World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:17, No:11, 2023

# Performance Analysis and Optimization for Diagonal Sparse Matrix-Vector Multiplication on Machine Learning Unit

Qiuyu Dai, Haochong Zhang, Xiangrong Liu

*Abstract*—Efficient matrix-vector multiplication with diagonal sparse matrices is pivotal in a multitude of computational domains, ranging from scientific simulations to machine learning workloads. When encoded in the conventional Diagonal (DIA) format, these matrices often induce computational overheads due to extensive zero-padding and non-linear memory accesses, which can hamper the computational throughput, and elevate the usage of precious compute and memory resources beyond necessity. The 'DIA-Adaptive' approach, a methodological enhancement introduced in this paper, confronts these challenges head-on by leveraging the advanced parallel instruction sets embedded within Machine Learning Units (MLUs). This research presents a thorough analysis of the DIA-Adaptive scheme's efficacy in optimizing Sparse Matrix-Vector Multiplication (SpMV) operations. The scope of the evaluation extends to a variety of hardware architectures, examining the repercussions of distinct thread allocation strategies and cluster configurations across multiple storage formats. A dedicated computational kernel, intrinsic to the DIA-Adaptive approach, has been meticulously developed to synchronize with the nuanced performance characteristics of MLUs. Empirical results, derived from rigorous experimentation, reveal that the DIA-Adaptive methodology not only diminishes the performance bottlenecks associated with the DIA format but also exhibits pronounced enhancements in execution speed and resource utilization. The analysis delineates a marked improvement in parallelism, showcasing the DIA-Adaptive scheme's ability to adeptly manage the interplay between storage formats, hardware capabilities, and algorithmic design. The findings suggest that this approach could set a precedent for accelerating SpMV tasks, thereby contributing significantly to the broader domain of high-performance computing and data-intensive applications.

*Keywords*—Adaptive method, DIA, diagonal sparse matrices, MLU, sparse matrix-vector multiplication.

## I. Introduction

A critical challenge in scientific computing and deep learning is the efficient execution of diagonal sparse matrix-vector multiplication (SpMV), and optimizing this process has become a focal point for researchers. With the advent of heterogeneous computing architectures, there is an increasing capability for high-powered computation. Presently, the majority of SpMV algorithms are optimized specifically for the architecture of Graphics Processing Units (GPUs), which have gained prominence as data volumes expand, particularly in deep learning. The field has seen a dramatic rise in demand for computational power due to the rapid advancement of artificial neural networks, and GPUs offer substantial advantages in parallel processing capabilities compared to traditional CPU architectures. However, despite their versatility, GPUs are inherently general-purpose processors, leading to the adoption of specialized multicore processors for certain computations where GPUs may not be the most efficient choice. The challenge with these multicore processors is the potential for load imbalance due to their extensive number of cores, which can lead to performance being bottlenecked by memory bandwidth rather than raw computational power [4].

To mitigate these performance limitations and bottlenecks, a plethora of strategies have been explored by the research community. These include minimizing memory access latency [2], crafting efficient parallel algorithms [12], [14], [19], and integrating high-performance multicore processors [7], [22] to augment SpMV efficiency. The study capitalizes on the advanced capabilities of a state-of-the-art multicore processor, developing a parallel algorithm meticulously optimized for the processor's distinctive hardware architecture and memory hierarchy. The objective is to augment the computation of diagonal sparse matrix-vector multiplication, leveraging the inherent parallelism of the processor to expedite processing and elevate throughput.

Sparse matrix-vector multiplication (SpMV) for diagonal sparse matrices is mathematically represented as:

$$y = Ax \tag{1}$$

where $A \in \mathbb{R}^{m\times n}$ denotes the diagonal sparse matrix, $x \in \mathbb{R}^n$ is the input vector, and $y \in \mathbb{R}^m$ is the resultant output vector. In contrast to general sparse matrices, the non-zero elements of a diagonal sparse matrix are confined to the principal diagonal and its immediate sub- and super-diagonals. This distinctive structure allows for the potential optimization of SpMV computations on hardware specifically designed to exploit such patterns.

The Cambrian MLU is a domain-specific processor specifically designed for artificial intelligence (AI) applications. It has undergone meticulous optimization for AI-centric operations, including convolution, pooling, and activation functions, resulting in enhanced performance and energy efficiency compared to general-purpose computing devices such as GPUs [6], [8].

Designed with specialized data pathways and computational components, the MLU caters to the distinct nature of AI data streams, ensuring their effective access and isolation. This architectural choice, coupled with the software's flexible

X. Liu and Q. Dai are with Xiamen University, Xiamen, China (e-mail: xrliu@xmu.edu.cn, qydai@stu.xmu.edu.cn).
H.Zhang is with Institute of Artificial Intelligence, Hefei Comprehensive National Science Center, Hefei, China (e-mail: solomonz@mail.ustc.edu.cn).

on-chip storage accessibility, facilitates enhanced performance outcomes. The cornerstone of the MLU's architecture is its core—a versatile processing unit equipped with comprehensive computational, I/O, and control functionalities. These cores are capable of operating autonomously or in synergy with fellow MLU cores.

A cluster within the MLU is composed of four such MLU cores, alongside an ancillary memory core and a segment of shared RAM accessible to both the memory core and its quartet of MLU cores. The memory core is tasked with handling data transfers between the Shared RAM and DDR (Double Data Rate) SDRAM but is not designed to execute vector or tensor computations. In this research, the chosen hardware platform is the MLU270 model, the architecture of which is illustrated in Fig. 1.

The abstract model of the system is stratified into five hierarchical levels: server, board, chip, cluster, and MLU core. Each level is architecturally composed of three principal components: an abstract control unit, a computation unit, and a storage unit.

- Level 0, the server level, incorporates a control unit with multiple CPUs, a local DDR storage unit, and a computation unit consisting of several MLU board cards.
- Level 1, the board level, features each MLU board with a local control unit, a DDR memory unit, and an MLU chip that constitutes the computation unit.
- Level 2, the chip level, includes each chip with a local control unit, an L2 Cache as the local storage unit, and a computation unit comprised of one or more clusters.
- Level 3, the cluster level, consists of each cluster with a local control unit, shared storage, and multiple MLU core computation units.
- Level 4, the MLU core level, is where each MLU core contains a local control unit, a private storage unit, and a computation unit capable of both instruction-level and data-level parallelism.

This delineation ensures that at each tier of the model, there is a dedicated balance between control, computation, and storage functionalities, tailored to optimize the overall performance of the machine learning tasks at hand.

The remainder of this paper is structured as follows: Section II reviews the related literature. Section III describes the storage formats and the MLU architecture. The algorithms and their implementation on the MLU are detailed in Section IV. Section V assesses the performance of the kernel. Finally, Section VI provides concluding remarks and outlines future directions for this research.

## II. Related Work

Sparse matrix-vector multiplication (SpMV) stands as a fundamental operation in scientific computing, drawing significant attention for its optimization potential. Researchers have introduced numerous algorithms to accelerate SpMV, recognizing its importance in large-scale computational tasks.

Typically, large sparse matrices undergo compression using specific schemes before operations are carried out. Established methods such as the coordinate (COO) format and the compressed sparse column/row (CSC/CSR) formats are widely used for their efficiency in reducing the computational and storage overhead of sparse matrices [24]. However, certain types of sparse matrices, like diagonal sparse matrices, benefit from specialized storage formats. The diagonal (DIA) format, which stores non-zero elements along the diagonals, is one such example [20]. Another is the Ellpack (ELL) format, optimal for matrices with a relatively uniform distribution of non-zero elements across rows [13]. Tailoring storage structures to align with specific hardware architectures can significantly enhance SpMV performance [1].

Bell and Garland were trailblazers in leveraging the computational prowess of CUDA-based GPUs for SpMV, devising algorithms that seamlessly integrate with diverse memory configurations and capitalize on the inherent architecture of GPUs[5]. They established that the optimal storage format could be matched to the type of sparse matrix in question to extract peak GPU performance. Since their work, a multitude of storage structures and their corresponding kernels have emerged, such as the hybrid ELL/COO (HYB) [5], sliced Ellpack (SELL-C-sigmoid) [18], compressed sparse row 5 (CSR5) [16], and the block format combining CSR and ELL (BCE) [26]. These developments leverage the parallelism inherent in GPU and x86 CPU SIMD architectures to great effect, as extensively discussed in the literature [10].

The Diagonal (DIA) format is adept at storing diagonally sparse matrices by aligning elements along the same diagonal within the same column, which streamlines processing [5], [15], [12]. Nonetheless, the DIA structure can sometimes lead to inefficient space utilization. This inefficiency is particularly evident when diagonals with sparse non-zero elements result in a surplus of zeros in the dense matrix representation. To mitigate this space wastage issue, improved DIA-based storage structures have been developed [3], [23], effectively optimizing space compared to the conventional DIA format.

Xia et al. [11] introduced the DIA-adaptive strategy, enhancing the original DIA kernel by devising two novel storage structures and corresponding kernel functions. These advancements cater to the variegated characteristics of diagonal sparse matrices and are specifically optimized for GPU execution. Their approach demonstrates the dynamic adaptability of sparse matrix storage formats to hardware capabilities, paving the way for more efficient computations.

## III. Diagonal Sparse Matrix Compression and MLU Architecture

In this section, the authors present the DIA-Adaptive model, which is the foundation of the proposed approach. The model incorporates specific optimizations that exploit the distinctive hardware attributes of the MLU to improve computational performance.

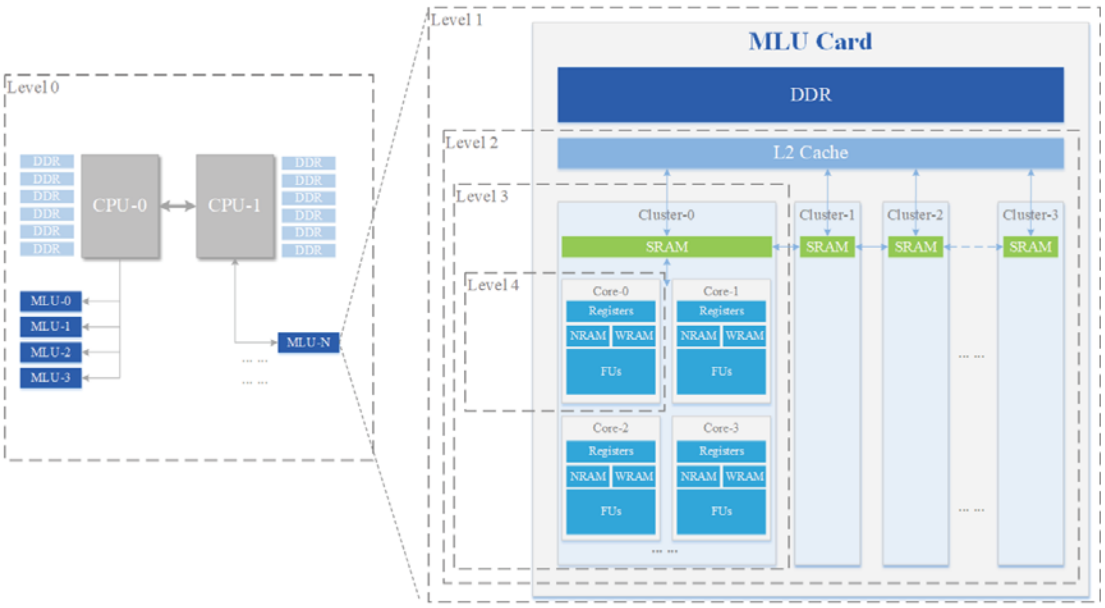

Fig. 1 A MLU architecture

### A. *DIA-Adaptive*

To illustrate the DIA-adaptive and its kernel function, the study assumes two diagonal sparse matrices as follows:

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 & 0 \\ 0 & 4 & 5 & 0 & 0 & 6 \\ 7 & 0 & 8 & 9 & 0 & 0 \\ 0 & 10 & 0 & 11 & 12 & 0 \\ 0 & 0 & 13 & 0 & 14 & 15 \\ 0 & 0 & 0 & 16 & 0 & 17 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & 2 & 0 & 0 & 0 \\ 3 & 4 & 0 & 5 & 0 & 0 \\ 0 & 0 & 6 & 7 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 10 \\ 0 & 0 & 0 & 0 & 0 & 11 \end{pmatrix}$$

*1) DIA Format:* For any given matrix $A$, its representation in the DIA storage format is composed of two distinct components: a matrix *data* and a vector *offset*. The matrix *data* contains the non-zero elements of the original matrix organized by their respective diagonals, while the vector *offset* records the relative position of each diagonal within *data* in relation to the main diagonal.

$$data = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 4 & 5 & 6 \\ 7 & 8 & 9 & 0 \\ 10 & 11 & 12 & 0 \\ 13 & 14 & 15 & 0 \\ 16 & 17 & 0 & 0 \end{pmatrix}, offset = \begin{pmatrix} -2 & 0 & 1 & 4 \end{pmatrix}$$

*2) BRCSD Format:* When non-zero matrix elements are located far from the main diagonal, the DIA storage format tends to introduce excessive zero padding in the *data* array. For matrices with substantial non-zero entries distant from the main diagonal, more space-efficient storage strategies are essential. In response to the limitations of DIA, the Diagonal Compressed Storage based on Row-Blocks (BRCSD) has been introduced, as discussed in [25]. Initially, the diagonal sparse matrix is segmented into row-based blocks, each minimized in size. For instance, matrix $A$ might be segmented into two such blocks. The sparse matrix is then described as:

$$\text{matrix} = \{\text{offset}[0], \text{offset}[1], ..., \text{offset}[n]\}$$

Here, offset$[i]$ denotes the deviation of each diagonal from the main diagonal within the $i$th block. Consequently, matrix $A$ can be represented in a more compact form:

$$A = \left\{ \begin{pmatrix} 0 & 1 & 4 \end{pmatrix}, \begin{pmatrix} -2 & 0 & 1 \end{pmatrix} \right\}$$

Finally, the sparse matrix can be represented by two arrays:

$$brcsd\_offsets = \{r_0|offsets\,[0]\,, ..., r_n|offsets\,[n]\}\,,$$
$$brcsd\_data = \{data\,[0]\,, data\,[1]\,, ..., data\,[n]\}$$

where $r_i$ is the starting row number of the $i$th row piece. The matrix $A$ is represented as follows:

$$brcsd\_offsets = \left\{0|\begin{pmatrix} 0 & 1 & 4 \end{pmatrix}, 2|\begin{pmatrix} -2 & 0 & 1 \end{pmatrix}\right\}$$

$$brcsd\_data = \left\{ \begin{matrix} \begin{pmatrix} 1 & 4 & 2 & 5 & 3 & 6 \end{pmatrix}, \\ \begin{pmatrix} 7 & 10 & 13 & 16 & 8 & 11 \\ 14 & 17 & 9 & 12 & 15 & 0 \end{pmatrix} \end{matrix} \right\}$$

*3) BRCSD-II Format:* Although DIA and BRCSD storage formats can efficiently store a majority of diagonal sparse matrices, they encounter limitations with matrices that exhibit scattered non-zero elements or contain numerous zero entries along the diagonals. These conditions can lead to substantial zero padding within the storage structure. For example, considering a diagonal sparse matrix $B$, its storage using DIA and BRCSD would result in the following representations:

$$data = \begin{pmatrix} 0 & 1 & 0 & 2 \\ 3 & 4 & 0 & 5 \\ 0 & 6 & 7 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 9 & 10 & 0 \\ 0 & 11 & 0 & 0 \end{pmatrix}, offsets = \begin{pmatrix} -1 & 0 & 1 & 2 \end{pmatrix}$$

$$brcsd\_offsets = \left\{0|\begin{pmatrix} -1 & 0 & 1 & 2 \end{pmatrix} \quad 4|\begin{pmatrix} 0 & 1 \end{pmatrix}\right\}$$

$$brcsd\_data = \left\{ \begin{matrix} \begin{pmatrix} 0 & 3 & 0 & 0 & 1 & 4 & 6 & 8 \\ 0 & 0 & 7 & 0 & 2 & 5 & 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 9 & 11 & 10 & 0 \end{pmatrix} \end{matrix} \right\}$$

Despite the utilization of either DIA or BRCSD storage strategies, matrices such as $B$, which contain numerous scattered non-zero elements and extended sequences of zeros, will inevitably result in substantial zero padding. To counter this inefficiency, Xia et al. proposed an enhanced version of BRCSD, termed BRCSD-II. This method initiates with the segregation of the matrix into $data$ and $offsets$ as prescribed by BRCSD. Subsequently, it aligns $nrows$ with the GPU's thread block count. The sparse matrix is then partitioned into row sections, reflective of the sparse matrix's total row count, denoted by $m$. The concluding step involves the consolidation of the $offset$ array. Consequently, the sparse matrix's structure is encapsulated by the subsequent pair of arrays:

$$brcsdII\_offsets = \{p_0|offsets\,[0]\,, ..., p_s|offsets\,[s]\}$$
$$brcsdII\_data = \{data\,[0]\,, data\,[1]\,, ..., data\,[p-1]\}$$

where $p$ is the number of row pieces, $s$ is the size of $offsets$ after accumulating row pieces. The matrix $B$ is represented as follows:

$$brcsdII\_offsets = \left\{1|\begin{pmatrix}-1 & 0 & 2\end{pmatrix}, 2|\begin{pmatrix}0 & 1\end{pmatrix}\right\}$$
$$brcsdII\_data = \left\{\begin{matrix}\begin{pmatrix}0 & 3 & 1 & 4 & 2 & 5\end{pmatrix}, \\ \begin{pmatrix}6 & 8 & 7 & 0\end{pmatrix}, \\ \begin{pmatrix}9 & 11 & 10 & 0\end{pmatrix}\end{matrix}\right\}$$

It can be observed that the number of zeros in BRCSD-II decreases from 13 to 3 and from 11 to 3 compared to DIA and BRCSD, respectively.

### *B. MLU Architecture*

A MLU device is architecturally composed of several subsystems, including a memory subsystem, a Multi-Tensor Processor (MTP) subsystem, and a media subsystem. At the heart of the Cambrian MLU architecture lies the MTP subsystem, which is integral to the device's operation. An MLU chip is typically equipped with an assemblage of hardware including one or more MTP clusters, a PCIe interface, a memory controller, an L2 cache, a media processing unit, and an MLU-Link interconnect.

Each MTP cluster contains multiple Intelligence Processing Unit (IPU) cores and a block of Shared RAM, which collectively form the fundamental execution unit within the MTP architecture. The MTP configuration allows for programmatic compatibility with the Tensor Processor (TP) architecture; when the architectures are aligned, the MTP is capable of executing programs that have been developed for the TP, ensuring binary compatibility.

The TP architecture, often referred to by its codename as a single-core entity, is essentially a hardware ensemble that integrates an IPU core with a dedicated memory system. A TP core is furnished with an Arithmetic Logic Unit (ALU) for scalar computations, a Vector Function Unit (VFU) or Tensor Function Unit (TFU) for artificial intelligence operations, and various Direct Memory Access (DMA) units to facilitate data movement.

To maximize the efficiency of data handling and to exploit the available bandwidth, the TP core is also designed with on-chip Neuron RAM (NRAM) and Weight RAM (WRAM), which are directly coupled with the VFU/TFU. This architecture is meticulously crafted to leverage data locality, thereby enhancing the performance of the MLU device.

The MLU is structured with a hierarchical storage system that spans several layers: General Purpose Registers (GPR), Neuron RAM (NRAM), Weight RAM (WRAM), Shared RAM, L2 Cache, Local DRAM (LDRAM), and Global DRAM (GDRAM). GPR, WRAM, and NRAM serve as the private storage for an individual core. It should be noted that memory cores are not allocated their own WRAM and NRAM resources. The L2 Cache operates as an on-chip, globally shared memory, predominantly utilized for caching instructions, kernel parameters, and read-only data.

LDRAM is designated as the private storage for each MLU and memory core, boasting greater capacity than that of WRAM and NRAM. This level is often employed to mitigate on-chip storage limitations. Conversely, GDRAM is a globally shared storage, facilitating data interchange both between host and device and among computing tasks.

The MLU empowers software applications with the ability to meticulously manage data transfers across these diverse storage tiers. The associated compiler is responsible for providing robust address space declarations for software higher in the stack, in addition to a plethora of mechanisms and interfaces designed for both explicit and implicit data movements. Such features grant users the capability to precisely orchestrate data transit between storage levels, optimizing the interplay between computation and I/O to enhance overall computational efficacy.

At the heart of MLU's design is its support for parallel processing across seven distinct levels: server, board, chip, cluster, core, pipeline, and data. Server-level and board-level parallelism are contingent upon the particular system configuration. In contrast, chip-level, cluster-level, and core-level parallelisms are determined by user-defined task dimensions and classifications on the host side. Similarly, pipeline-level and data-level parallelism within each core are subject to user programming on the device side.

On the device, the primary unit of user programming is termed a 'task.' Each task is confined to execution on a single core during any given run, with no task-switching permitted mid-execution. Within a cluster, multiple tasks can proceed concurrently, and the support for the number of clusters is variable across chips. Every core operates as a processing unit, adept at managing streaming, scalar, and vector operations. Scalar and control flow instructions primarily facilitate control functionalities, while vector instructions are harnessed for parallel data processing, with the capability to handle data of variable lengths.

MLU cores are outfitted with a quartet of instruction pipelines that facilitate a multiplexed processing environment, optimizing for efficiency and performance. The delineation of these pipelines is as follows:

- *IO Stream*: Dedicated to orchestrating the input/output operations involving off-chip DDR memory, the IO stream is optimized for managing the extensive reads and writes that span beyond the on-chip environment.

- *Move Stream*: This stream is specifically allocated for handling memory access instructions that are contained within the on-chip boundaries, excluding those that involve off-chip DDR transfers.
- *Compute Stream*: Serving as the backbone for intense computational tasks, the compute stream is tasked with executing tensor and vector calculations alongside scalar instructions that cater to such operations.
- *Scalar Stream*: The scalar stream is responsible for the execution of all scalar operations, which typically involve non-vectorized calculations that are executed one at a time.

A notable feature of these instruction pipelines is their ability to operate in tandem. By design, the IO, move, compute, and scalar streams execute concurrently, providing a seamless parallel processing capability. To maintain computational correctness and synchronization, the hardware is architected with mechanisms to ensure register dependencies are respected across the streams. This is exemplified in scenarios where an instruction from the IO, move, or compute stream modifies a universal scalar register; the hardware automatically institutes a sequential execution order, mandating that any dependent instruction within the scalar stream is postponed until the register modification is finalized. Conversely, in the event that a scalar stream instruction modifies the state of a register, any dependent instructions from other streams are required to pause until the completion of the scalar write operation. This synchronization mechanism upholds the integrity of register state and guarantees that all subsequent instructions accessing that register receive the updated value, thereby mitigating data hazards and ensuring consistency in computational flow.

## IV. Parallel Algorithm of Diagonal SpMV and Its MLU Implementation

In this section, the proposed methodology for optimizing SpMV on diagonal sparse matrices utilizing the MLU hardware is detailed. The approach is designed to capitalize on the unique architectural features of the MLU, aiming to surpass the efficiency and performance benchmarks set by extant techniques. The ensuing discourse will meticulously expound upon the methodological framework and delineate the superiority of this innovative approach.

### A. DIA Kernel

Parallelization of Sparse Matrix-Vector Multiplication (SpMV) on the Machine Learning Unit (MLU) for diagonal matrices employing the DIA format follows a straightforward approach: each task is assigned to process an individual row. Algorithm 1 delineates the core steps of the DIA kernel operation. Preliminary data movement operations transfer $x$, $data$, and $offsets$ from the DDR to NRAM, which is posited to hasten each memory access during computation. Furthermore, the contiguous nature of memory accesses to $data$ and $x$ contributes to the optimization of performance for the DIA kernel.

**Algorithm 1:** The kernel function of DIA for SpMV

```
Data: The known vector $x$, the number of rows $num\_rows$, cols $num\_cols$, and diagonals
      $num\_diags$ of the matrix and the arrays in DIA format($data$, $offsets$)
Result: The output vector $y$
1  $row = taskId$;
2  if $row \leq num\_rows$ then
3      $sum = 0.0$;
4      $NRAM\ int\ offset\_n[num\_diags]$;
5      $NRAM\ float\ data\_n[num\_diags]$;
6      $NRAM\ float\ x\_n[num\_cols]$;
7      memory copy from $offsets$ to $offset\_n$;
8      memory copy from $data$ to $data\_n$ with stride $num\_rows$;
9      memcpy copy from $x$ to $x\_n$;
10     for $i = 0\ to\ num\_diags\ with\ i = i + 1$ do
11         $uint32\_t\ col = row + offset\_n[i]$;
12         $float\ val = data\_n[num\_rows * i + row]$;
13         if $col \geq 0\ and\ col \leq num\_cols$ then
14             $sum += val * x\_n[col]$;
15         end
16     end
17     $y[row] = sum$;
18 end
```

### B. BRCSD Kernel

The parallel execution strategy for SpMV utilizing the Block Row Compressed Sparse Diagonal (BRCSD) format on MLUs is straightforward, wherein each computational task is dedicated to processing a discrete row piece. Nevertheless, given that the row piece sizes can vary, this approach inherently leads to an imbalance in the distribution of computational workloads across tasks. Such imbalance is particularly notable when handling matrices like $B$, where the quantity of rows within the row pieces may differ substantially.

To mitigate workload disparities, the proposed algorithm specifies a maximum value, $n$, representing the upper limit of rows a single task is designated to process. If the size of a row piece, $t$, exceeds $n$, the row piece is subdivided into $\left\lceil \frac{t}{n} \right\rceil$ smaller row pieces, thereby standardizing the row count for each task to a maximum of $n$. For instance, if the authors set $n$ to 2 in the context of matrix $B$, the algorithm divides the matrix into two initial row pieces, with the second row piece subsequently being partitioned into smaller segments. As a result, each task is aligned to process three rows, as depicted in Algorithm 2.

Within this kernel implementation, two specialized instructions are employed: `bang_mul` and `bang_reduce_sum`. The `bang_mul` instruction executes element-wise multiplication across vector pairs, recording the output in a designated result vector, while the `bang_reduce_sum` instruction aggregates the elements within a vector, with the sum being assigned to a temporary variable, tmp. The count of elements subjected to each operation is predetermined by the last parameter in both

**Algorithm 2:** The kernel function of BRCSD for SpMV

```
Data: The known vector $x$, the number of rows
      $num\_rows$, cols $num\_cols$, and diagonals
      $n\_diags$ of the matrix and the arrays in BRCSD
      format($brcsd\_data$, $brcsd\_offsets$)
Result: The output vector $y$
$local\_id = taskId$;
$offset\_id = taskId\ < 1\ ?\ 0\ :\ 1$;
$NRAM\ float\ tmp = 0.0$;
$NRAM\ float\ data\_n[n\_diags]$;
$NRAM\ float\ x\_n[n\_diags]$;
$NRAM\ float\ result[n\_diags]$;
switch $offset\_id$ do
    case $0$ do
        memory copy from $brcsd\_data$ starting from
          $local\_id + 2$ to $data\_n$ in strides of 2;
        memory copy from $x$ starting from $local\_id$ to
          $x\_n$ in strides of 3;
        $bang\_mul(result,\ data\_n,\ x\_n,\ 3)$;
        $bang\_reduce\_sum(\&tmp,\ result,\ 3)$;
        $y[local\_id] = tmp$;
    end
    case $1$ do
        memory copy from $brcsd\_data$ starting from
          $local\_id + 4$ to $data\_n$ in strides of 4;
        memory copy from $x$ starting from $local_id - 2$
          to $x\_n$ in strides of 3;
        $bang\_mul(result,\ data\_n,\ x\_n,\ 3)$;
        $bang\_reduce\_sum(\&tmp,\ result,\ 3)$;
        $y[local\_id + coreId]\ =\ tmp$;
    end
end
```

instructions.

*C. BRCSD-II Kernel*

In the context of SpMV on MLUs, employing the BRCSD-II format facilitates intuitive parallelization. Each core is assigned to handle a distinct row piece, and within that core, individual tasks are mapped to specific rows. Algorithm 3 delineates this approach.

To commence the process, the absolute core identifier, $core_id$, is computed. Considering the MLU's architecture consists of four clusters, each with four cores, the calculation of $core_id$ is achieved by the expression $clusterId \times 4 + coreId$. Subsequently, the $local_id$ is derived from $taskId$, representing the absolute thread ID within a core. Here, $core_id$ determines the row piece index for matrix processing, and $local_id$ pinpoints the precise row within that row piece.

Prior to executing SpMV, it is crucial to transfer the $offset$ array into NRAM, streamlining direct access during the computation phase. The next step involves discerning the specific segment of $offset$ needed to retrieve corresponding values of vector $x$ for the given $core_id$. The multiplication and addition operations are then performed using MLU's vector processing capabilities, with interim results stored in Shared RAM (SRAM) instead of directly in the result vector $y$, which resides in DDR memory. This intermediate storage strategy in SRAM—accessible by all cores—minimizes memory bandwidth wastage, as SRAM offers significantly faster access speeds than DDR.

Upon completion of these computations and ensuring synchronization across all cores, the data stored in SRAM are collectively written to DDR. This final step can be executed efficiently with a single vector operation instruction, showcasing an optimization that leverages the shared nature of SRAM for enhanced performance of the kernel.

**Algorithm 3:** The kernel function of BRCSD-II for SpMV

```
Data: The known vector $x$, the number of rows
      $num\_rows$, cols $num\_cols$, and diagonals
      $n\_diags$ of the matrix and the arrays in BRCSD
      format($brcsd\_data$, $brcsdII\_offsets$)
Result: The output vector $y$
$core\_id = coreId$;
$local\_id = taskId$;
$offset\_id = taskId\ < 1\ ?\ 1\ :\ 2$;
$NRAM\ float\ tmp = 0.0$;
$NRAM\ float\ data\_n[n\_diags]$;
$NRAM\ float\ x\_n[n\_diags]$;
$NRAM\ float\ result[n\_diags]$;
$SRAM\ float\ y\_n[n\_diags]$;
$NRAM\ uint32\_t\ offset\_n[n\_diags][n\_diags]$;
memory copy from $brcsdII\_offsets$ to $offset\_n$;
switch $offset\_id$ do
    case $1$ do
        memory copy from $brcsdII\_data$ starting from
          $core\_id * 3 + local\_id$ to $data\_n$ in strides of
          2;
        memory copy from $x$ starting from
          $offset\_n[offset\_id]$ to $x\_n$;
        $bang\_mul(result,\ data\_n,\ x\_n,\ 3)$;
        $bang\_reduce\_sum(\&tmp,\ result,\ 3)$;
        $y\_n[local\_id + core\_id * 2] = tmp$;
    end
    case $2$ do
        memory copy from $brcsdII\_data$ starting from
          $(core\_id - 1) * 2 * 2 + local\_id + 3 * 2 * 1$ to
          $data\_n$ in strides of 2;
        memory copy from $x$ starting from
          $offset\_n[offset\_id]$ to $x\_n$;
        $bang\_mul(result,\ data\_n,\ x\_n,\ 2)$;
        $bang\_reduce\_sum(\&tmp,\ result,\ 2)$;
        $y\_n[local\_id + coreId * 2]\ =\ tmp$;
    end
end
synchronize all cores;
memory copy from $y\_n$ to $y$;
```

### D. MLU Implementation

In this subsection, the implementation of the algorithm on MLUs is elucidated, given that their architecture and programming paradigms diverge from those associated with general-purpose Graphics Processing Units (GPUs). The hardware platform for the conducted experiments is the MLU270, detailed in Fig. 1.

During experimentation, the default configuration for the MLU is employed, comprising four clusters, each housing four cores. These clusters share a singular main memory, which typically houses data transferred from the Central Processing Unit (CPU). Each cluster contains a Shared Random Access Memory (RAM) utilized by its cores, along with private Near RAM (NRAM) and Wide RAM (WRAM) for each core. Similar to a CPU, each core possesses its own register set; however, these are not programmable. Direct programming is feasible only for memory structures above the NRAM and WRAM levels. The execution of a kernel necessitates initial data and basic MLU configuration transfer to the MLU, followed by a startup function invocation from the CPU to activate the MLU operations. For efficiency, data that are frequently accessed are first relocated from Double Data Rate (DDR) memory to the NRAM within each core, with operational results being temporarily stored in the Shared RAM accessible to all four cores, thereby hastening data retrieval and enhancing performance.

Conventional practice entails the MLU assigning default data to each thread's stack, where access is expedient, yet limitations emerge due to non-shared data across threads and the stack's limited capacity. This proves inadequate for high-performance SpMV tasks. Hence, commonly used data are stored in NRAM while operational outcomes are placed in Shared RAM, facilitating data sharing within core threads and ensuring results from the same cluster are collectively stored. This strategy circumvents excessive delays during data reads and writes. Subsequent to cluster computation, core synchronization is imperative before collective result writing to the DDR, further diminishing the time penalties associated with MLU bus transfers. Concurrent cluster operations amplify the kernel's parallelism. Additional optimization techniques, such as loop unrolling, are applied during kernel compilation and execution to enhance performance.

## V. Evaluation and Analysis

The experimental evaluation pursues dual objectives: firstly, to benchmark the performance of the proposed algorithm against a suite of contemporary SpMV kernels executed on GPUs; and secondly, to assess the performance enhancement derived from memory-level optimizations on MLUs in comparison with their non-optimized counterparts.

Table I delineates the specifications of the MLU270 utilized in the study. The performance metric is the kernel runtime, gauged from the moment the program invokes the functions on the MLU side to the point when control is relinquished back to the CPU. Since the data are pre-loaded onto the MLU's DDR memory prior to computation, the runtime excludes any write-back duration for the results, thus equating the MLU's computation time with the total runtime. The sparse diagonal matrices for this study originate from the University of Florida Sparse Matrix Collection [9]. Table II compiles details of the diagonal sparse matrices evaluated, encompassing dimensions, the count of diagonals with nonzero entities, and the aggregate of nonzero elements.

TABLE I
Experiment Environment

| Processor | MLU270-S4 |
|---|---|
| Architecture | MLUv02 |
| INT16 peak/TOPS | 64 |
| Precision Support | INT16, INT8, FP32, FP16 |
| Memory | 16GB DDR4, ECC |
| Bandwidth | 102GB/s |
| Interface | x16 PCIe Gen.3 |
| Bitwidth | 256-bit |

TABLE II
Descriptions of Test Matrices

| Matrix | Dimension | Diagonals | nonzeros |
|---|---|---|---|
| wang3 | 26,064×26,064 | 21 | 177,168 |
| wang4 | 26,068×26,068 | 23 | 177,196 |
| s3dkt3m2 | 90,449×90,449 | 655 | 3,686,223 |
| s3dkq4m2 | 90,449×90,449 | 661 | 4,427,725 |
| kim1 | 38,415×38,415 | 25 | 933,195 |
| kim2 | 456,976×456,976 | 25 | 11,300,020 |
| nemeth21 | 9,506×9,506 | 169 | 1,173,746 |
| nemeth22 | 9,506×9,506 | 197 | 1,358,832 |
| af_1_k101 | 503,625×503,625 | 897 | 17,550,675 |
| af_2_k101 | 503,625×503,625 | 897 | 17,550,675 |
| crystk02 | 13,965×13,965 | 99 | 968,583 |
| crystk03 | 24,696×24,696 | 99 | 1,751,178 |
| pde225 | 225×225 | 5 | 1,065 |
| pde900 | 900×900 | 5 | 4,380 |
| pde2961 | 2,961×2,961 | 5 | 14,585 |

For clarity and to reduce computational complexity, subsequent experiments were conducted using single-precision data.

### A. Experimental Analysis

In this section, the deployment of the kernel on the MLU is delineated, along with the classification methodology for

TABLE III
Optimal Kernels for Test Matrices

| Matrix | Optimal kernel |
|---|---|
| wang3 | BRCSD-II |
| wang4 | BRCSD-II |
| s3dkt3m2 | BRCSD-II |
| s3dkq4m2 | BRCSD-II |
| kim1 | BRCSD-II |
| kim2 | BRCSD-II |
| nemeth21 | BRCSD-II |
| nemeth22 | BRCSD-II |
| af_1_k101 | BRCSD-II |
| af_2_k101 | BRCSD-II |
| crystk02 | BRCSD-II |
| crystk03 | BRCSD-II |
| pde225 | BRCSD-I |
| pde900 | BRCSD-I |
| pde2961 | BRCSD-I |

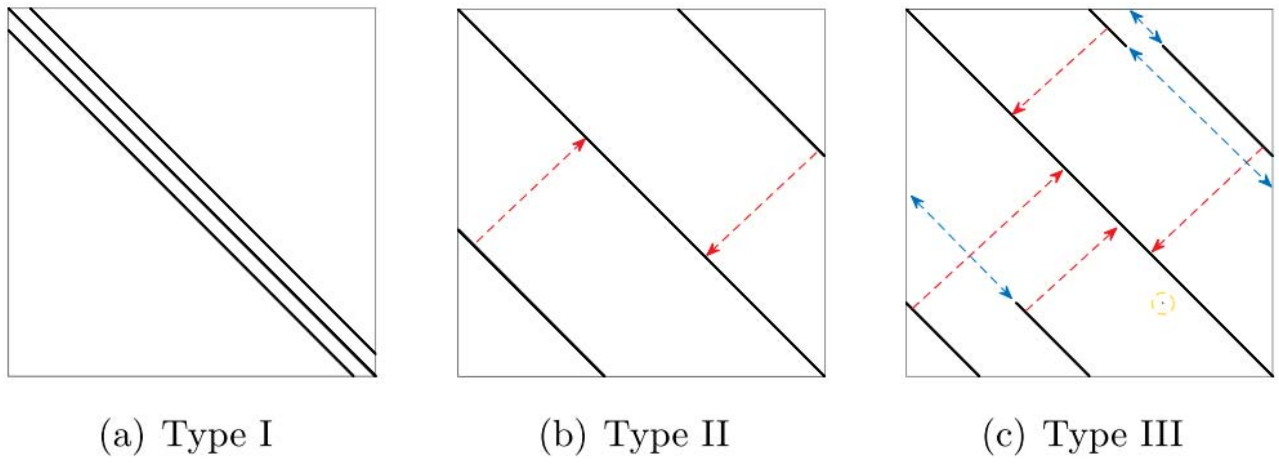

Fig. 2 Three types of diagonal sparse matrices

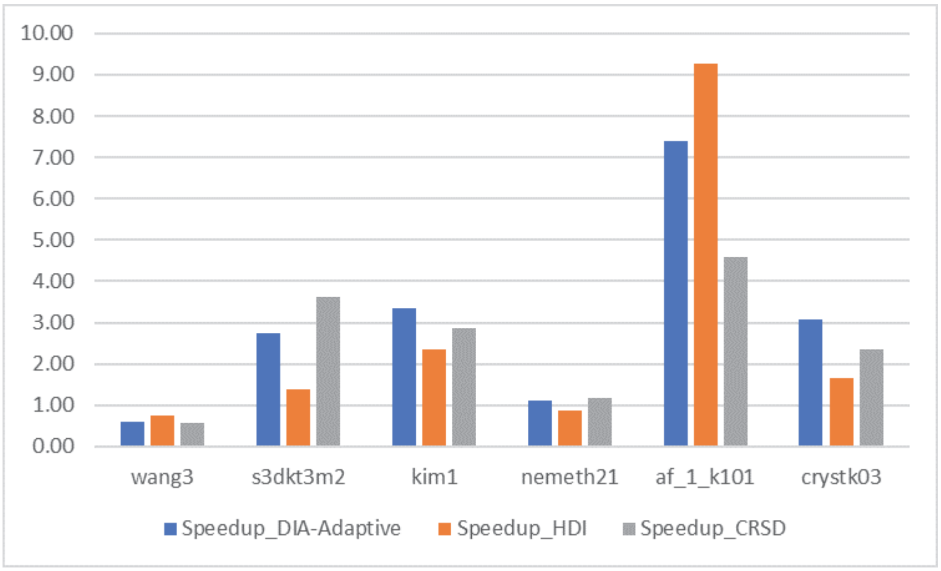

Fig. 3 Speedup of CRSD, HDI, and DIA-Adaptive on MLU versus GPU

test matrices. These matrices were sorted into three distinct categories based on structural characteristics. Correspondingly, discrete kernels were applied to each category, with the most efficacious kernel for each detailed in Table III. A comparative performance evaluation of these kernels on both the MLU and GPUs indicates that the MLU kernels, for the majority of cases, exhibit superior performance over both the GPU kernels and the extant state-of-the-art alternatives for GPUs. Conclusively, a comparative performance analysis of four algorithms—CRSD, HDI, DIA, and the novel DIA-Adaptive—is presented. The DIA-Adaptive algorithm demonstrates consistent superiority over the other algorithms when applied to identical test matrices.

In this study, the research delineated three categories of diagonal sparse matrices, as illustrated in Fig. 2 [25]. These categories are defined by their structural attributes and the pattern in which their non-zero elements are distributed:

- *Type I matrices* are characterized by dense diagonal elements predominantly clustered around the main diagonal. These can be efficiently stored in the conventional DIA format, as their non-zero elements' proximity to the main diagonal minimizes the necessity for zero padding.
- *Type II matrices* exhibit dense diagonal elements situated on off-center diagonals with shorter spans. For such matrices, the Block Row Compressed Sparse Diagonal (BRCSD) format is the optimal storage solution, given that the DIA format would lead to an excess of zero padding, thereby increasing storage requirements without a corresponding benefit. Alternatively, the BRCSD-II format can also be applied to Type II matrices, despite its higher complexity.
- *Type III matrices* are composed of elements from Type II as well as zero elements along their diagonals, in conjunction with additional isolated non-zero entries. To store Type III matrices effectively and reduce

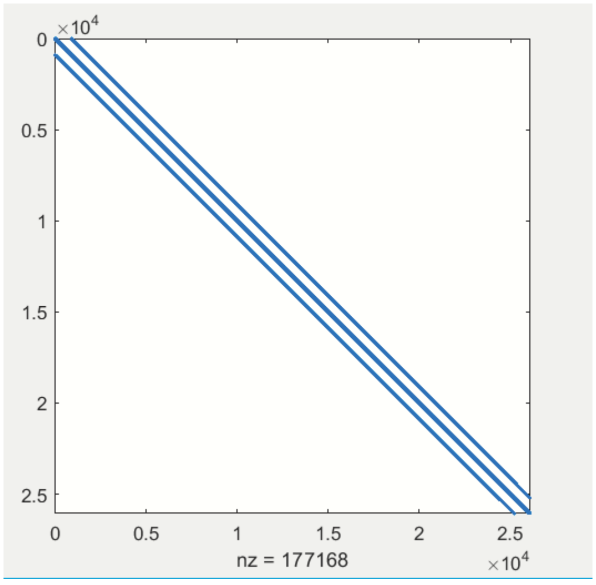

Fig. 4 Structure of wang3

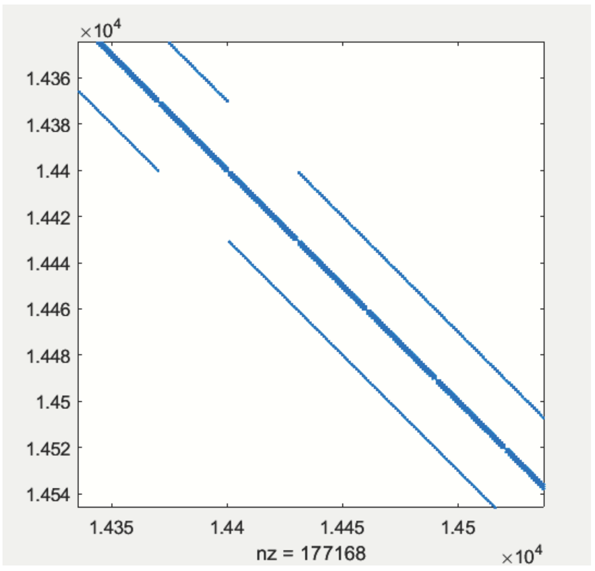

Fig. 5 Structure of wang3 after scaling

zero padding, the BRCSD-II format is necessary, accommodating the sparse nature of these matrices with enhanced storage efficiency.

By categorizing matrices based on these types and choosing the appropriate storage formats, the research aims to optimize the storage and subsequent computational operations on these sparse structures.

In the conducted experiments, the selection of 16,384, 4,096, and 1,024 threads for each of the four clusters on the MLU was made in consideration of the operational efficiency tied to the number of threads. Fig. 7 demonstrates that kernel performance on the MLU is enhanced with a decrease in the number of threads per core, notably when the count is reduced from 16,384 to 4,096. Contrary to expectations, diminishing the thread count further to 1,024 does not yield a marked improvement. This deviation from theoretical predictions, where a higher number of threads is typically associated with improved performance, stems from the distinct hardware traits and optimization tactics of the MLU's software stack as opposed to traditional GPU architectures.

The optimal number of threads for kernels on the MLU is capped due to the nature of its parallel processing, which is predominantly core-dependent, as opposed to the thread-reliant parallelism on GPUs. On the MLU, each core is designed to undertake parallel tasks more autonomously, and as a result, the cores' computational prowess is more pronounced than that of GPU threads. This delineates the necessity for a tailored approach to thread allocation on MLUs, ensuring that the inherent hardware and software configurations are aptly leveraged for maximal efficiency.

## B. Performance Evaluation

*1) MLU Acceleration for CRSD, HDI, and DIA-Adaptive:* Fig. 3 displays the comparative performance enhancements of CRSD, HDI, and DIA-Adaptive formats when executed on MLUs and GPUs. The vertical axis quantifies the performance speedup. On the MLU, the CRSD, HDI, and DIA-Adaptive formats report average performance speedups of 2.53-fold, 2.71-fold, and 3.05-fold, respectively.

The data underscore the pronounced impact of the MLU on enhancing computational speed. Yet, it is noteworthy that the speedup for the matrix labeled 'wang3' is relatively modest. Analysis of 'wang3'—depicted in Fig. 4—and its scaled structure in Fig. 5, identifies it as a type III matrix. The modest performance speedup for 'wang3' on the MLU may be due to the significant offsets incurred during its conversion to the BRCSD-II format. As 'wang3' exhibits characteristics akin to a type I matrix, its conversion to BRCSD-II format results in fragmentation into numerous smaller blocks. This fragmentation potentially undermines the efficiency of BRCSD-II kernel optimizations due to the memory hierarchy's structure. Consequently, this leads to an increased frequency of data transfers from the DDR to the NRAM, adversely affecting performance.

Subsequent experimentation has corroborated that Sparse Matrix-Vector Multiplication (SpMV) operations utilizing CRSD, HDI, and DIA-Adaptive formats benefit from acceleration when deployed on an MLU.

*2) Performance of DIA-Adaptive:* This section of the study focuses on the comparative analysis of the computational times for SpMV on the MLU utilizing CRSD, HDI, and DIA-Adaptive kernels. The findings indicate that the DIA-Adaptive kernel outshines the others, delivering the most efficient performance in the majority of test scenarios. Illustrated in Fig. 6, there is a discernible performance boost with the DIA-Adaptive approach compared to SpMV operations using CRSD, HDI, and traditional DIA formats on the MLU.

Further into the experiments, the researchers ascertained that the DIA-Adaptive format for SpMV significantly exceeds the CRSD, HDI, and DIA formats in performance metrics. Specifically, DIA-Adaptive showcased performance enhancements of 35.21%, 37.36%, and 69.69% over the

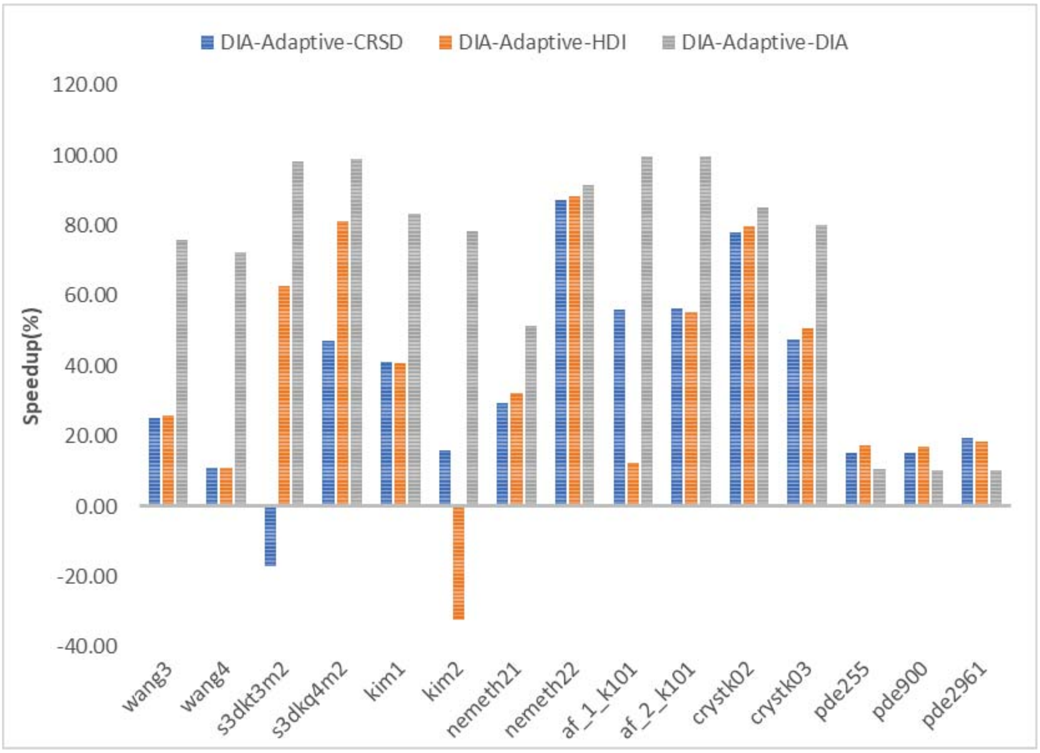

Fig. 6 The performance improvement of DIA-Adaptive

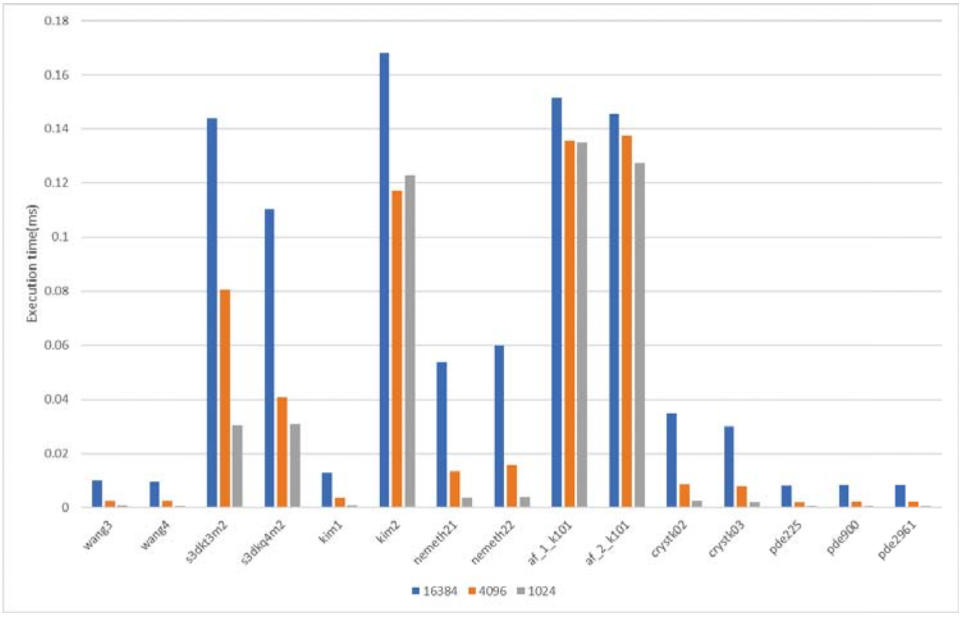

Fig. 7 The performance of SpMV with different threads

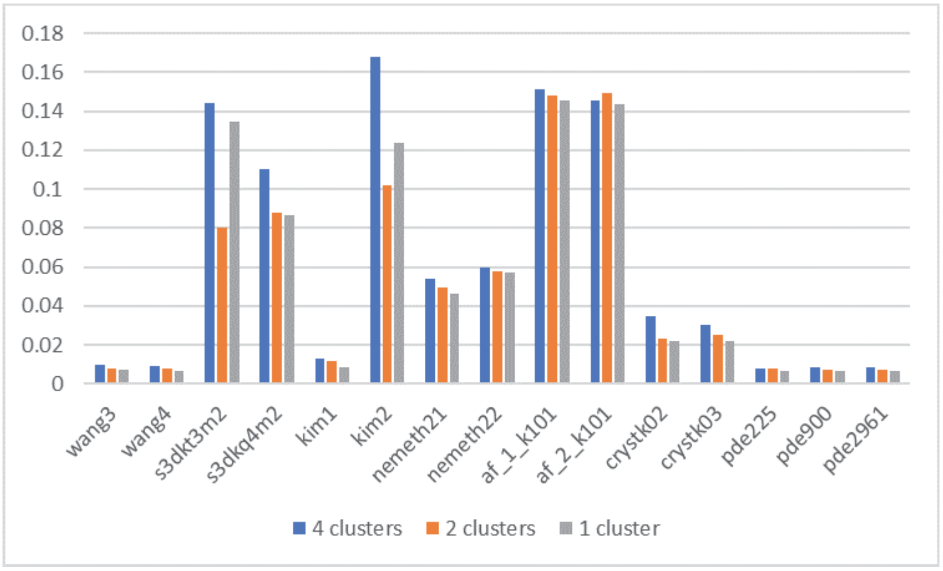

Fig. 8 The performance of SpMV with different clusters

other three formats, respectively. These statistics reinforce the efficacy of the DIA-Adaptive format in optimizing SpMV operations on MLUs, affirming its superiority in accelerating computational tasks in this domain.

*3) Performance of Different Threads:* In preparation for kernel execution, the MLU's configuration must be adjusted to define the thread allocation, similar to the process for GPUs. The MLU270, by default, employs four clusters to carry out parallel computations, and each cluster is comprised of four cores, which also conduct parallel processing. Task execution on an MLU mandates the delineation of the requisite cluster and thread count. Within this context, the investigators set out to scrutinize the effects of varying cluster and thread counts on the efficacy of the DIA-Adaptive format.

The influence of the thread and cluster quantities on the SpMV performance using the DIA-Adaptive approach is delineated through the findings exhibited in Figs. 7 and 8. The data from Fig. 7 indicate a correlation between the number of threads and the DIA-Adaptive kernel's performance within a uniform configuration of four clusters. The trend that emerges from the figure suggests that kernel performance is inversely proportional to thread count across almost all evaluated sparse matrices. A plausible explanation for this phenomenon is that fewer threads may diminish the overhead associated with thread switching, thereby enhancing overall performance.

Furthermore, Fig. 8 delineates the DIA-Adaptive kernel's performance subject to varying cluster counts, while maintaining a consistent thread configuration. Echoing the observations from Fig. 7, there is an apparent trend where performance gains are more pronounced with a reduced number of clusters. This underscores the significance of optimal cluster and thread configuration in achieving maximal performance from the MLU, particularly when deploying the DIA-Adaptive kernel for SpMV computations.

## VI. Conclusion

In the presented study, the authors unveil the DIA-Adaptive scheme alongside its bespoke kernel tailored for MLUs, emphasizing the employment of vector instruction sets to refine the parallel execution efficiency of single-precision SpMV. The investigative focus then shifts to assessing the enhancements proffered by the DIA-Adaptive scheme and kernel in comparison to other storage frameworks, examining their performance across GPU and MLU architectures. Additionally, the study delves into how the modulation of thread and cluster numbers influences overall performance metrics.

Looking ahead, the researchers are poised to extend their inquiry into the dynamics of additional varieties of diagonal sparse matrices and the ramifications of utilizing multi-MLU environments on kernel performance. This prospective research promises to not only broaden the understanding of performance scaling across various hardware platforms but also to advance the optimization of SpMV operations, which are crucial in high-performance computing applications.

## References

[1] Abubaker, N., et al. (2018). "Spatiotemporal graph and hypergraph partitioning models for sparse matrix-vector multiplication on many-core architectures." IEEE Transactions on Parallel and Distributed Systems 30(2): 445-458.
[2] Aleksei, S., et al. (2022). "Comparing the performance of general matrix multiplication routine on heterogeneous computing systems." Journal of Parallel and Distributed Computing 160.
[3] Barbieri, D., et al. (2015). "Three storage formats for sparse matrices on GPGPUs."
[4] Beaumont, O., et al. (2019). "Recent Advances in Matrix Partitioning for Parallel Computing on Heterogeneous Platforms." IEEE Transactions on Parallel and Distributed Systems 30(1).
[5] Bell, N. and M. Garland (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. Proceedings of the conference on high performance computing networking, storage and analysis.
[6] Chen, T., et al. (2014). "DianNao." ACM SIGARCH Computer Architecture News 42(1).

[7] Chen, Y., et al. (2019). ”Performance-Aware Model for Sparse Matrix-Matrix Multiplication on the Sunway TaihuLight Supercomputer.” IEEE Transactions on Parallel and Distributed Systems 30(4).

[8] Chen, Y., et al. (2014). ”DaDianNao.” Microarchitecture.

[9] Davis, T. A. and Y. Hu (2011). ”The University of Florida sparse matrix collection.” ACM Transactions on Mathematical Software (TOMS) 38(1): 1-25.

[10] Filippone, S., et al. (2017). ”Sparse matrix-vector multiplication on GPGPUs.” ACM Transactions on Mathematical Software (TOMS) 43(4): 1-49.

[11] Gao, J., et al. (2021). ”Adaptive diagonal sparse matrix-vector multiplication on GPU.” Journal of Parallel and Distributed Computing 157: 287-302.

[12] Gao, J., et al. (2017). ”A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm.” Parallel Computing 63: 1-16.

[13] Im, E.-J., et al. (2004). ”Sparsity: Optimization framework for sparse matrix kernels.” The International Journal of High Performance Computing Applications 18(1): 135-158.

[14] Kunchum, R., et al. (2017). ”On improving performance of sparse matrix-matrix multiplication on GPUs.” Supercomputing.

[15] Li, K., et al. (2014). ”Performance analysis and optimization for SpMV on GPU using probabilistic modeling.” IEEE Transactions on Parallel and Distributed Systems 26(1): 196-205.

[16] Liu, W. and B. Vinter (2015). CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. Proceedings of the 29th ACM on International Conference on Supercomputing.

[17] Ma, S., et al. (2019). ”Coordinated DMA: Improving the DRAM Access Efficiency for Matrix Multiplication.” IEEE Trans. Parallel Distrib. Syst. 30(10).

[18] Monakov, A., et al. (2010). Automatically tuning sparse matrix-vector multiplication for GPU architectures. International Conference on High-Performance Embedded Architectures and Compilers, Springer.

[19] Ruoxi, W., et al. (2021). ”PBBFMM3D: a parallel black-box algorithm for kernel matrix-vector multiplication.” Journal of Parallel and Distributed Computing(prepublish).

[20] Saad, Y. (1990). SPARSKIT: A basic tool kit for sparse matrix computations.

[21] Cambrican[Image]. MLU Server Hierarchy. https://www.cambricon.com/docs/bangc/developer_guide_html/_images/4.1.png

[22] Sergio, B., et al. (2022). ”Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors.” Journal of Parallel and Distributed Computing 167.

[23] Sun, X., et al. (2011). Optimizing SpMV for diagonal sparse matrices on GPU. 2011 international conference on parallel processing, IEEE.

[24] Williams, S., et al. (2007). Optimization of sparse matrix-vector multiplication on emerging multicore platforms. SC’07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, IEEE.

[25] Xia, Y., et al. (2018). A parallel solving algorithm on GPU for the time-domain linear system with diagonal sparse matrices. Workshop on Big Scientific Data Benchmarks, Architecture, and Systems, Springer.

[26] Yang, W., et al. (2018). ”A parallel computing method using blocked format with optimal partitioning for SpMV on GPU.” Journal of Computer and System Sciences 92: 152-170.