution times of SpMV and SpMVT with an ideal matrix for GCSB when block size varied on TITAN RTX (msec).

| | COO | | eCSB | | CSR | | CSB-baseline | | GCSB | |
|---|---|---|---|---|---|---|---|---|---|---|
| Block size | SpMV | SpMVT | SpMV | SpMVT | SpMV | SpMVT | SpMV | SpMVT | SpMV | SpMVT |
| 128 | 0.48 | 0.23 | 0.12 | 0.12 | 0.21 | 0.75 | 0.91 | 0.82 | 0.93 | 0.93 |
| 64 | 0.48 | 0.23 | 0.12 | 0.12 | 0.21 | 0.75 | 0.83 | 0.81 | 0.34 | 0.32 |
| 32 | 0.48 | 0.23 | 0.12 | 0.12 | 0.21 | 0.75 | 0.75 | 0.73 | 0.22 | 0.24 |
| 16 | 0.48 | 0.23 | 0.12 | 0.12 | 0.21 | 0.75 | 0.60 | 0.43 | 0.20 | 0.23 |
| 8 | 0.48 | 0.23 | 0.12 | 0.12 | 0.21 | 0.75 | 0.47 | 0.26 | 0.23 | 0.26 |

Table 4.6: Execution times of SpMV and SpMVT with an ideal matrix for GCSB when block size varied on A100 (msec).

| | COO | | eCSB | | CSR | | CSB-baseline | | GCSB | |
|---|---|---|---|---|---|---|---|---|---|---|
| Block size | SpMV | SpMVT | SpMV | SpMVT | SpMV | SpMVT | SpMV | SpMVT | SpMV | SpMVT |
| 128 | 0.50 | 0.21 | 0.12 | 0.11 | 0.15 | 0.50 | 0.87 | 0.88 | 0.91 | 0.90 |
| 64 | 0.50 | 0.21 | 0.12 | 0.11 | 0.15 | 0.50 | 0.51 | 0.55 | 0.38 | 0.41 |
| 32 | 0.50 | 0.21 | 0.12 | 0.11 | 0.15 | 0.50 | 0.18 | 0.28 | 0.17 | 0.27 |
| 16 | 0.50 | 0.21 | 0.12 | 0.11 | 0.15 | 0.50 | 0.15 | 0.25 | 0.14 | 0.25 |
| 8 | 0.50 | 0.21 | 0.12 | 0.11 | 0.15 | 0.50 | 0.26 | 0.24 | 0.13 | 0.25 |

Table 4.7 provides the required memory usage for COO, eCSB, CSR, CSB-baseline, and GCSB at each block size. The required memory usage for those sparse matrix storage formats are the same on both TITAN RTX and A100. The kernels for SpMV and SpMVT using each sparse matrix storage format only reverse the matrix indices during write operations, and they utilize the same arrays. Therefore, the required memory usage is the same for both SpMV and SpMVT. The memory usage for COO, eCSB and CSR remains unchanged as it is not influenced by the block sizes of CSB-baseline and GCSB. CSB-baseline and GCSB have memory footprints similar to CSR, compared to COO and eCSB. For block sizes of 16 or lower, the required memory usage of CSB-baseline and GCSB exceeds that of CSR by more than 4 MB.

Table 4.7: Required memory usage with an ideal matrix for GCSB when block size varied (MB).

| Block size | COO | eCSB | CSR | CSB-baseline | GCSB |
|---|---|---|---|---|---|
| 128 | 163.50 | 163.50 | 109.13 | 109.06 | 109.07 |
| 64 | 163.50 | 163.50 | 109.13 | 109.26 | 109.50 |
| 32 | 163.50 | 163.50 | 109.13 | 110.05 | 110.15 |
| 16 | 163.50 | 163.50 | 109.13 | 113.19 | 113.35 |
| 8 | 163.50 | 163.50 | 109.13 | 125.78 | 126.13 |

We compare the L1 cache miss counts for CSB-baseline and GCSB with a block size of 32, using an ideal matrix for GCSB. L1 cache miss counts refer to cache misses at the sector level. A sector represents a 32-byte chunk within an L1 cache line, and a 128-byte L1 cache line is divided into four sectors. Access to a sector is considered a miss when sector data is not present within the cache line. The counts of L1 cache misses are obtained using NVIDIA

profiler. Table 4.8 presents the L1 cache miss counts for SpMV and SpMVT on each platform. The L1 cache miss counts include cache misses resulting from accessing the vector *values*, *comb_row_col_indices*, *block_indices*, $block\_offset$ or $group\_offset$, as well as accesses to vectors $x$ and the output vector $y$. Many of these cache misses are primarily attributed to accesses to *values* and *comb_row_col_indices*, which constitute a significant portion of the memory required for Kernel execution. From these results, it can be observed that in both the TITAN RTX and A100, the L1 cache miss counts for SpMV and SpMVT are lower for GCSB compared to CSB-baseline.

Table 4.8: Comparison of L1 cache miss counts between CSB-baseline and GCSB.

| | CSB-baseline | | GCSB | |
|---|---|---|---|---|
| Platform | SpMV | SpMVT | SpMV | SpMVT |
| TITAN RTX | 4,846K | 2307K | 961K | 897K |
| A100 | 1,521K | 1,148K | 964K | 898K |

## 4.4 Comparison of GCSB with various other sparse matrix storage formats across different matrices

In this section, we evaluate the performance of GCSB using various matrices selected from the University of Florida Sparse Matrix Collection and sparse matrices with uniformly random non-zero elements. The comparison includes COO, eCSB, CSR, and CSB-baseline as reference storage formats. For evaluation, we compare the speedup and the memory usage relative to those of CSR.

### 4.4.1 Performance evaluation of GCSB on the University of Florida Sparse Matrix Collection

First, we compare the speedup and memory usage of other sparse matrix storage formats relative to CSR using various matrices selected from the University of Florida Sparse Matrix Collection.

Figures 4.3 and 4.4 show the speedup of COO, eCSB, CSB-baseline, and GCSB relative to CSR on TITAN RTX and A100, respectively. The x-axis of the figures displays matrices sorted in ascending order of their sizes

from left to right. GCSB is 1.45×, 1.47×, 1.27×, and 1.23× faster than CSR for the matrices appu, human_gene2, human_gene1, and mouse_gene, respectively on TITAN RTX. Furthermore, GCSB outperforms CSB-baseline for these matrices. GCSB is 1.08×, 1.41×, 1.41×, 2.10×, and 2.75× faster than CSR for the matrices sm3Da, appu, human_gene2, human_gene1, and mouse_gene, respectively on A100. Additionally, GCSB is faster than CSB-baseline for the matrices appu, human_gene1, and mouse_gene. GCSB outperforms COO for appu, human_gene1, and mouse_gene matrices in both TITAN RTX and A100. GCSB is slower than eCSB for all matrices.

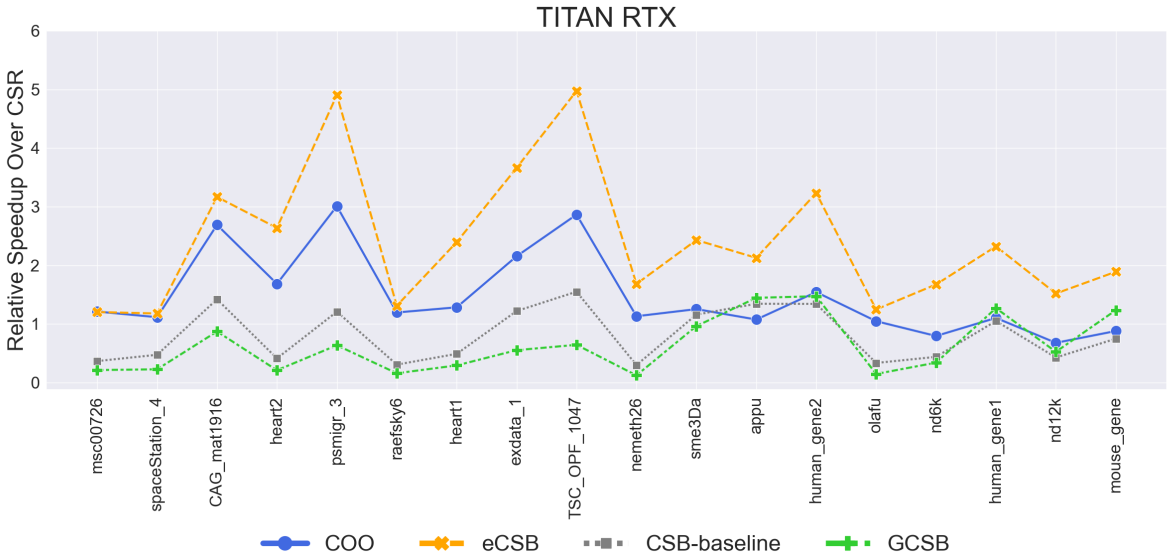



Figure 4.3: Speedup of the total execution time of SpMV and SpMVT for COO, eCSB, CSB-baseline, and GCSB on the University of Florida Sparse Matrix Collection compared to CSR on TITAN RTX.

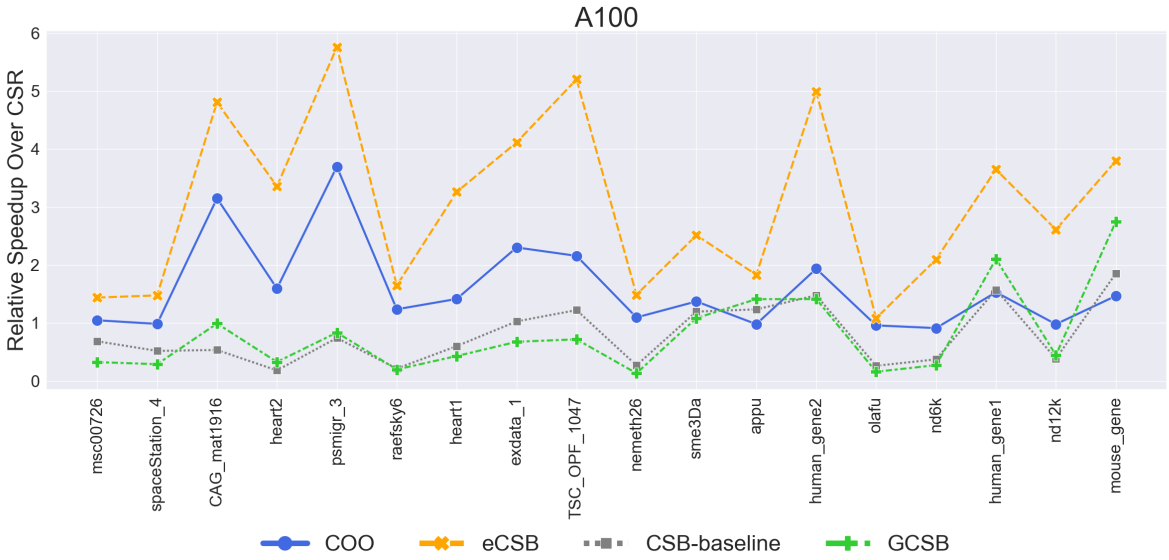



Figure 4.4: Speedup of the total execution time of SpMV and SpMVT for COO, eCSB, CSB-baseline, and GCSB on the University of Florida Sparse Matrix Collection compared to CSR on A100.

Based on these results and Figure 4.1, GCSB achieves a speedup over CSR in sparse matrices where non-zero elements are distributed across the entire matrix, unlike the other sparse matrices. Although psmigr_3 and CAG_mat1916 also exhibit a relatively even distribution of non-zero elements across the matrix, these matrices are smaller in size compared to the sparse matrices where GCSB achieves a speedup over CSR.

Figures 4.5 shows the memory usage of COO, eCSB, CSB-baseline, and GCSB relative to CSR on TITAN RTX and A100. The memory usage of SpMV or SpMVT is consistent across both TITAN RTX and A100 platforms. The x-axis in Figures 4.5 displays matrices sorted in ascending order based on their sizes. For matrices where GCSB achieves a speedup of more than 1.0 against CSR on both TITAN RTX or A100, the memory usage of GCSB relative to CSR is 1.09× for sm3Da, 1.05× for appu, 1.01× for human_gene2, 1.02× for human_gene1, and 1.07× for mouse_gene. This indicates that GCSB has a smaller memory footprint compared to eCSB and consumes memory similar to CSR.

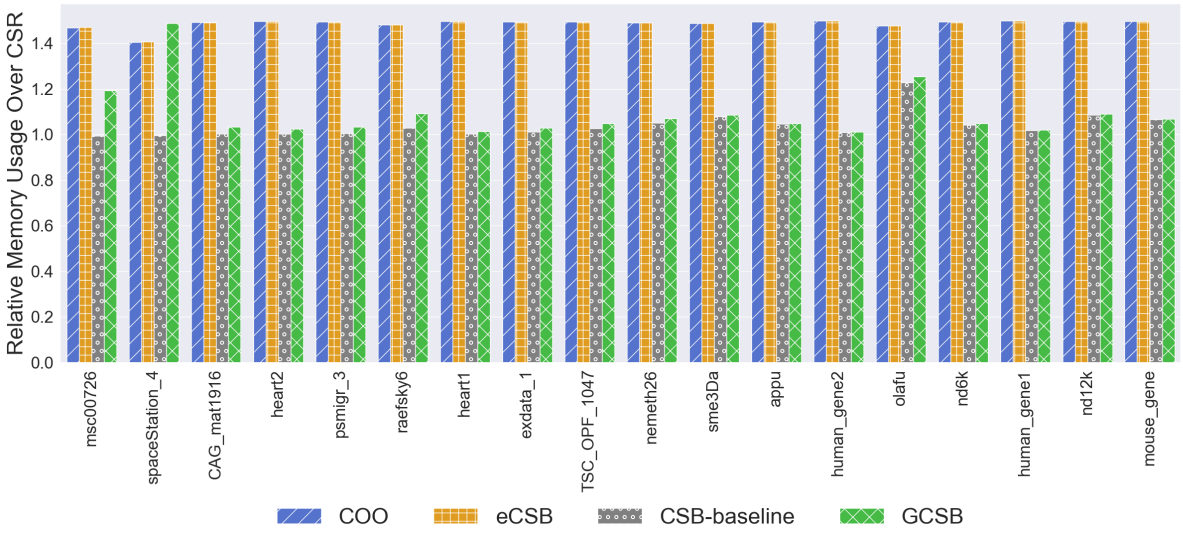



Figure 4.5: Memory usage of SpMV or SpMVT for COO, eCSB, CSB-baseline, and GCSB on the University of Florida Sparse Matrix Collection compared to CSR.

### 4.4.2 Performance evaluation of GCSB on random sparse matrices

Next, we compare the speedup and memory usage of CSR, COO, eCSB, CSB-baseline, and GCSB based on the proportion of non-zero elements distributed uniformly at random across the entire matrix. Figure 4.6 and 4.7 show the speedup of various sparse matrix storage formats (COO, eCSB, CSB-baseline, and GCSB) with respect to CSR on each of TITAN RTX and A100, using the matrices listed in Table 4.4. The x-axis of Figures 4.6 and 4.7 is arranged in ascending order of the proportion of non-zero elements in the sparse matrices. On TITIAN RTX, GCSB outperforms CSR when the proportion of non-zero elements across the entire matrix is above 0.5%. On A100, GCSB surpasses CSR when the proportion of non-zero elements across the entire matrix is above 0.1%. The maximum speedup of GCSB over CSR is 1.59× for random_0.2 on TITAN RTX and 2.8× on A100. When comparing GCSB to CSB-baseline, GCSB performs better in matrices where the proportion of non-zero elements across the entire matrix is higher than CSR. In comparison to eCSB, GCSB is faster on TITAN RTX when the proportion of non-zero elements across the entire matrix is above 5% and on A100 when the proportion is above 20%. It is evident that GCSB exhibits

improved performance when the proportion of non-zero elements across the entire matrix is relatively large, compared to other sparse matrix storage formats.

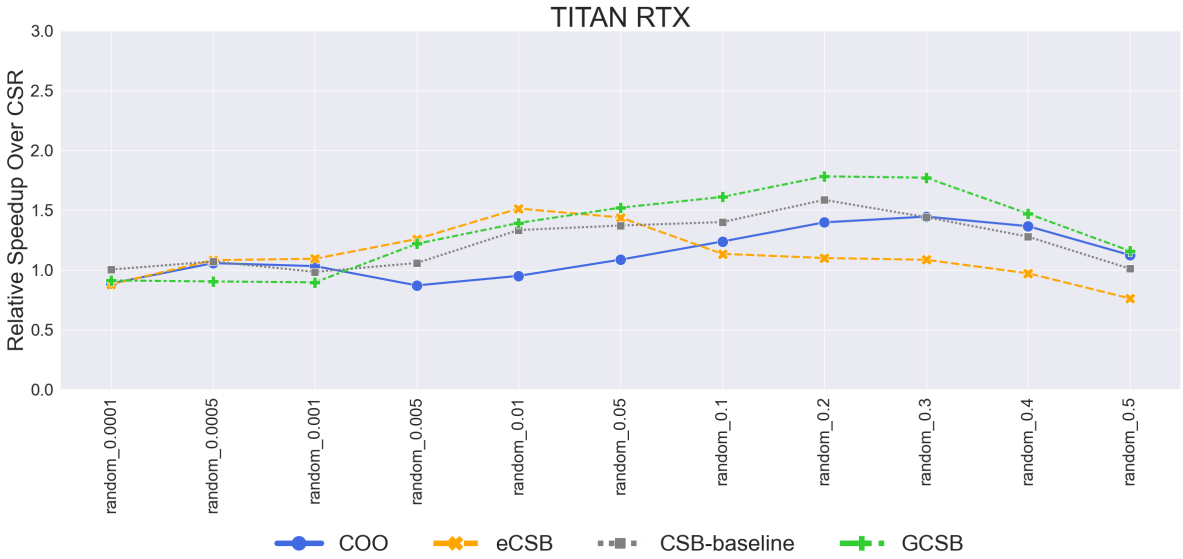



Figure 4.6: Speedup of the total execution time of SpMV and SpMVT for COO, eCSB, CSB-baseline, and GCSB on random sparse matrices compared to CSR on TITAN RTX.

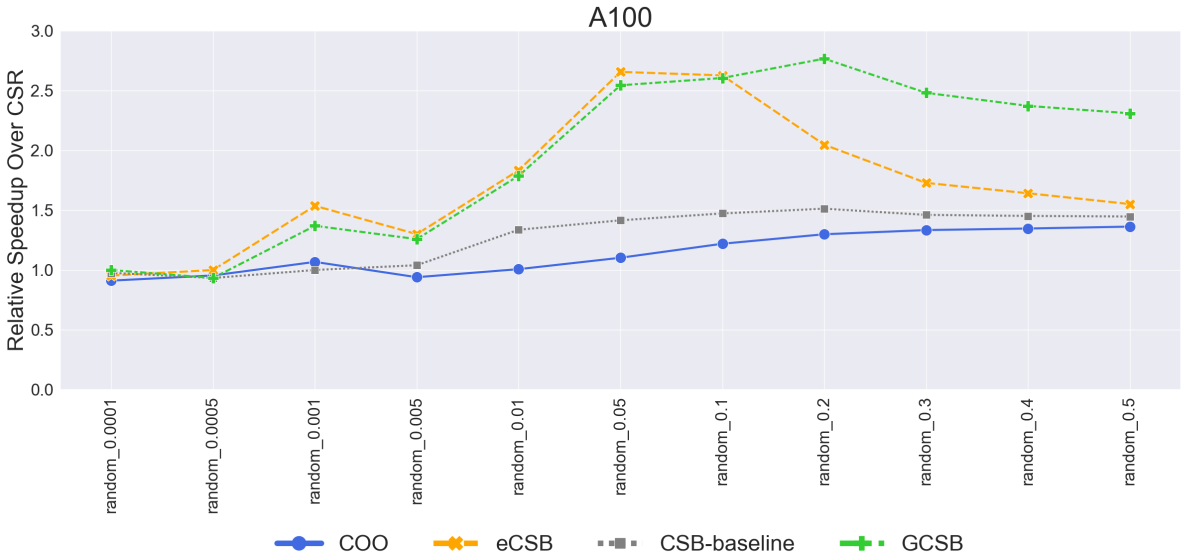



Figure 4.7: Speedup of the total execution time of SpMV and SpMVT for COO, eCSB, CSB-baseline, and GCSB on random sparse matrices compared to CSR on A100.

Figure 4.8 illustrates the proportion of memory usage for COO, eCSB, CSB-baseline, and GCSB with respect to CSR in each sparse matrix for SpMV or SpMVT. When the proportion of non-zero elements in the entire matrix is 0.1% or less, the memory usage of GCSB and CSB-baseline exceeds that of CSR. Even when compared to COO and eCSB, the memory usage of GCSB and CSB-baseline is either greater or similar. This is due to the small number of non-zero elements (nnz) compared to the number of blocks (n_blocks). When the proportion of non-zero elements in the entire matrix is 0.5% or more, the memory usage of GCSB and CSB-baseline becomes smaller than that of COO and eCSB. The proportion of memory usage of GCSB with respect to CSR is 1.09× when the proportion of non-zero elements in the entire matrix is 0.5%, and in matrices with a proportion of non-zero elements greater than 1%, the proportion of memory usage with respect to CSR becomes even smaller. For COO and eCSB, the proportion of memory usage with respect to CSR is 1.48× when the proportion of non-zero elements in the entire matrix is 0.5%, and in matrices with a proportion of non-zero elements greater than 0.5%, the proportion of memory usage with respect to CSR increases further.

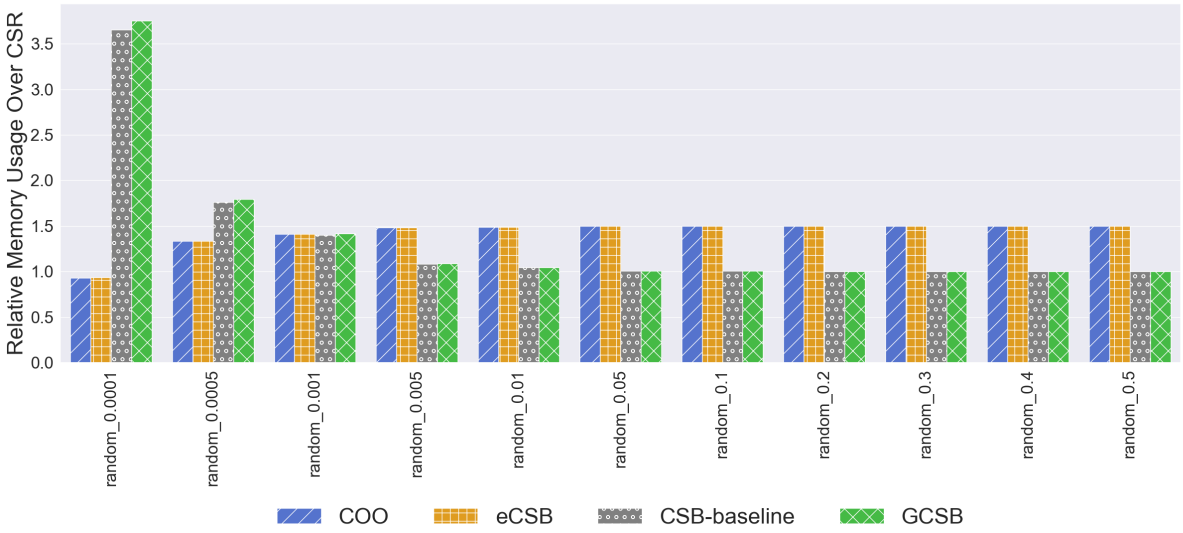



Figure 4.8: Memory usage of SpMV or SpMVT for COO, eCSB, CSB-baseline, and GCSB on random sparse matrices compared to CSR.

From the results of speedup and memory usage relative to CSR in these matrices, it is evident that when the proportion of non-zero elements in the entire matrix is high, GCSB maintains memory usage similar to CSR while achieving higher speedup than CSR. Furthermore, compared to eCSB, it is observed that GCSB exhibits approximately 2/3 of the memory usage while also being faster.

## 4.5 Summary

In this chapter, we evaluated the performance of GCSB through experiments. In experiments with matrices considered ideal for GCSB, we demonstrated that when the block size of GCSB is 32, it achieves faster execution times for the total SpMV and SpMVT while maintaining memory usage equivalent to CSR. Furthermore, under this configuration, GCSB was shown to be faster than CSB-baseline and exhibited reduced L1 cache miss counts.

In experiments utilizing the University of Florida Sparse Matrix Collection, GCSB outperformed CSR in terms of total execution time for SpMV and SpMVT while maintaining memory usage equivalent to CSR for several sparse matrices. It was explained that these sparse matrices share common characteristics, including relatively large matrix sizes and a distribution of non-zero elements throughout the entire matrix.

In experiments using matrices with randomly distributed non-zero elements, it was shown that for matrices where the number of blocks exceeds the number of non-zero elements, the total execution time for SpMV and SpMVT of GCSB can increase, and the required memory usage can exceed not only CSR but also COO and eCSB. On the other hand, for matrices where the number of non-zero elements is sufficiently high compared to the number of blocks, GCSB maintains memory usage similar to CSR while achieving a maximum speedup of up to 2.8× compared to CSR, and it is faster than eCSB, which requires approximately 1.5× the memory usage of GCSB, depending on the matrix.

# Chapter 5

# Conclusions and Future Works

## 5.1 Conclusions

Executing SpMV and SpMVT within the same application with basic sparse matrix storage formats on a GPU poses challenges in terms of memory access efficiency, and an existing method aiming to support both SpMV and SpMVT on GPU lacks sufficient compression.

To tackle these challenges, we redefine CSB for GPU, referred to as CSB-baseline, and introduce GCSB as an extension of CSB-baseline. Our goal with GCSB is to execute SpMV and SpMVT faster than CSR by achieving load balancing and minimizing L1 cache miss counts, while maintaining theoretical memory usage equivalent to that of CSR.

Our experimental results demonstrate that GCSB achieves a significant speedup compared to CSR, with a maximum improvement of 1.47× on TITAN RTX and 2.75× on A100. Remarkably, GCSB accomplishes this while maintaining theoretical memory usage equivalent to CSR for specific matrices from the University of Florida Sparse Matrix Collection. Additionally, GCSB exhibits the ability to reduce L1 cache miss counts compared to CSB-baseline. Furthermore, our qualitative evaluation indicates that GCSB outperforms COO, CSR, and eCSB in scenarios where non-zero elements within a matrix are widely distributed across the entire matrix, the matrix size is sufficiently large, and the proportion of non-zero elements within the matrix is relatively high.

## 5.2 Future works

There are several potential directions to further enhance the performance of GCSB. Firstly, reducing the zero-padding in GCSB is one direction. Varia-

tion in the number of non-zero elements within each block in groups of GCSB result in additional zero-padding, increasing the required memory usage. Introducing methods to minimize zero-padding, similar to the approach used in VCSR [13], may address this challenge effectively.

Secondly, optimizing memory accesses for vectors $x$ and $y$ in $y = Ax$ or $y = A^T x$ is also crucial. Strategies for achieving coalesced memory access to matrix $B$ in Spare Matrix-Matrix Multiplication (SpMM) $C = AB$ have been proposed [25]. Insights from these approaches may offer valuable directions for improving memory access for vectors $x$ and $y$ in GCSB, potentially reducing L1 cache miss counts and achieving coalesced memory access.

Furthermore, investigating methods for efficiently implementing GCSB in scenarios involving SpMM and Spare Matrix-Transpose-Matrix Multiplication (SpMMT) is an intriguing area of research. Sparse matrices encountered in DNNs often exhibit characteristics align with the strengths of GCSB, deviating from the sparse matrices commonly encountered in scientific computations. Specifically, DNN-related sparse matrices tend to have a higher proportion of non-zero elements within the matrix [9]. Additionally, these non-zero elements are distributed in a non-structured manner throughout the matrix. These unique characteristics suggest that GCSB has the potential to excel in terms of performance when dealing with such matrices.

# Bibliography


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

[2] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

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

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

[5] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[6] Mayank Daga and Joseph L Greathouse. Structural agnostic spmv: Adapting csr-adaptive for irregular matrices. In *2015 IEEE 22nd International conference on high performance computing (HiPC)*, pages 64–74. IEEE, 2015.

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

[8] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[9] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[10] Joseph L Greathouse and Mayank Daga. 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*, pages 769–780. IEEE, 2014.

[11] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015.

[12] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. *Advances in neural information processing systems*, 34:21099–21111, 2021.

[13] Elmira Karimi, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. Vcsr: An efficient gpu memory-aware sparse format. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3977–3989, 2022.

[14] Zbigniew Koza, Maciej Matyka, Sebastian Szkoda, and Łukasz Mirosław. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM Journal on Scientific Computing*, 36(2):C219–C239, 2014.

[15] Marcin Krotkiewski and Marcin Dabrowski. Parallel symmetric sparse matrix–vector product on scalar multi-core cpus. *Parallel Computing*, 36(4):181–198, 2010.

[16] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350, 2015.

[17] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of*

*the IEEE/CVF international conference on computer vision*, pages 3296–3305, 2019.

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

[19] GM Morton. A computer oriented geodetic data base, and a new technique in file sequencing. ibm canada ltd. Technical report, mimeo, 1966.

[20] Sreepathi Pai. How the fermi thread block scheduler works. *Retrieved February*, 3:2021, 2014.

[21] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

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

[23] Yuan Tao, Yangdong Deng, Shuai Mu, Zhenzhong Zhang, Mingfa Zhu, Limin Xiao, and Li Ruan. Gpu accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication. *Concurrency and Computation: Practice and Experience*, 27(14):3771–3789, 2015.

[24] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007.

[25] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, pages 672–687. Springer, 2018.

[26] Sai Qian Zhang, Bradley McDanel, and HT Kung. Fast: Dnn training under variable precision block floating point with stochastic rounding. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 846–860. IEEE, 2022.

# Publications

## Domestic Conference Proceedings

(1) Ryohei Izawa, Yasushi Inoguchi, "An Efficient Sparse Matrix Storage Format for Sparse Matrix-Vector Multiplication and Sparse Matrix-Transpose-Vector Multiplication on GPUs", In IEICE Technical Committee on Computer Systems (CPSY), Okinawa, Japan, Dec. 6, 2023

# Acknowledgements

I would like to express my sincere gratitude to Professor Inoguchi for his invaluable guidance and support throughout the course of my research. His insightful advice and encouragement have been instrumental in shaping the direction of my work.

I am also grateful for the insightful discussions I had with Mr. Watanabe, Mr. Iwamura, and Mr. Kaneda from Fujitsu Japan Co., Ltd. Their valuable inputs and perspectives greatly enriched my understanding of the practical implications of my research.

I would like to extend my thanks to all the lab members and colleagues who have contributed to my academic and personal growth. Their wisdom and assistance have been crucial in shaping my academic endeavors.

Lastly, I am deeply grateful to my wife and son for their understanding and patience during this demanding academic journey. Their constant support and encouragement have been a source of strength and motivation for me.