2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS) | 979-8-3503-3071-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICPADS60453.2023.00262

Efficiently Running SpMV on Multi-Core DSPs for Block Sparse Matrix

Deshun Bi*, Xiaowen Tian†, Shengguo Li*‡, Dezun Dong*‡
*National University of Defense Technology, China
{bds123, nudtlsg, dong}@nudt.edu.cn
†China University of Petroleum-Bejing, China
xiaowentian@student.cup.edu.cn

***Abstract*—Sparse Matrix-Vector Multiplication (SpMV) is a fundamental operation in sparse computations. Although many techniques have been developed to speed up SpMV, optimizing this process on low-power multicore digital signal processors (DSPs) has often been neglected. This paper presents the FT-M7032, a cutting-edge CPU-DSP hybrid multi-core processor. We assess the data transfer efficiency among various units to identify performance constraints of SpMV on multicore DSPs. Based on our evaluation, we develop a method for block sparse matrices, namely SpMV_BLOCK, which can break the bandwidth bottleneck of SpMV and achieve significant performance gains. We then propose a load-balancing strategy for each thread by using binary search and devise a pipeline that overlaps data transfers and computations to improve SpMV performance. To measure our method's effectiveness, we compared its performance against a baseline on the FT-M7032's general-purpose CPU cores. Our experiments show that our approach delivers a notable $5.80\times$ speedup over the baseline.**

***Index Terms*—SpMV, Multi-Core DSP, Performance Optimization, Load Balancing, Double Buffering**

I. Introduction

Sparse matrix-vector multiplication (SpMV) operations are pivotal in computational science and engineering [1]. With the expanding applications of SpMV, continuous research is being conducted to accelerate its performance on various high-performance architectures [2]–[4]. As energy efficiency becomes paramount, low-power embedded architectures, like digital signal processors (DSPs) [5], [6], are making their way into heterogeneous high-performance computing domains. While some efforts have been made to optimize general matrix multiplication on these platforms [7], [8], a standardized approach to optimizing SpMV on emerging multi-core DSPs remains elusive.

For better SpMV performance, optimizing the irregular memory access pattern is crucial. A basic method might store matrix A and vector $\mathbf{x}$ in main memory (like DDR RAM) and access x inconsistently. A more efficient strategy employs on-chip memory units, such as the FT-M7032 platform's scalar memory (SM), array memory (AM), and global shared memory (GSM) for data caching. Each memory unit has its own storage space and bandwidth. It's essential to distribute data across these units thoughtfully to maximize their use and enhance performance. To address this, we assess the data transfer capabilities between units to allocate matrix elements appropriately.

Sparse matrices arising from the discretization of partial differential equations, in which each grid point is associated with several degrees of freedom, exhibit a dense block substructure [9]. We show that such block sparse matrices are suitable for FT-M7032 when stored in SELL format [10], and we propose a new SpMV algorithm for its base FT-M7032. Taking the example of a sparse matrix containing 4*4 dense blocks, this approach requires only 1/4 of the access overhead for x to complete the SpMV computation, breaking the bandwidth bottleneck of SpMV. We refer to this new SpMV algorithm as SpMV_BLOCK and storage format based on the SELL format. We also use optimizations such as load balancing and double buffering to further improve SpMV's performance. We test our approach on the FT-M7032 and compare it with the SpMV implementation on standard CPU cores. Our results show that SpMV_BLOCK is superior on the FT-M7032, achieving up to $5.80\times$ faster speeds when using DSP to boost SpMV.

The main contributions of this paper are as follows:

- We conduct a straightforward and useful analysis to steer SpMV optimization on DSPs for block sparse matrix.
- We introduce a novel SpMV algorithm for sparse matrices with dense blocks, which can break the bandwidth bottlenecks of SpMV and efficiently perform SpMV.
- We design a load balancing method using binary search and propose a double buffer strategy to overlap data transfers and computation.

II. Background

A. FT-M7032 Heterogeneous Processor

The FT-M7032 platform consists of a 16-core ARMv8 CPU with four DSP clusters, providing a heterogeneous computing environment.

Fig. 1 presents the architecture of the FT-M7032 platform. It has a general-purpose (GP) area with CPU cores and an acceleration (ACC) area with four clusters, each cluster includes eight DSP cores, its own global shared memory (GSM), and

‡Corresponding authors.

2690-5965/23/$31.00 ©2023 IEEE
DOI 10.1109/ICPADS60453.2023.00262

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

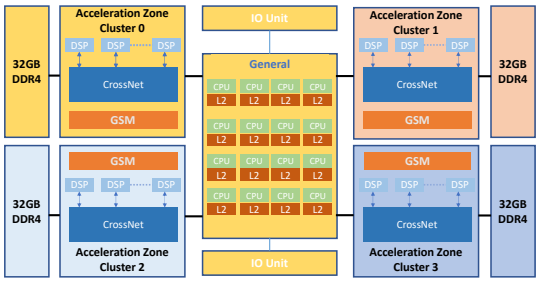


Fig. 1: Overview of the FT-M7032 architecture

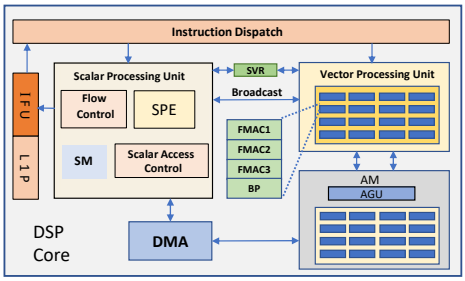


Fig. 2: Micro-Architecture of DSP Core in FT-M7032

off-chip DDR memory. While CPUs run at 2.0 GHz, DSPs operate at 1.8 GHz. Each cluster works independently, and the CPU cores manage inter-cluster data transfers.

As depicted in Fig. 2, the DSP core features a VLIW architecture, integrating an instruction function unit (IFU), scalar processing unit (SPU), vector processing unit (VPU), and a DMA engine. The IFU can handle 11 instructions per cycle, with five for scalar and six for vector operations. The SPU, containing a scalar processing element (SPE) and a 64KB scalar memory (SM), manages scalar tasks. In contrast, the VPU, designed for SIMD operations, boasts 16 vector processing elements (VPEs) and a 768KB array memory (AM), each VPE with 64-bit registers. The VPU process up to 1024 bits at once. The SPU and VPU communicate via broadcast instructions and SVR registers, while the DMA engine enables efficient data transfers between different levels of the memory hierarchy.

On the software front, `hthread` boosts multi-core DSPs' usability, while its runtime bridges the GP and ACC sections [11]. The GP acts as the host, and each acceleration cluster serves as a compute unit. The FT-M7032 has a dedicated communication channel, APip, in its DSP's DMA, which supports indirect memory access through a mode named SuperGather (SG) [12] or dma_sg. This mode transfers data across various memory spaces, and we utilize it to minimize SpMV memory access latency.

B. Sparse Matrix Storage Format

Sparse matrices are usually stored in a compressed format. This section introduces a mainstream sparse matrix storage format suitable for SIMD-based processors.

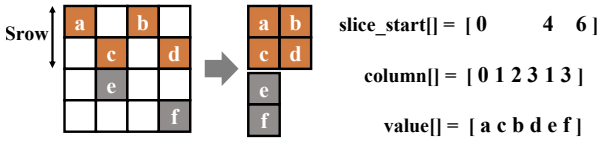


Fig. 3: An example of the SELL format for `Srow`=2. The matrix is horizontally sliced into two slices, each stored in column-major order.

Algorithm 1: SpMV pseudocode for the SELL format

```
Input: matrix A, vector x and Srow.
Output: vector y.
// rslice: number of slices
// colwid = (A.slice_start[k+1] − A.slice_start[k])/Srow
for k = 0 to rslice do
    for i = 0 to colwid do
        for j = 0 to Srow do
            idx = A.slice_start[k] + j + i × Srow
            y[k × Srow + j] += x[A.column[idx]] × A.value[idx]
        end
    end
end
```

1) Sliced ELL: The SELL (Sliced ELL) [10] format is an enhanced version of the ELL [13] format, offering greater efficiency in storing and processing sparse matrices. This format segments the sparse matrix into row slices, with each slice being represented in ELL format. The parameter, `Srow`, defines the number of consecutive rows within each slice. The SELL format consists of three data structures: `slice_start`, `column`, and `value`. As shown in Fig. 3, `column` and `value` store the column indexes and values of the nonzero elements in column-major order, and `slice_start` stores the index of the first element in each slice. Thus, the last number in `slice_start` denotes the total number of elements, e.g., there are 6 elements in Fig. 3. Then, we can calculate the number of elements per row in a slice i including the padding elements as $(slice_start[i+1] - slice_start[i])/\texttt{Srow}$.

Algorithm 1 presents the SpMV algorithm using the SELL format. Comprising three loops: The outer loop cycles through each slice. The middle loop traverses the columns of the current slice. The innermost loop performs multiplication and addition for nonzero elements with vector $\mathbf{x}$ to produce the vector $\mathbf{y}$.

C. Block Sparse Matrix

The block sparse matrix is characterized by having multiple dense blocks, where each dense block has a fixed size and consists of nonzero elements (see Fig. 5). Sparse matrices arising from higher-order discretizations and problems with multiple degrees of freedom per mesh point often exhibit a dense block substructure [9], [14], [15]. The dense block consists of many nonzero elements, while the rest of the matrix is zero. Block sparse matrices are widely used in

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

scientific computing, engineering, and data analysis. For example, stiffness matrices in finite element analysis often contain dense blocks to describe local deformations in a structure. In statistical modeling, covariance matrices can also have a dense block structure, reflecting the correlation between variables. The handling of block sparse matrices is therefore crucial for many applications in high-performance computing and scientific simulation.

III. Our Approach

A. Architecture Analysis

In our study, we analyze the memory bandwidth characteristics of the FT-M7032 hierarchy. Following this, we conduct a thorough assessment of data transfer patterns and, based on our findings, provide a prediction for SpMV's peak performance in this context.

1) Performance evaluation: As illustrated in Fig. 4 [16], the DSP's memory architecture is structured into three main layers: an external DDR, an on-chip GSM used collectively by the DSP cores, and an AM/SM dedicated to each individual DSP core. AM serves as the storage space for VPU, requiring the loading of matrix nonzeros and vector $\mathbf{x}$ before performing multiplications and additions of SpMV. SM serves as the storage area for scalar operations, including scalar multiplication and addition, along with vector reduction tasks. GSM functions as a high-speed on-chip memory serving as a shared L2 cache, accessible by both SPUs and VPUs.

The DMA engines of the FT-M7032 facilitate data transfer between memory units through two distinct mechanisms: the direct point-to-point mode (dma_p2p) and the more complex super-gather mode (dma_sg). While dma_p2p is a non-blocking direct transfer, dma_sg specializes in gathering operations on $\mathbf{x}$. Table I presents bandwidths between storage layers. dma_sg has notably less bandwidth than dma_p2p, especially from DDR to AM (around 4 GB/s in Table I). Thus, optimizing dma_sg usage is vital for SpMV performance on FT-M7032. In SpMV, matrix A's nonzero elements are stored sequentially, but vector $\mathbf{x}$ is accessed irregularly. Boosting data locality by caching or prefetching $\mathbf{x}$ can uplift SpMV's efficiency. Thus, it's advisable to cache $\mathbf{x}$'s elements in GSM instead of directly using dma_sg to transfer vector $\mathbf{x}$ from DDR to AM.

Fig. 4 shows the data transfer process: matrix values move from DDR to AM via dma_p2p. The vector $\mathbf{x}$ moves from DDR to GSM via dma_p2p, and then it is gathered from GSM to AM via dma_sg. The maximum bandwidth achievable with the dma_sg from GSM to AM is approximately 28 GB per second. Translated into computational terms, it can move 3.5G double-type elements per second when operating in double precision (FP64) for SpMV, calculated as $28 * 8 * 10^9/64$. This gives a peak throughput of 7 GFLOPs (3.5G elements per second times two operations per element), which is the upper limit of SpMV performance achievable with dma_sg on the FT-M7032.

This analysis forms the foundation for our SpMV implementation and performance prediction on DSP, independent of the chosen storage format. For this study, we chose the SELL format. Subsequent subsections detail our idea stemming from this analysis.

TABLE I: Measured bandwidth between storage layers

Function	Direction	Bandwidth
dma_p2p	DDR↔SM	39 GB/s
dma_p2p	DDR↔AM	34 GB/s
dma_sg	DDR↔AM	4 GB/s
dma_p2p	GSM↔AM	206 GB/s
dma_sg	GSM↔AM	28 GB/s

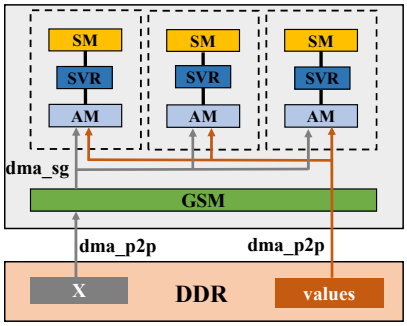


Fig. 4: The FT-M7032 storage hierarchy is abstracted into a three-level memory structure, with an example of three DSP cores in a cluster.

2) Breaking the bottleneck of dma_sg: From the prior section (Section III-A1), we discern that **dma_sg** is SpMV's bandwidth bottleneck, with a performance peak of just 7 GFLOPs, even when caching $\mathbf{x}$ in GSM. To break the bottleneck of **dma_sg**, we develop a new SpMV algorithm called SpMV_BLOCK for the sparse matrices with regular block substructures. If they are stored in the column major order, **dma_sg** is not the bottleneck any more, and the performance of SpMV can be improved significantly. The main idea is shown in Fig. 5.

For the matrix shown in Fig. 5, the column indexes of the elements in the same column of each sub-block are identical. As a result, we can exploit this structure to optimize the SpMV computation. Specifically, we can use **dma_sg** to gather only $1/4$ of the data (elements in green), and the remaining three entries can be copied directly in the AM (elements in blue). This approach can effectively alleviate the access bottleneck of **dma_sg**. In the next subsections, we describe our design based on the above analysis.

B. SpMV_BLOCK Design and Implementation

In this section, we introduce the basic implementation of SpMV_BLOCK and use optimization strategies such as thread-level load balancing and pipelining to further improve performance.

1) Reduce startup overhead: As shown in Fig. 4, the transfer of the `value` array and vector $\mathbf{x}$ requires calls to dma_p2p and dma_sg. A transfer process can be divided into

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

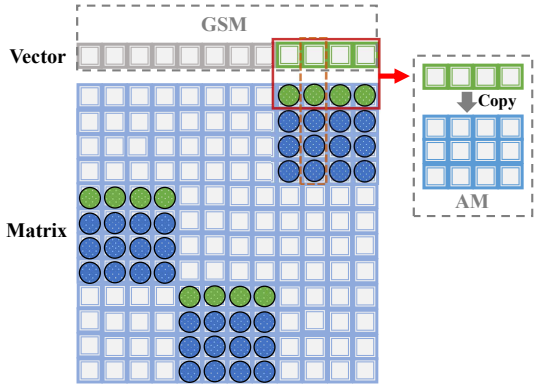


Fig. 5: The block sparse matrix. It consists of multiple dense blocks, each of which is a nonzero block with fixed constant size.

a startup overhead and a transfer overhead. Once the amount of data to be transferred is determined, the transfer overhead is unchanged. However, the number of calls affects the startup overhead, and if too many calls are made, the startup overhead will be high. So use the full bandwidth of dma_p2p and dma_sg as much as possible for each transfer. The AM has a capacity of 768KB. To perform the SpMV operation on the AM, we need to partition it into at least three sections to store: the transmitted $\mathbf{x}$, the `A.value`, and the calculated $\mathbf{y}$. From the storage space of AM, we can figure out that the maximum length of each transmission is: $768KB/(64*3) = 32768$, which is about $30*1024$. So each time we can transfer $30*1024$ `value` and $\mathbf{x}$ of double type to AM.

2) SpMV_BLOCK*:* Algorithm 2 outlines the workflow of SpMV_BLOCK. According to the multi-core feature of FT-M7032, we design a multi-thread parallel SpMV algorithm, namely SpMV_BLOCK.

The operations conducted between the head and tail are multi-thread parallel SpMV operations (Lines 1 to 33). The vector $\mathbf{x}$ is transferred from DDR memory to the GSM using the **dma_p2p** operation prior to the SpMV operation, as depicted in Line 3. After moving the vector $\mathbf{x}$, get the indexes of the start and end slices that each thread is responsible for processing (Line 7). After that, each thread processes its own slices in parallel, `nslice` slices at a time (Lines 8 to 32). Note that if the number of slices processed by the thread is less than `nslice`, it processes $stop-start$ slices. Lines 10 to 30 represent the processing of `nslice` slices. The `st` array represents the `slice_start` array in SELL format (Fig. 3).

Initially, we use dma_p2p and dma_sg to load matrix values and column indexes into AM (Lines 11, 12). Note that since only 1/4 of the column indices are stored, only the length/4 vector x is loaded from the GSM to the AM. Lines 14 to 18 show the exact procedure for the expansion of the vector x. We traverse each element of the vector x stored on AM from back to front, expanding it in turn. where the `mov_to_vlr` instruction indicates the switching of the VPE. `0x0001` indicates that only one core is turned on, at which point the VPU has only scalar functionality. `0xFFFF` indicates that 16 VPEs are enabled, restoring the VPU to vector operation. This instruction facilitates access to a single element for the expansion process. Once the vector x has been expanded, the SpMV calculation can be performed (Lines 22 to 29). The column widths of each slice are recorded in advance in the `colwid` array (Line 2 in Algorithm 1). `slice_id` is the index of the current slice (Line 19). `precol` and `curcol` are the cumulative number of columns from the `begin` slice to the previous slice and the current slice respectively (Line 20). Lines 22 to 29 indicate the number of columns processed at a time, these columns of length `Srow` come from multiple slices. Lines 23 to 25 show the computation process for the current slice, where 16 double-type elements can be calculated by VPU each time. Lines 26 to 28 indicate that the calculation results are buffered on AM and the next slice is calculated.

Algorithm 2: SpMV_BLOCK algorithm

Input: matrix A, vector xv, $Srow$, and $coreNum$.
Output: vector yv.

```
1  Function SpMV_BLOCK(A, xv, Srow, coreNum)
2      // Moving x from DDR to GSM via dma_p2p and m is the
         dimension of the vector x.
3      dma_p2p(xv_ddr, m * sizeof(double), xv_gsm)
4      // define length 30 × 1024
5      nslice = length/Srow
6      // tid indicates the thread index. tid ∈ (0,...,coreNum-1).
7      start = block_ptr[tid], stop = block_ptr[tid + 1]
8      for sli = start; sli < stop; sli += nslice do
9          begin = sli, end = sli + nslice
10         for offset = A.st[begin]; offset < A.st[end];
             offset += length do
11             dma_p2p(&A.value[offset], length *
                 sizeof(double), val_am)
12             dma_sg(xv_gsm, &A.column[offset/4], (length/4)*
                 sizeof(double), xv_am)
13             // Extend the vector x on AM by copying it.
14             mov_to_vlr(0x0001)
15             for i = length/4 to 0 do
16                 temp = vec_ld(i, xv_am)
                   vec_st(temp, i * 4 + 0, xv_am)
                   vec_st(temp, i * 4 + 1, xv_am)
                   vec_st(temp, i * 4 + 2, xv_am)
                   vec_st(temp, i * 4 + 3, xv_am)
17             end
18             mov_to_vlr(0xFFFF)
19             slice_id = begin
20             precol = 0, curcol = colwid[slice_id]
21             colnum = length/Srow
22             while curcol <= colnum && slice_id < end do
23                 for j = precol to curcol do
24                     vec += val_am[j] * xv_am[j]
25                 end
26                 yv_am[slice_id − begin] = vec
27                 precol = curcol
28                 curcol += colwid[++slice_id]
29             end
30         end
31         dma_p2p(yv_am, nslice *
             sizeof(double), &yv_ddr[begin * Srow])
32     end
33 end
```


Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3: binary_search algorithm

Input: $A.slice_start$, $A.rslice$, and $bound$.
Output: $index$.

```
1  Function binary_search(slice_start, rslice, bound)
2      // rslice: number of slices.
3      start = 0, end = rslice − 1
4      while end >= start do
5          mid = (end + start)/2
6          key = slice_start[mid]
7          if bound >= key then
8              start = mid + 1
9          end
10         else
11             end = mid − 1
12         end
13     end
14     return start
15 end
```

When `nslice` slices are computed, the results are written back to DDR memory (Line 31).

SPMV_BLOCK assigns slices to each thread, allowing efficient parallel execution of threads. By exploiting the structural properties of matrices, only part of the data of vector x is moved to AM, avoiding the bandwidth bottleneck of SpMV. The VPU consists of 16 VPEs, which can process 1024 bits in parallel, i.e. 16 double-precision operations simultaneously, further developing data-level parallelism.

3) Load balancing: To allow the DSP cores to achieve better thread-level parallelism, we design a load balancing strategy. We use binary search to determine the indexes of the start and end slices processed by each thread. In this way, each thread processes a similar number of nonzero elements to achieve load balancing between threads.

Algorithm 3 describes our binary search algorithm. The input parameters are the `slice_start` array of the sparse matrix A, the number of slices `rslice`, and `bound`. The `bound` indicates the start boundary of the thread's processing range under ideal load conditions. The `bound` is computed for each thread as follows:

$$avennz = \frac{slice_start[end]}{\texttt{coreNum}}, \texttt{bound} = coreId \times avennz.$$

First, calculate the average number of elements processed by each thread, `avennz`. The index of the DSP core is multiplied by `avennz`, which is ideally the start boundary for thread processing. The start boundary of the next thread is the end boundary of the current thread. The slice just larger than the ideal start boundary is searched using Algorithm 3, this is then used as the start slice for thread processing. The indexes of the slices returned by Algorithm 3 are stored in the `block_ptr` array in preparation for performing the SpMV operations (Line 7 in Algorithm 2).

Fig. 6 illustrates two allocation strategies, here using four threads as an example. Fig. 6-I shows an average allocation based on the number of slices, and Fig. 6-II shows an allocation using the load balancing strategy. Fig. 6-I allocates slices based on the number of slices, without considering the number of elements contained in each slice. There are 12 slices and each thread is allocated 3 slices. The number of elements processed by each thread is 13, 9, 8, and 6 respectively. After the last thread completes its processing, there's an additional wait time for 7 more elements before finalizing the matrix computation. Fig. 6-II uses Algorithm 3 to allocate slices to four threads. The start slice indexes processed by each thread are 0, 2, 5, and 8 respectively. The number of elements processed by each thread is 10, 9, 9, and 8 respectively, which achieves better load balancing compared to Fig. 6-I.

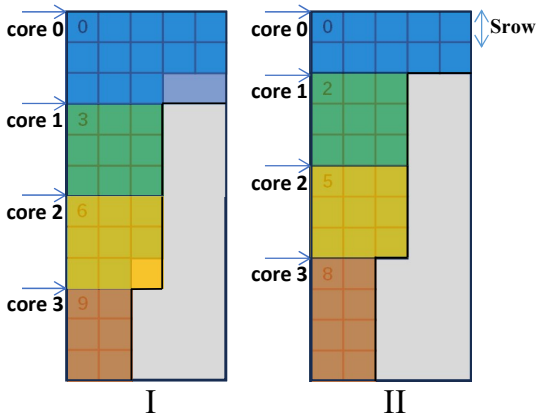


Fig. 6: Two allocation strategies. Figure I allocates slices based on the number of slices, without considering the number of elements contained in each slice. Figure II allocates slices by using Algorithm 2, which implements thread-level load balancing.

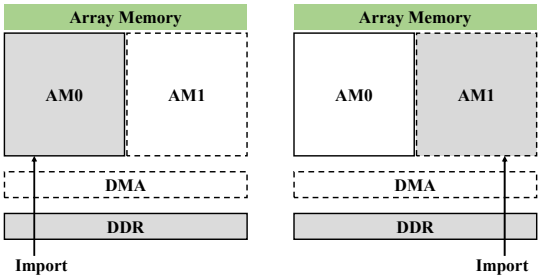


Fig. 7: Double buffering mechanism. We divide the AM space into two parts, one for data import and one for computation and export, in alternating order.

4) Double buffering: We implement a double buffering technique, which allows DMA data transfers and computational tasks to occur simultaneously [16]. As illustrated in Fig. 7, the AM is split into two sections: one handles data import, while the other manages computation and export. They work in turn. This setup lets us overlap data transfer with computations for all but the final data segment, enhancing the algorithm's overall speed and efficiency.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Input matrices used in our evaluation

ID	Input	Rows(N)	#nnz	#nnz/N	S(%)*
1	1w-4	0.01M	2.00M	200.13	2
2	2w-4	0.02M	7.98M	398.78	2
3	3w-4	0.03M	17.98M	599.43	2
4	4w-4	0.04M	32.03M	800.77	2
5	5w-4	0.05M	50.01M	1000.21	2
6	6w-4	0.06M	36.00M	600.03	1
7	7w-4	0.07M	48.99M	699.91	1
8	8w-4	0.08M	63.98M	799.75	1
9	9w-4	0.09M	81.01M	900.10	1
10	10w-4	0.10M	100.02M	1000.17	1
11	1w-2	0.01M	2.00M	200.12	2
12	2w-2	0.02M	8.00M	400.00	2
13	3w-2	0.03M	18.00M	600.03	2
14	4w-2	0.04M	32.00M	799.94	2
15	5w-2	0.05M	50.00M	1000.05	2
16	1w-8	0.01M	2.02M	202.37	2
17	2w-8	0.02M	7.99M	399.52	2
18	3w-8	0.03M	18.03M	601.03	2
19	4w-8	0.04M	32.12M	803.11	2
20	5w-8	0.05M	50.32M	1006.31	2

*S: The sparsity of block sparse matrices.

IV. Performance Results

A. Experimental Setup

The matrices in Table II are the block sparse matrices we generated. These matrices vary in size and sparsity, simulating different application scenarios. Take the matrix `1w-4.mtx` for example, where 1w is the dimension of the matrix and 4 means that it contains 4*4 dense blocks. We compare the performance of SpMV_BLOCK with the classical SpMV on the CPU cores in the GP area. The CPU cores run the Linux kernel version 5.4.0. The SpMV kernels are compiled with GCC version 9.3 and "-O2 -lpthread -fopenmp" option. Note that we use four threads for CPU execution.

B. Performance Evaluation

1) Performance of SpMV_BLOCK*:* Using the first ten matrices from Table II, we evaluate the SpMV_BLOCK's performance. We compare SpMV_BLOCK with a highly optimized SpMV kernel that runs on CPU cores with CSR storage formats, labeled as CSR_CPU in Fig. 8. The CPU kernel leverages four CPU cores in the FT-M7032's GP area, whereas SpMV_BLOCK is executed on eight DSP cores within a single DSP cluster. Fig. 8 provides a visual representation of the results.

We implemented four versions of SpMV on DSPs to facilitate analysis of the strengths and weaknesses of our approach:

1) SELL transfer the entire vector x from GSM to AM;
2) BLOCK is the base implementation of SpMV_BLOCK, effectively breaking the bandwidth bottleneck of dma_sg;
3) BLOCK-LB use binary search for thread-level load balancing;
4) BLOCK-LB-DB overlap data transfer and computation using double buffering strategy based on load balancing;

The comparison between `SELL` and `BLOCK` highlights SpMV bandwidth constraints. `BLOCK` overcomes the dma_sg bandwidth bottleneck, the process as depicted in Fig. 5. Referring to Fig. 8, `BLOCK` achieves an impressive average speedup of $1.57\times$ over `SELL`.

The performances of `BLOCK` and `BLOCK-LB` are compared to show the impact of load balancing optimization. The only difference between these two versions is the use of load-balancing optimization. Due to the typically regular distribution of block sparse matrices we generated, there is no performance gain from load balancing. However, for general sparse matrices, load balancing methods can significantly improve SpMV performance. To evaluate the benefits of double buffering, we compare `BLOCK-LB` with `BLOCK-LB-DB`. The latter uses pipelines to overlap data transfer and computation. From Fig. 8, `BLOCK-LB-DB` outperforms `BLOCK-LB` by an average speedup of 3.3%. The CPU results in Fig. 8 are obtained by running SpMV on the four CPU cores that access the same DDR in one DSP cluster. Compared with the CSR_CPU implementation, the speedup of `BLOCK-LB-DB` is $5.80\times$ times on average and is used as the final implementation of SpMV_BLOCK.

2) Impact of sparsity on performance: To explore the effect of different sparsities on the performance of SpMV, we generated five more matrices with a sparsity of 6%, as illustrated in Fig. 9. We use the first five matrices in Table II to compare performance with them. Fig. 9 shows that performance improves as sparsity increases. Yet, this improvement tapers off with increasing matrix size. On average, tripling the matrix's sparsity leads to a 16% performance boost.

3) Scalability of SpMV_BLOCK*:* This section uses the matrices numbered 1-5 and 11-20 in Table II to test SpMV_BLOCK. These matrices have the same sparsity but contain dense blocks of different sizes, which can be used to evaluate the algorithm's scalability. Each set of test cases in Fig. 10 contains three matrices, consisting of `2*2`, `4*4`, and `8*8` dense blocks respectively. From Fig. 10, it's evident that for every doubling of the dense block size, the average performance increases by 0.65 GFLOPs.

4) Bandwidth utilization analysis: For memory-bound operations, bandwidth utilization serves as a crucial metric for assessment. It's computed as follows:

$$bandwidth\ utilization = \frac{Actual_bandwidth}{Stream_bandwidth} \times 100\%,$$

where *Actual_bandwidth* is the actual bandwidth cost of SpMV_BLOCK and *Stream_bandwidth* is the bandwidth measured by using the STREAM benchmark [17], which is modified by using `hthread` to run on FT-M7032. The total access memory of SpMV_BLOCK equals to

$$slice_start[end] * (4/4 + 8) + n * 8 + m * 8,$$

where $slice_start[end]$ is the length of arrays `column` and `value` of SELL format (see Fig. 3), n and m are the row and column dimensions of the matrix, respectively. Note that

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

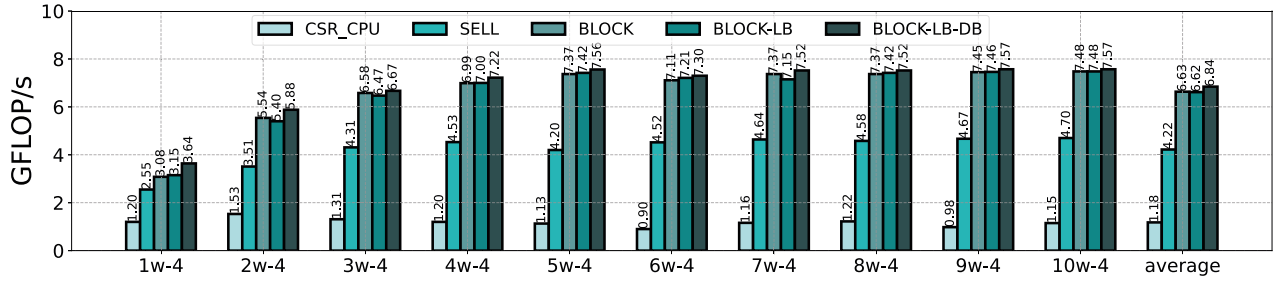


Fig. 8: Performance comparison of CPU-based SpMV vs SPMV_BLOCK

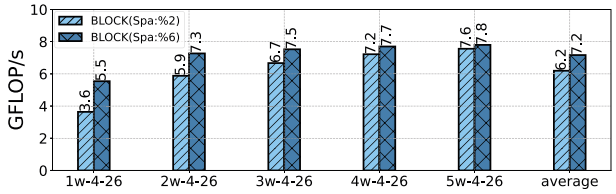


Fig. 9: Impact of sparsity on performance. We use two types of matrices with sparsities of 2 and 6. For example, for the `1w-4-26`, the left bar represents `1w-4-2.mtx` and the right bar represents `1w-4-6.mtx`.

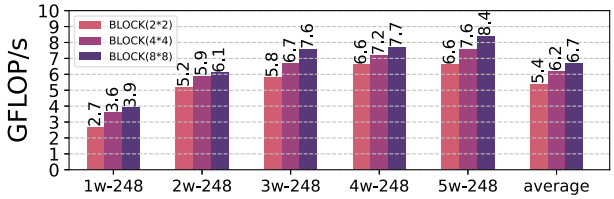


Fig. 10: Scalability of SPMV_BLOCK. Each test case set comprises three matrices with `2x2`, `4x4`, and `8x8` dense blocks, respectively, to assess the algorithm's scalability.

the actual access of the `column` array needs to be divided by 4 ($4/4$ in the formula), as we transmit only 1/4 of the column index. As shown in Fig. 11, the bandwidth utilization of the test matrices in Fig. 8 ranges from 47% to 96%, with an average of 86.8% (Stream_bandwidth is measured by using **dma_p2p** from DDR to AM, i.e., 34 GB/s).

V. Related Work

A lot of effort has gone into making SpMV faster by using various tricks, such as workload balancing [18]–[21], exploiting data locality [22]–[25], and using machine learning [26]–[30] to choose the best formats. Among the various strategies explored, tapping into block structures has gained significant interest. Earlier research [31]–[34] focused on using CPU registers to optimize tiny blocks, especially those coming from areas like finite element methods. Buluç [35] et al. developed the CSB format to take advantage of block layout and cache locality. Martone [36] came up with a recursive blocking method specifically for SpMV. Meanwhile, Yan [37] et al.

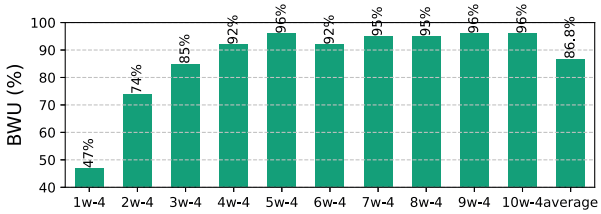


Fig. 11: Bandwidth utilization of SPMV_BLOCK. We test the bandwidth utilization of the matrices in Fig. 8 The maximum and minimum bandwidth utilization is 47% and 96% respectively, with an average bandwidth utilization of 86.8%.

focused on storing dense blocks and adjusting their sizes to boost performance.

Ryan [9] et al. proposed the BCSR format for block sparse matrix storage and computation with good compression efficiency. However, the bandwidth utilization of BCSR-based SpMV on Intel Xeon Phi KNC and K80 GPUs is very low. In this study, we introduce a novel SpMV algorithm tailored for block sparse matrices using the SELL format. Tested on the FT-M7032 architecture, our approach outperforms conventional CPU-based SpMV and enhances bandwidth utilization.

VI. Conclusion

In this paper, we propose a new SpMV algorithm for block sparse matrices, which can break the bandwidth bottlenecks of SpMV and efficiently perform SpMV on the FT-M7032. We further develop a load-balancing approach using binary search and design a computation pipeline to enhance memory access efficiency. Experimental results show that our approach significantly improves the performance of SpMV for block sparse matrices, with good scalability and high bandwidth utilization. Compared to SpMV on CPUs, we achieve a $5.80\times$ speedup, with an average bandwidth utilization of 86.8%.

Acknowledgement

This work was supported in part by the National Science Foundation of China (NSFC) under grant agreements 61902411, 62032023, 12002382, 11275269, 42104078, 62073333, and 62102444, the Excellent Youth Foundation of Hunan Province under grant agreement 2021JJ10050. We're

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.

grateful to Xiaoxiong Zhu and Yi Xie for their valuable suggestions for this article.

REFERENCES

[1] Y. Zhang, S. Li, F. Yuan, D. Dong, X. Yang, T. Li, and Z. Wang, "Memory-aware Optimization for Sequences of Sparse Matrix-Vector Multiplications," in *37th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2023.

[2] S. Ohshima, T. Katagiri, and M. Matsumoto, "Performance optimization of SpMV using CRS format by considering OpenMP scheduling on CPUs and MIC," in *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*. IEEE, 2014, pp. 253–260.

[3] Y. Kubota and D. Takahashi, "Optimization of sparse matrix-vector multiplication by auto selecting storage schemes on GPU," in *Computational Science and Its Applications-ICCSA 2011: International Conference, Santander, Spain, June 20-23, 2011. Proceedings, Part II 11*. Springer, 2011, pp. 547–561.

[4] H. Maeda and D. Takahashi, "Parallel sparse matrix-vector multiplication using accelerators," in *Computational Science and Its Applications–ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II 16*. Springer, 2016, pp. 3–18.

[5] A. Tiwari, V. Kumar, and G. Mitra, "High performance and energy optimal parallel programming on CPU and DSP based MPSOC," Ph.D. dissertation, Ph. D. dissertation, IIIT-Delhi, 2018.

[6] Y. Wang, C. Li, C. Liu, S. Liu, Y. Lei, J. Zhang, Y. Zhang, and Y. Guo, "Advancing DSP into HPC, AI, and beyond: challenges, mechanisms, and future directions," *CCF Transactions on High Performance Computing*, vol. 3, pp. 114–125, 2021.

[7] Z. Liu and X. Tian, "Vectorization of matrix multiplication for multi-core vector processors," *Chinese Journal of Computers*, vol. 10, 2018.

[8] S. Yin, Q. Wang, R. Hao, T. Zhou, S. Mei, and J. Liu, "Optimizing Irregular-Shaped Matrix-Matrix Multiplication on Multi-Core DSPs," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2022, pp. 451–461.

[9] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 663–672.

[10] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*. Springer, 2010, pp. 111–125.

[11] J. Fang, P. Zhang, C. Huang, T. Tang, K. Lu, R. Wang, and Z. Wang, "Programming Bare-Metal Accelerators with Heterogeneous Threading Models: A Case Study of Matrix-3000," *Frontiers of Information Technology & Electronic Engineering*, vol. 24, no. 4, pp. 509–520, 2023.

[12] S. Liu, Y. Cao, and S. Sun, "Mapping and Optimization Method of SpMV on Multi-DSP Accelerator," *Electronics*, vol. 11, no. 22, p. 3699, 2022.

[13] D. R. Kincaid, T. C. Oppe, and D. M. Young, "ITPACKV 2D user's guide," Texas Univ., Austin, TX (USA). Center for Numerical Analysis, Tech. Rep., 1989.

[14] R. W. Vuduc and H.-J. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure," in *High Performance Computing and Communications: First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005. Proceedings 1*. Springer, 2005, pp. 807–816.

[15] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," *ACM sigplan notices*, vol. 45, no. 5, pp. 115–126, 2010.

[16] D. Bi, S. Li, Y. Zhang, X. Yang, and D. Dong, "Efficiently Running SpMV on Multi-Core DSPs for Banded Matrix," in *International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2023)*, 2023, in press.

[17] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[18] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarath, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 781–792.

[19] M. Daga and J. L. Greathouse, "Structural agnostic spmv: Adapting csr-adaptive for irregular matrices," in *2015 IEEE 22nd International conference on high performance computing (HiPC)*. IEEE, 2015, pp. 64–74.

[20] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *SC'14*. IEEE, 2014, pp. 769–780.

[21] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[22] C. Alappat, A. Basermann, A. R. Bishop, H. Fehske, G. Hager, O. Schenk, J. Thies, and G. Wellein, "A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 3, pp. 1–37, 2020.

[23] F. GÖBEL, T. GRÜTZMACHER, P. NAYAK, and T. RIBIZEL, "G: A modern linear operator algebra framework for high performance computing," 2022.

[24] K. Kourtis, V. Karakasis, G. Goumas, and N. Koziris, "Csx: an extended compression format for spmv on shared memory systems," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 247–256, 2011.

[25] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient sparse matrix-vector multiplication on x86-based many-core processors," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 273–282.

[26] A. Benatia, W. Ji, Y. Wang, and F. Shi, "Sparse matrix format selection with multiclass svm for spmv on gpu," in *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 496–505.

[27] Z. Du, J. Li, Y. Wang, X. Li, G. Tan, and N. Sun, "Alphasparse: generating high performance spmv codes directly from sparse matrices," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[28] J. Li, G. Tan, M. Chen, and N. Sun, "Smat: An input adaptive auto-tuner for sparse matrix-vector multiplication," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 117–126.

[29] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on gpus," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 99–108.

[30] G. Tan, J. Liu, and J. Li, "Design and implementation of adaptive spmv library for multicore and many-core architecture," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 4, pp. 1–25, 2018.

[31] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 2002, pp. 26–26.

[32] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 293–312, 2005.

[33] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 135–158, 2004.

[34] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *Computational Science—ICCS 2001: International Conference San Francisco, CA, USA, May 28–30, 2001 Proceedings, Part I 1*. Springer, 2001, pp. 127–136.

[35] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. 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*, 2009, pp. 233–244.

[36] M. Martone, "Efficient multithreaded untransposed, transposed or symmetric sparse matrix–vector multiplication with the recursive sparse blocks format," *Parallel Computing*, vol. 40, no. 7, pp. 251–270, 2014.

[37] S. Yan, C. Li, Y. Zhang, and H. Zhou, "yaspmv: Yet another spmv framework on gpus," *Acm Sigplan Notices*, vol. 49, no. 8, pp. 107–118, 2014.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 22,2026 at 22:03:18 UTC from IEEE Xplore. Restrictions apply.