

Efficiently Running SpMV on Multi-core DSPs for Banded Matrix

Deshun Bi, Shengguo Li(✉), Yichen Zhang, Xiaojian Yang, and Dezun Dong(✉)

National University of Defense Technology, Changsha, China
{bds123,nudtlsg,zhangyichen,yangxj,dong}@nudt.edu.cn

Abstract. Sparse matrix-vector multiplication (SpMV) plays a pivotal role in large-scale scientific computing. Despite the increasing use of low-power multicore digital signal processors (DSPs) in high performance computing (HPC) systems, optimizing SpMV on these platforms has been largely overlooked. This paper introduces the FT-M7032, a new CPU-DSP heterogeneous processor multi-core platform for high-performance computing. The FT-M7032 provides programmable memory units at multiple levels, but effectively utilizing these units poses a challenge. To address this, we evaluate the transfer capability between different units to map matrix elements to storage units. Based on our evaluation, we propose an efficient parallel implementation, SpMV_Band, specifically designed for banded matrices. Furthermore, we devise a computation pipeline that optimizes memory access overhead by overlapping data transfers and computations. To evaluate our approach, we compare its performance with a baseline executed on the general-purpose CPU cores of the FT-M7032 heterogeneous platform. Experimental results demonstrate that our techniques achieve a significant speedup of 2.0× compared to the competing baselines.

Keywords: SpMV · Multi-Core DSP · Performance Optimization · Double Buffering

1 Introduction

Sparse matrix-vector multiplication (SpMV), as the key program of the Basic Linear Algebra Subroutine (BLAS) library, has been widely used in scientific simulation, data analysis, deep learning and other fields [12,22,39]. Due to the wide application of SpMV, research on accelerated SpMV in various high-performance architectures such as multi-core CPU [34], GPGPU [15,41] and MIC [40] has been constantly emerging. Due to energy efficiency and power constraints, low-power embedded architectures are being introduced into heterogeneous high performance computing domains, such as digital signal processors (DSPs) [11,32,33]. Compared to CPUs and GPUs, DSPs typically have very long instruction words (VLIWs) or vector cores without out-of-order execution [38]. DSP cores typically operate on software-managed on-chip memory and incorporate a direct memory access (DMA) engine. Therefore, existing optimizations

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2024
Z. Tari et al. (Eds.): ICA3PP 2023, LNCS 14491, pp. 201–220, 2024.
https://doi.org/10.1007/978-981-97-0808-6_12

for SpMV on CPUs and GPUs cannot be directly applied to SpMV on DSPs. Although some studies have optimized general matrix multiplication [23,38], there is no consensus on how best to optimize SpMV on these emerging multi-core DSP platforms.

To improve the performance of SpMV, it is essential to optimize the irregular memory access pattern [21]. A naive implementation would simply store the matrix A and the vector **x** in main memory (e.g., DDR RAM) and access x irregularly. However, a more effective approach is to use on-chip programmable memory units, such as the scalar memory (SM), array memory (AM) and global shared memory (GSM) provided by the FT-M7032 platform [38], to cache the data. These memory units have different capabilities in terms of memory space, access latency and bandwidth. Therefore, it is crucial to strategically distribute data among these units to maximise their utilisation and improve performance. We evaluate the data transfer between different storage units. Our results show that GSM has a particularly significant impact on SpMV. By exploiting the caching capabilities of GSM for the vector **x**, significant improvements in SpMV performance can be achieved.

Banded matrices are a common type of matrix with a wide range of applications, and many sparse matrices in SuiteSparse [6] can be converted into banded matrices by using the Reverse Cuthill McKee (RCM) ordering [5,16]. To optimize the use of onchip storage for large-scale SpMV, we implemented a horizontal matrix partitioning approach. This approach partitions the banded matrix into row blocks, each of which corresponds to a vector **x** segment, which promotes good data locality for x. The row blocks are then computed sequentially using GSM. We refer to this new SpMV algorithm as SpMV_Band and storage format based the SELL format. We evaluate our approach on FT-M7032 and compare it with the SpMV implmenetaion on the general-purpose CPU cores. Our experimental results demonstrate that SpMV_Band outperforms other competing methods on the FT-M7032, achieving speedups of up to 2.0×, when DSP is used to accelerate SpMV.

The main contributions of this paper are as follows:

- We make an effective analysis to guide SpMV optimization on DSPs.
- We propose a new SpMV algorithm for banded matrices, which partitions the sparse matrix horizontally to achieve a better data locality of x.
- We design a computation pipeline that overlaps data transfers and computation to minimize memory access overhead.

2 Background

2.1 FT-M7032 Heterogeneous Processor

The FT-M7032 platform combines CPU and DSP capabilities for heterogeneous computing. It features a 16-core ARMv8 CPU that runs the operating system and four DSP clusters for general-purpose computing (GPDSP).

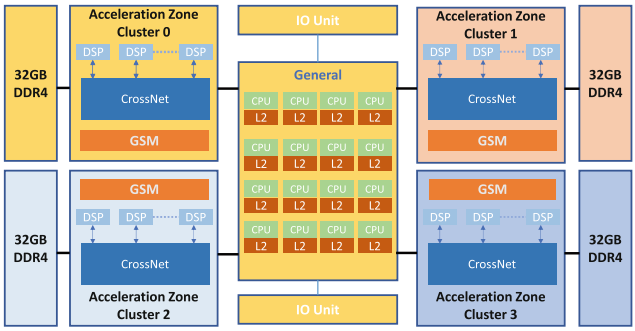


Fig. 1. Overview of the FT-M7032 architecture

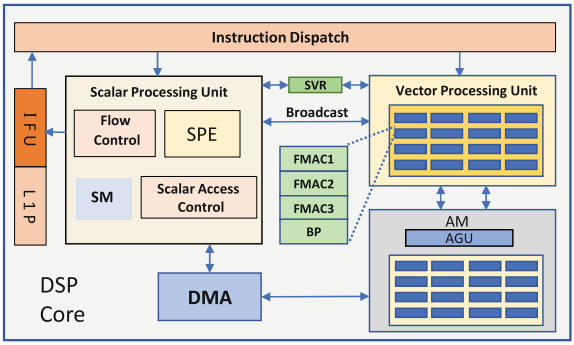


Fig. 2. Micro-Architecture of DSP Core in FT-M7032

Figure 1 shows the FT-M7032 platform's architecture. It consists of two areas: the general-purpose area (GP) with CPU cores and the acceleration area (ACC) with DSP cores. The ACC area has four clusters, each with eight DSP cores, a global shared memory (GSM), and an off-chip DDR memory. The CPU cores run at 2.0 GHz and the DSP cores run at 1.8 GHz. Each cluster in the ACC area can work independently and access its own GSM. The CPU cores manage the data transfers between different DSP clusters.

The DSP core is a VLIW-based architecture, as shown in Fig. 2. It has an instruction function unit (IFU), a scalar processing unit (SPU), a vector processing unit (VPU), and a DMA engine. The IFU can issue up to 11 instructions per cycle, including five scalar and six vector instructions. The SPU handles instruction flow control and scalar computations. It has an SPE (Scalar Processing Element) and an SM (Scalar Memory) of 64KB. The VPU performs vector computations in a SIMD manner. It has 16 VPEs (Vector Processing Elements), each with 64-bit registers and various units for arithmetic and memory operations. The VPU can process 1024 bits at once. It also has 768KB Array Memory (AM). The SPU and VPU communicate through broadcast instructions

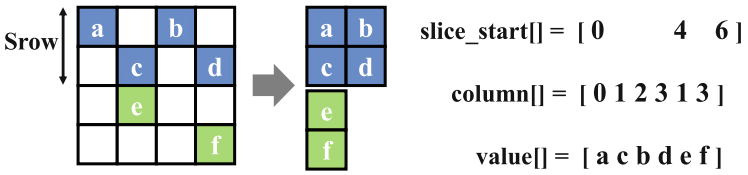


Fig. 3. An example of the SELL format for `Srow` = 2. The matrix is horizontally sliced into two slices, each stored in column-major order.

and shared registers. The DMA engine transfers data between different memory levels, such as main memory, GSM, and SM/AM.

On the software, `hthread` is provided to improve the programmability of multi-core DSPs [7]. The `hthread` runtime handles the interaction between GP and ACC regions. The GP side acts as the host and each acceleration cluster as a computing device. FT-M7032 has a dedicated host physical communication channel, APip, in the DSP's DMA. It supports indirect memory access by using a special data transfer mode called SuperGather (SG) [18] or dma_sg. It can move data between different memories. We use the dma_sg to support indirect memory accesses to reduce the latency of SpMV memory access.

2.2 Sparse Matrix Storage Format

This section presents two common storage formats for sparse matrices that suit SIMD-based processors. They compress the sparse matrix to save space.

ELLPACK. The ELLPACK [13] format is for GPUs and vector architectures. It stores nonzero elements in column order. Each row has K elements, where K is the largest number of nonzero elements in any row of the Sparse Matrix. Rows with fewer nonzero elements are padded with zeros. The drawback of ELL is that it wastes space with padding. It works well for matrices with similar numbers of nonzero elements per row.

Sliced ELLpack. The SELL (Sliced ELLpack) [27] format is a modified version of the ELL format. It is more efficient than ELL for storing and computing sparse matrices. It splits the sparse matrix into row slices. Each slice is stored in the ELL format. The slice size `Srow` is a parameter of this format. It determines how many adjacent rows are in each row slice.

The SELL format has three data structures: `slice_start`, `column`, and `value`. Figure 3 shows how they store the sparse matrix. `column` and `value` store the column indexes and values of the nonzero elements in column-major order. `slice_start` stores the index of the first element in each slice. The last number in `slice_start` is the total number of elements. For example, Fig. 3 has 6 elements. We can calculate the number of elements per row of a slice i, including the padding elements, by $(slice_start[i+1] - slice_start[i])/\texttt{Srow}$. When `Srow`

Algorithm 1: SpMV pseudocode for the SELL format

```
Input: Matrix A, vector x and Srow.
Output: Vector y.
// ns: number of slices
for k = 0 to ns do
    for i = 0 to (A.slice_start[k + 1] − A.slice_start[k])/Srow 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, SELL is the same as the CSR format and does not need padding. When `Srow` = n (the dimension of the matrix), SELL is the same as the ELL format and has the most padding. A small `Srow` saves space and improves bandwidth usage. A large `Srow` is better for vectorization, which suits the DSP architecture.

Algorithm 1 shows the SELL-based SpMV algorithm. It has three loops. The outer loop goes through each slice. The middle loop goes through each column of the current slice. The inner loop multiplies and adds the nonzero element values of each column and the vector **x** to get the result vector y. Section 3 explains how the SELL format enables parallelization and vectorization.

2.3 Banded Matrix

If the nonzero elements of a matrix are distributed only along the main diagonal and its adjacent diagonal lines, it is called a banded matrix [10,30]. Banded matrices have a wide range of applications and many matrices can be converted to banded matrices by reordering. Since the nonzero elements of banded matrices have certain distribution characteristics, some specific storage and computation methods can be used to improve the efficiency of operations.

The bandwidth of a banded matrix refers to the maximum distance that nonzero element deviates from the main diagonal. In a banded matrix, the nonzero elements can be divided into three parts: the upper triangle region, the main diagonal and the lower triangle region. The upper triangle region contains all nonzero elements above the main diagonal, and its bandwidth is called the upper bandwidth. The lower triangle region contains all nonzero elements below the main diagonal, and its bandwidth is called the lower bandwidth. The upper bandwidth and the lower bandwidth can be different. As shown in the band matrix in the Fig. 7, we use `SBL` and `SBU` to denote the lower bandwidth and upper bandwidth respectively.

Table 1. Measured bandwidth between storage layers

Function	Direction	Bandwidth
dma_p2p	DDR↔SM	39 GB/s
dma_p2p	DDR↔AM	30 GB/s
dma_sg	DDR↔AM	4 GB/s
dma_p2p	GSM↔AM	206 GB/s
dma_sg	GSM↔AM	28 GB/s

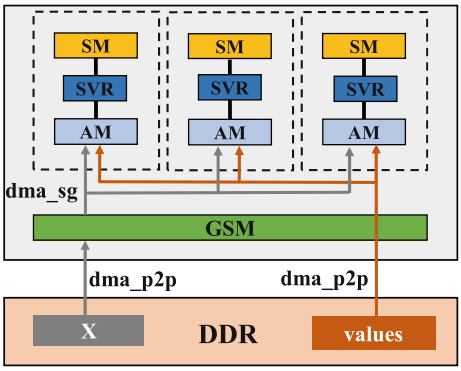


Fig. 4. The FT-M7032 storage hierarchy is abstracted into a three-level memory structure, with an example of three DSP cores in a cluster.

3 Our Approach

3.1 Architecture Analysis

We evaluate the memory access bandwidth between the levels of the FT-M7032 memory hierarchy. Based on the evaluation, we perform a data transfer analysis and predict the peak performance of SpMV. We then propose that a horizontal partitioning approach can guarantee data locality and efficiently execute SpMV on FT-M7032.

Performance Evaluation. The DSP memory architecture has three main levels, as shown in Fig. 4: an off-chip DDR, an on-chip GSM shared by DSP cores, and AM/SM private to a DSP core. AM is the storage space for VPUs. It needs to load the matrix nonzeros and the vector $\mathbf{x}$ before doing multiplications and additions of SpMV. SM is for scalar processing, such as scalar multiplication and addition, and vector reduction. GSM is a high-speed on-chip memory that acts as a shared L2 cache. Both SPUs and VPUs can access it.

The DMA engines on FT-M7032 move data between memory storages using two modes: the peer-to-peer mode (dma_p2p) and the super-gather mode

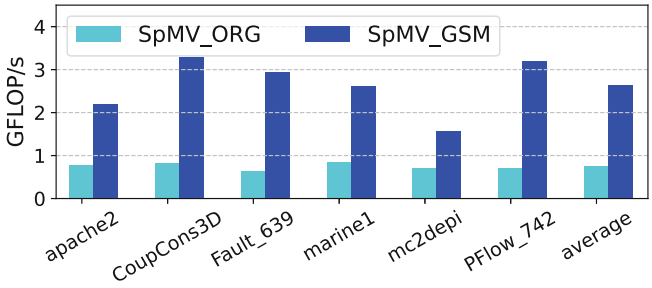


Fig. 5. Performance comparison of SpMV_ORG vs SpMV_GSM. SpMV_GSM, which uses GSM to cache x, but SpMV_ORG, which stores x in DDR without caching.

(dma_sg). The former is a non-blocking point-to-point transfer function. The latter is for gathering operations on **x**. Table 1 shows the measured bandwidth between storage layers. This is an average value selected after several tests, and the actual values may fluctuate around the table values. We can see that dma_sg has a much lower bandwidth than dma_p2p, especially when moving data from DDR to AM. Therefore, effective use of dma_sg is important for SpMV performance optimization on FT-M7032.

For the SpMV operation, the nonzero elements of matrix A are stored continuously while the elements of vector **x** are accessed irregularly. Therefore, improving data locality through caching or prefetching **x** can enhance SpMV performance. However, the bandwidth of the DMA gathering operation (i.e., dma_sg) for transferring data from DDR to AM is low (around 4 GB/s in Table 1), so we should avoid using dma_sg to transfer vector **x** from DDR to AM. Instead, we use GSM to cache as many elements of x as possible, which allows for faster data access and better bandwidth utilization. Figure 4 illustrates how we move matrix values and vector **x** among different storage units. First, we transfer the nonzero values of the sparse matrix from DDR to AM directly via dma_p2p. Second, we transfer vector **x** from DDR to GSM via dma_p2p and then gather it to AM from GSM via dma_sg. We can use SM to cache intermediate arrays used in SpMV, such as the `slice_start` in SELL format or the `row_pointer` [39] in CSR format, to reduce access latency. We transfer the intermediate arrays from DDR to SM via dma_p2p.

The peak bandwidth of dma_sg from GSM to AM is about 28 GB per second. This means that we can transfer 28/8=3.5G elements per second using double precision (FP64) for SpMV. Therefore, the peak throughput is 3.5* 2=7 GFLOPs, which is the maximum SpMV performance we can achieve using dma_sg on FT-M7032. This analysis provides a basis for our implementation and performance estimation of SpMV on DSP, regardless of the sparse matrix storage format we use. In this work, we choose the SELL format to implement. In the following subsections, we describe our design based on above analysis.

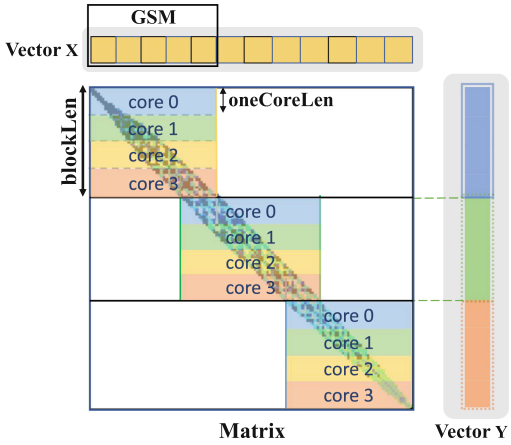


Fig. 6. Band matrix for horizontal division. SpMV_Band divides the matrix into row blocks. Each block is equally allocated to each DSP core for parallel computation. When a row block is to be computed, the corresponding segments of vector x are moved into GSM.

Strengthen Data Locality. In the previous section, we showed that caching the vector **x** in the on-chip memory GSM can reduce the memory access latency for SpMV. To evaluate the performance benefit of this optimization, we compared SpMV using GSM with SpMV_ORG, which stores **x** in DDR without caching. Figure 5 illustrates the speedup of SpMV using GSM over SpMV_ORG for different matrices. We can see that GSM improves the SpMV performance for all cases. Therefore, we conclude that storing **x** in GSM for indirect memory access is a general and effective optimization strategy on FT-M7032.

Although SpMV using GSM can perform well on the FT-M7032, it should be noted that the memory capacity of GSM is limited. If the size of the vector **x** is larger than `GSMlenMax` (6MB or a double precision vector of about 780,000 elements), GSM cannot hold the entire vector **x**. To solve this problem, we develop the SpMV_Band algorithm, which partitions the matrix in the horizontal direction, each row block is calculated one by one using GSM, and can handle arbitrarily size banded matrices.

As shown in the Fig. 6. Because the nonzero elements of each row block in the banded matrix are centrally distributed, good data locality is ensured and irregular memory accesses to the vector **x** are avoided.

3.2 SpMV_Band Design and Implementation

Loading Vector x to GSM. To move vector **x** into GSM space, the vector **x** range is calculated firstly based on the row block length `blockLen`, upper bandwidth `UBL`, and lower bandwidth `SBL`. As shown in the Fig. 7, according to the start row number `StartRow` and end row number `EndRow` (`StartRow` + `blockLen`) of the current row block, the moving range of **x** can be calculated

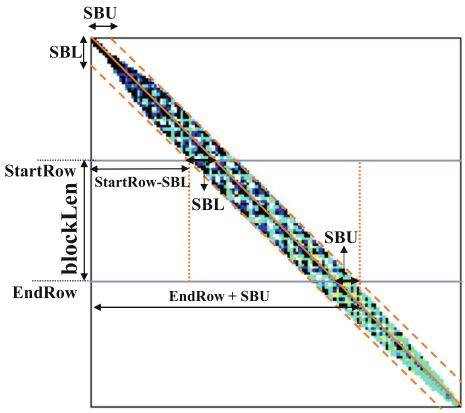


Fig. 7. Loading X to GSM in segments. According to the start row number StartRow and the end row number EndRow of the current row block, the corresponding movement range of x can be calculated.

as (`StartRow - SBL`, `EndRow + SBU`). Note that the **x** fragment corresponding to each row block has the overlap of `SBU + SBL` length.

SpMV_Band partitions the matrix into row blocks. When a row block is to be computed, the corresponding segments of vector **x** are moved into GSM. Since the segments of **x** are moved to GSM one by one, new segment will overwrite the old segment of **x** in GSM. So, the index of current segment of **x** in GSM is changed. Therefore, the column indexes of sparse matrix should be updated when accessing the elements of **x** stored in GSM via dma_sg. We can preprocess the `column` index array on the host side to ensure the subsequent matches is correct, and the preprocessing is shown in Algorithm 2.

`alpha` is the number of row blocks divided, it can be calculated from the matrix dimension `n` and `blockLen`. First, we determine whether there are more than one row block divided, and if only one row block is divided, no preprocessing is required (Line 1). Otherwise, the `column` array from the second row block to the penultimate row block is index shifted (Lines 4–11). The column index offset of the i-th row block is: $column[i] - col_shift$, where $col_shift = i * \texttt{blockLen} - \texttt{SBL}$. The last row block is the boundary value and needs to be handled separately (Lines 13–17). Before updating the `column` array, the indexes of the first and last slices in the row block are calculated (Lines 5–6). The column index of the element in the current row block can then be updated (Lines 7–9).

Multilevel Partitioning. Since AM is often too small to fit all matrix and vector elements, we designed a multilevel partitioning mechanism for SpMV to take full advantage of the memory hierarchy on FT-M7032. As shown in the Fig. 6, we partition the input matrix, A, into row blocks of length `blockLen`, where `blockLen` is equal to `coreNum*Srow*beta`. Here `beta` is a hyperparameter indicating that in each row block, each DSP core is responsible for processing

Algorithm 2: Column index preprocessing

Input: Banded Matrix A, SBL, $beta$, $coreNum$ and $blockLen$.
Output: $A.column$.

```
if alpha > 1 then
    // The second to last-1 BLOCK.
    col_shift = blockLen − SBL
    for i = 1 to alpha − 1 do
        StartSlice = beta ∗ coreNum ∗ i
        EndSlice = StartSlice + beta ∗ coreNum
        for j=A.slice_start[StartSlice] to A.slice_start[EndSlice] do
            A.column[j] = (A.column[j] − col_shift)
        end
        col_shift+ = blockLen
    end
    // The last BLOCK.
    StartSlice = beta ∗ coreNum ∗ (alpha − 1)
    col_shift = (alpha − 1) ∗ blockLen − SBL
    for j=A.slice_start[StartSlice] to A.slice_start[end] do
        A.column[j] = (A.column[j] − col_shift)
    end
end
```

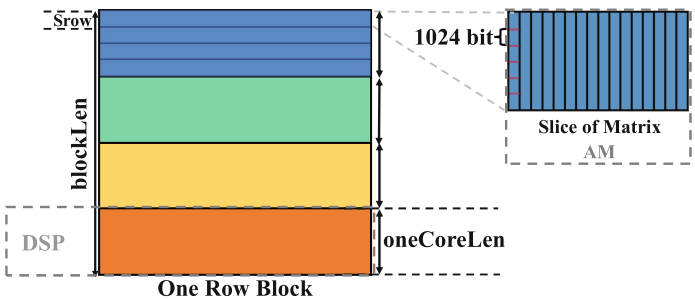


Fig. 8. Multilevel partitioning. For each row block, it is equally assigned to all DSP cores for processing. Four cores are used here as an example, and there are eight DSP cores in the actual experiment.

`beta` slices of length `Srow`. For each row block, multiple cores compute in parallel. The single row block partitioning process is illustrated in Fig. 8. Due to the capacity of AM, we use `AMlenMax` to denote the maximum size of matrix values and vector `x` that can be moved into AM at one time. The length of `Srow` cannot exceed `AMlenMax`. Each time, matrix elements of length `Srow` are processed. The computation is performed in the column-major order, and each column is divided into cells with a length of 16, and is calculated cell by cell. If the matrix cannot

Algorithm 3: SpMV_Band Algorithm

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

```
1  Function SpMV_Band(A, xv, Srow, coreNum, Beta)
2      // Get the ID of each running thread.
3      coreID = get_thread_id()
4      oneCoreLen = Beta * Srow
5      blockLen = coreNum * oneCoreLen
6      for blockCycle = 0 to alpha do
7          // X_Start = StartRow − SBL and X_End = EndRow + SBU.
8          X_Len = X_End − X_Start + 1
9          // Moving x from DDR to GSM via dma_p2p.
10         dma_p2p(xv_ddr + X_Start, X_Len * sizeof(double), xv_gsm)
11         group_barrier()
12         // Where blockoffset is equal to blockCycle * blockLen and coreOffset is
             equal to coreID * oneCoreLen. coreID ∈ (0,...,coreNum-1).
13         offset = blockOffset + coreOffset
14         sj0 = offset/Srow
15         // oneCoreLen is divided into Beta * Srow.
16         for sliceCycle = 0 to Beta do
17             yv_ddr = &yv[offset + sliceCycle * Srow]
18             sj = sj0 + sliceCycle
19             colwid = (A.slice_start[sj + 1] − A.slice_start[sj])/Srow
20             // Iterate across the column width of each slice.
21             for colCycle = 0 to colwid do
22                 sliceOffset = A.slice_start[sj] + colCycle * Srow
23                 val_ddr = &A.value[sliceOffset]
24                 col_ddr = &A.column[sliceOffset]
25                 dma_p2p(val_ddr, Srow * sizeof(double), val_am)
26                 dma_sg(xv_gsm, col_ddr, Srow * sizeof(double), xv_am)
27                 // Srow is divided into vpuTimes*16, and 16 doubles are
                     calculated each time.
28                 for vpuCycle = 0 to vpuTimes do
29                     yv_am[vpuCycle]+ = val_am[vpuCycle] * xv_am[vpuCycle]
30                 end
31             end
32             dma_p2p(yv_am, Srow * sizeof(double), yv_ddr)
33         end
34         group_barrier()
35     end
36 end
```

be divided equally to multiple DSPs, the remaining elements can be computed by the CPU cores within the GP region.

SpMV_Band. According to the multi-core feature of FT-M7032, we design a multi-level parallel SpMV algorithm based on the SELL format for banded matrices, namely SpMV_Band. SpMV_Band is implemented in `hthread`, which consists of host side and kernel side.

The host is responsible for allocating buffers, loading matrices and vectors, and managing the kernel side. The kernel side divides the sparse matrix into multiple row blocks, each with a length of `blockLen`. When performing SpMV operations, each row block corresponds to a part of the result vector y (Fig. 6), and the computations of row blocks are independent of each other. Each row block is divided into `coreNum` blocks of `oneCoreLen` length. Because it is stored in SELL format, each subblock of length `oneCoreLen`, consist of `beta` slices, and each slices consists of `Srow` rows.

Algorithm 3 describes our SpMV_Band algorithm. The operations between head and tail are thread-parallel SpMV (Lines 1–36). Before execution, get the id of each thread (Line 3). The matrix is divided into `alpha` row blocks according to the matrix dimension and `blockLen`, and Lines 6–35 are calculated one by one for each row block. Line 8 calculate the length of the x segment corresponding to each row block (Fig. 7). The start and end points of each segment must be greater than or equal to 0 and less than the matrix dimension. The vector **x** segment is moved from DDR memory to the GSM via dma_p2p (Line 10). Each row block is executed in parallel by multiple cores. As the matrix is stored in the SELL format, each row block consists of `coreNum`$*$`beta` slices. Each core is responsible for executing `beta` slices. After moving the vector **x**, each slice is traversed for calculation (Lines 16 and 33). To get the index of each slice in advance (Line 13, 14, 18), the column width of each slice can be calculated later (Line 19). Lines 21 and 31 are the core of the algorithm. For the current slice, it iterates through each column and load `Srow` nonzero elements into AM at a time via dma_p2p (Line 25). For the matched `Srow` length element in vector **x**, we load them into AM space via dma_sg (Line 26). Lines 28–30 are the specific calculation process, where 16 elements of the double type can be calculated each time. After the current slice has been calculated, the result vector is stored back in the DDR and the next slice is calculated (Line 32). It is noted that barrier synchronization should be set to ensure that data movement and SpMV calculation for each row block is correct (Line 11, 34).

SpMV_Band divides the matrix into multiple row blocks, which are processed simultaneously by different DSP cores. Each core computes different elements simultaneously, realizing thread-level parallelism and data-level parallelism. The VPU in the DSP core provides powerful vector computation capability and consists of 16 VPEs. The VPU can process 1024 bits in parallel, which means 16 double-precision operations at once, further developing data-level parallelism.

Multi-core Migration As mentioned earlier, data in GSM can be shared by all threads within the same DSP cluster. If only core 0 is responsible for moving the data, other cores cannot access the data in GSM until core 0 has completed the transfer of x to GSM. To improve performance, multiple cores can simultaneously

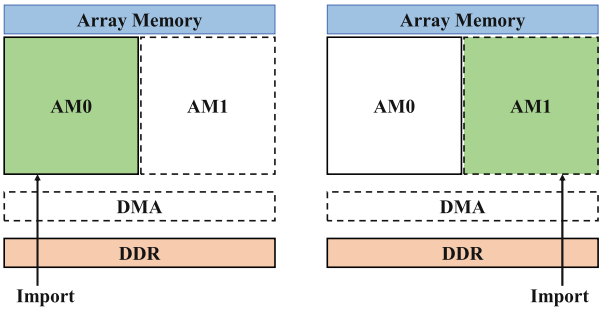


Fig. 9. Double buffer mechanism. We divide the AM space into two parts, one for data import and one for data computation and export, in alternating order.

move x from DDR to GSM via dma_p2p. Barrier synchronization should be used to ensure successful data movement before SpMV computation begins (Line 11). This strategy improves SpMV performance as shown in Fig. 10.

Double Buffering. We propose a two-level DMA-based double buffering strategy to optimize data transfer between multi-level memory. This strategy overlaps DMA data movement and computation time to further improve SpMV performance. As shown in Fig. 9, AM is equally divided into two parts: one for data import and the other for data computation and export, in alternating order. Except for the last part of the data, computations for all parts can be overlapped with data transfers to improve computational efficiency. The results are shown in Fig. 10.

4 Performance Results

4.1 Experimental Setup

Table 2 lists the banded matrices from the SuiteSparse Collection [6] used in our evaluation. These matrices vary in size and sparsity, which were extracted from different application domains. Therefore, they represent a wide range of inputs for SpMV in HPC applications. We compare the performance of SpMV_Band 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.

4.2 Performance Evaluation

In this section, we evaluate the performance of SpMV_Band by using the matrices in Table 2. We compare SpMV_Band with highly optimized SpMV kernel that runs on CPU cores with different storage formats such ELL, SELL and

Table 2. Input matrices used in our evaluation

ID	Input	Dimensions(N)	#nnz	#nnz/N
1	afshell10	1.51M	52.67M	34.65
2	apache2	0.72M	4.82M	6.74
3	atmosmodd	1.27M	8.81M	6.94
4	bone010	0.99M	47.85M	48.50
5	CoupCons3D	0.42M	17.28M	41.45
6	CurlCurl_2	0.81M	8.92M	11.06
7	ecology1	1.00M	5.00M	5.00
8	ecology2	1.00M	5.00M	5.00
9	Emilia_923	0.92M	40.37M	43.74
10	Fault_639	0.64M	27.25M	42.65
11	marine1	0.40M	6.23M	15.55
12	mc2depi	0.53M	2.10M	3.99
13	PFlow_742	0.74M	37.14M	50.00
14	t2em	0.92M	4.59M	4.98
15	tmt_unsym	0.92M	4.58M	5.00

CSR, which are denoted by ELL_CPU, SELL_CPU and CSR_CPU in Fig. 10, respectively. The CPU side codes run on four CPU cores in the GP area of FT-M7032 and SpMV_Band uses eight DSP cores, and the results are shown in Fig. 10.

We implemented three versions of SpMV_Band:

- Band_SIN uses a single DSP core to move x from DDR to GSM via dma_p2p;
- Band_MUL uses multiple DSP cores to move x from DDR to GSM;
- Band_MUL_DB uses multiple DSP cores to move x and further uses double buffer movement strategy to overlap data transfers with computations.

The performances of Band_SIN and Band_MUL are compared to show the impact of multi-core migration optimization. The only difference between these two versions is the use of multi-core migration optimization. On average, the overall performance gains from multi-core migration are limited. This is because SpMV_Band moves only one segment of x to GSM at a time. To evaluate the benefits of double buffering, Band_MUL is compared with Band_MUL_DB. According to the results in Fig. 10, Band_MUL_DB is on average 9% faster than Band_MUL.

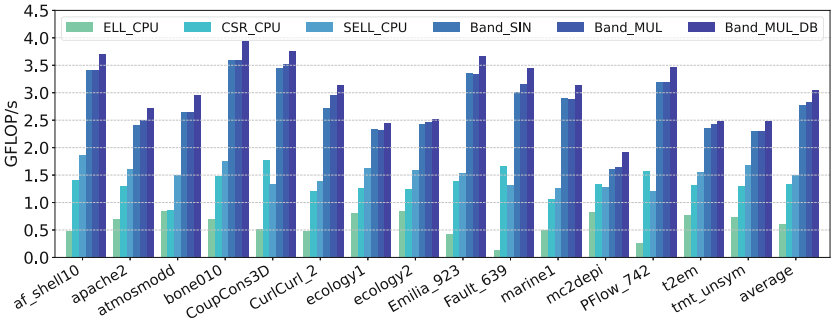


Fig. 10. Performance comparison of CPU-based SpMV vs SpMV_Band

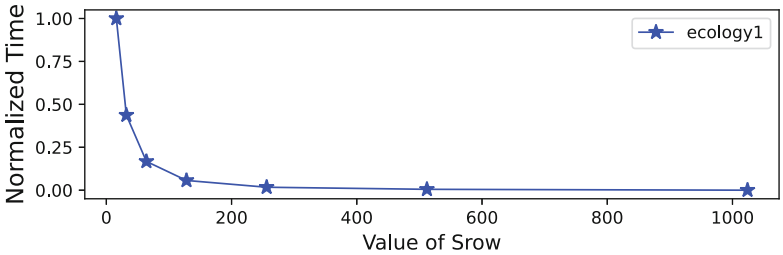


Fig. 11. The relationship between performance and the value of `Srow`. The vertical axis represents the normalized time. We normalize the time of ecology1 to a time range of 0 to 1 to make it easier to observe trends. Normalization formula is $time_normalized = (time - time_min)/(time_max - time_min)$.

SELL_CPU and CSR_CPU are generally superior to ELL_CPU for running CPU code in the GP area, with SELL_CPU typically showing the highest performance on average. The results in Fig. 10 show that the Band_MUL_DB implementation achieves an average speedup of 2.0× compared to the SELL_CPU implementation when running on the four CPU cores accessing the same DDR in a DSP cluster. And it is used as the final implementation of SpMV_Band.

Evaluation of the Parameter `Srow`. We evaluated the influence of the SELL format parameter `Srow` on the performance of the SpMV_Band algorithm. When `Srow` is small, the matrix is divided into many slices, resulting in higher latency costs (see Lines 25 and 26 of Algorithm 3). As `Srow` increases, more zero elements are filled, leading to higher bandwidth and computation costs. Thus, the time cost of the SpMV tends to decrease initially and then increase as `Srow` increases. However, the time cost is closely related to the sparse structure of the matrix, and different matrices may have different trends. The NNZ distribution of banded matrices is regular, such as matrix ecology1, with very few zero elements padding, the time cost generally decreases and then stabilizes as `Srow` increases (Fig. 11). In this work, we empirically set `Srow` to 512 as it gives good performance in our pilot studies.

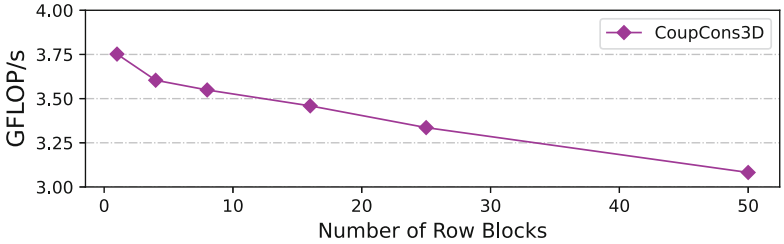


Fig. 12. The relationship between performance and the number of row blocks. As the number of row blocks increases, performance tends to decrease.

Evaluation of the Parameter `Beta`. SpMV_Band divides the matrix into row blocks and processes each row block one by one. We evaluated the impact of the number of row blocks on performance. We have conducted a number of experiments, which are illustrated here by the CoupCons3D matrix. Figure 12 shows that there is no performance benefit to increasing the number of partitioned row blocks. Therefore, our principle for choosing `blockLen` is to partition the matrix into as few row blocks as possible and to partition the matrix as evenly as possible, as long as the corresponding **x** segments can be stored in GSM. As can be seen in Fig. 7, the length of the **x** fragment corresponding to each row block is $\texttt{blockLen} + \texttt{SBL} + \texttt{SBU}$. The formula for choosing `blockLen` is as follows:

$$blockLenMax = \texttt{GSMlenMax} - SBL - SBU \tag{1}$$

$$num = \frac{n}{blockLenMax} \tag{2}$$

$$\texttt{blockLen} = \frac{n + num}{num + 1} \tag{3}$$

where n is the row dimension of sparse matrix. Since `blockLen` is equal to $\texttt{Srow} * \texttt{beta} * \texttt{coreNum}$, we can solve for `beta` as follows:

$$\texttt{beta} = \frac{\texttt{blockLen}}{\texttt{Srow} * \texttt{coreNum}}.$$

Bandwidth Utilization Analysis. Bandwidth utilization is an important evaluation criterion for memory-bounded operations and is calculated as:

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

where *Actual_bandwidth* is the actual bandwidth cost and *Stream_bandwidth* is the bandwidth measured by using the STREAM benchmark [24], which is modified by using `hthread` to run on FT-M7032. The total access memory of SpMV_Band equals to

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

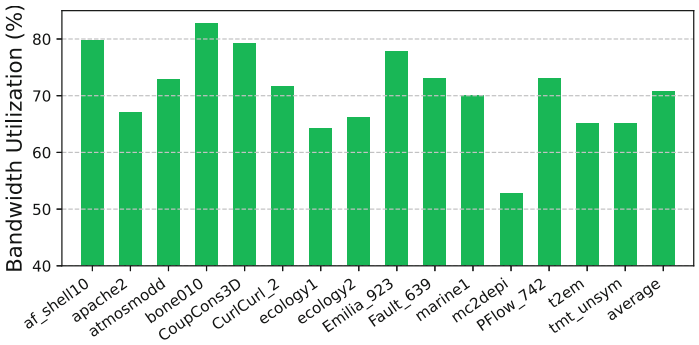


Fig. 13. Bandwidth utilization of SpMV_Band.

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.

Figure 13 shows the bandwidth utilization of the test matrices in Table 2 (Stream_bandwidth is measured by using the `dma_p2p` from DDR to AM, i.e., 30GB/s). The maximum and minimum bandwidth utilization ratio is 83% and 53%, respectively, with an average bandwidth utilization ratio of 71%.

5 Related Work

In this section, we present some of the current optimization methods for DSPs or multi-core platforms and some of the popular SpMV kernels. Prior works have optimized SpMV for CPUs [21,34] and GPUs [19,20,26,29] using storage formats like SELL-C-σ [14], SpV8 [17], CSR5 [19], ESB [21], HYB [2], HYB5 [4], and etc. Vectorization [3,14,35] and parallelization [1,25,28] approaches have also been proposed for SpMV.

Gao [8] et al. conducted a case study of SpMV on the TI C6678 DSP and found that DSP can achieve higher power efficiency than CPU and comparable power efficiency to GPU. Gao [9] et al. designed an on-chip scratchpad-based SpMV approach for the TI Keystone II DSP, improving performance by about 50%. Liu [23] et al. proposed a dense matrix multiplication vectorization method for a multicore long vector processor (MATRIX2). Sun [31] et al. introduced the SWCSR-SpMV, a method based on the CSR format, which effectively reduces the parallel SpMV bandwidth cost on the SW26010 platform. Xu [36] et al. proposed an adaptive sparse Matrix partitioning method based on the storage hierarchy of Matrix 2000b, which effectively improved the data reusability of SpMM.

Banded matrix has good data locality and can develop efficient SpMV operation with its own data locality. However, no one has yet optimized SpMV specifically for banded matrices. Yang [37] et al. proposed a parallel multi-core CSB format that has high scalability for banded matrices, but did not provide a specific

optimization algorithm. In this paper, we propose a SpMV algorithm specifically for banded matrices and evaluate its performance on the FT-M7032 architecture, achieving significant performance improvements over CPU-side SpMV.

6 Conclusion

In this paper, we design and implement an efficient SpMV on the FT-M7032. We evaluated the impact of GSM on the FT-M7032 on SpMV performance and provided a quantitative analysis. Based on the memory hierarchy of the FT-M7032, SpMV_Band can effectively improve the performance of SpMV operations through multi-layer partitioning and multi-level parallelism. We have designed a double buffering strategy on the FT-M7032 that overlaps data transfers and computation to minimize memory access overhead. Our experimental results show that our optimization method significantly improves the SpMV performance for banded matrices on the FT-M7032 platform. Compared to CPU-based SpMV implementations, an acceleration ratio of 2.0x can be achieved with an average bandwidth utilisation of 71%.

Acknowledgement. This work was supported in part by the National Key R&D Program of China under grant agreement 2021YFB0300101, the National Science Foundation of China (NSFC) under grant agreements 61902411, 62032023, 12002382, 11275269, 42104078 and 62073333, the Excellent Youth Foundation of Hunan Province under grant agreement 2021JJ10050.

References

1. Alappat, C., et al.: Performance modeling of streaming kernels and sparse matrix-vector multiplication on A64FX. In: IEEE/ACM PMBS, pp. 1–7. IEEE (2020)
2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11 (2009)
3. Chen, L., Jiang, P., Agrawal, G.: Exploiting recent SIMD architectural advances for irregular applications. In: IEEE/ACM CGO, pp. 47–58. IEEE (2016)
4. Chen, S., Fang, J., Xu, C., Wang, Z.: Adaptive hybrid storage format for sparse matrix-vector multiplication on multi-core SIMD CPUs. Appl. Sci. **12**(19), 9812 (2022)
5. Crane, H., Jr., Gibbs, N.E., Poole, W.G., Jr., Stockmeyer, P.K.: Algorithm 508: Matrix bandwidth and profile reduction. ACM Trans. Mathematical Softw. (TOMS) **2**(4), 375–377 (1976)
6. Davis, T., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011)
7. Fang, J., Zhang, P., Huang, C., Tang, T., Lu, K., Wang, R., Wang, Z.: Programming bare-metal accelerators with heterogeneous threading models: a case study of matrix-3000. Front. Inf. Technol. Electron. Eng. **24**(4), 509–520 (2023)
8. Gao, Y., Bakos, J.D.: Sparse matrix-vector multiply on the texas instruments c6678 digital signal processor. In: 2013 IEEE 24th ASAP, pp. 168–174. IEEE (2013)

9. Gao, Y., Zhang, F., Bakos, J.D.: Sparse matrix-vector multiply on the keystone ii digital signal processor. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2014)
10. Golub, G.H., Loan, C.F.V.: Matrix Computations, 3rd edn. The Johns Hopkins University Press, Baltimore, MD (1996)
11. Igual, F.D., Ali, M., Friedmann, A., Stotzer, E., Wentz, T., van de Geijn, R.A.: Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC. In: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 1–11. IEEE (2012)
12. Im, E.J., Yelick, K.: Optimization of sparse matrix kernels for data mining. In: Submitted to First SIAM Conference on Data Mining (2000)
13. Kincaid, D.R., Oppe, T.C., Young, D.M.: ITPACKV 2D user's guide. Technical Report, Texas University, Austin, TX (USA). Center for Numerical Analysis (1989)
14. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., Bishop, A.R.: A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. SIAM J. Sci. Comput. **36**(5), C401–C423 (2014)
15. Kubota, Y., Takahashi, D.: Optimization of sparse matrix-vector multiplication by auto selecting storage schemes on GPU. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011. LNCS, vol. 6783, pp. 547–561. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21887-3_42
16. Lewis, J.G.: Algorithm 582: The Gibbs-Poole-Stockmeyer and Gibbs-king algorithms for reordering sparse matrices. ACM Trans. Math. Softw. (TOMS) **8**(2), 190–194 (1982)
17. Li, C., Xia, T., Zhao, W., Zheng, N., Ren, P.: SpV8: Pursuing optimal vectorization and regular computation pattern in SpMV. In: 2021 58th ACM/IEEE Design Automation Conference (DAC), pp. 661–666. IEEE (2021)
18. Liu, S., Cao, Y., Sun, S.: Mapping and optimization method of SpMV on Multi-DSP accelerator. Electronics **11**(22), 3699 (2022)
19. Liu, W., Vinter, B.: CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In: 29th ACM ICS'15, pp. 339–350. ACM, New York (2015)
20. Liu, W., Vinter, B.: Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. Parallel Comput. **49**, 179–193 (2015)
21. Liu, X., Smelyanskiy, M., Chow, E., Dubey, P.: Efficient sparse matrix-vector multiplication on X86-based many-core processors. In: ICS'13, pp. 273–282. ACM, New York (2013)
22. Liu, Y., Schmidt, B.: LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In: 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 82–89. IEEE (2015)
23. Liu, Z., Tian, X.: Vectorization of matrix multiplication for multi-core vector processors. Chin. J. Comput. **41**(10), 2251–2264 (2018)
24. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25, December 1995
25. Merrill, D., Garland, M.: Merge-based parallel sparse matrix-vector multiplication. In: SC'16. Salt Lake (2016)
26. Mironowicz, P., Dziekonski, A., Mrozowski, M.: A task-scheduling approach for efficient sparse symmetric matrix-vector multiplication on a GPU. SIAM J. Sci. Comput. **37**(6), C643–C666 (2015)

27. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 111–125. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11515-8_10
28. Namashivayam, N., Mehta, S., Yew, P.C.: Variable-sized blocks for locality-aware SpMV. In: IEEE/ACM CGO, IEEE (2021)
29. Niu, Y., Zhengyang, L., Dong, M., Jin, Z., Liu, W., Tan, G.: TileSpMV: a tiled algorithm for sparse matrix-vector multiplication on GPUs. In: 35th IPDPS, pp. 68–78. IEEE (2021)
30. Saad, Y.: Iterative methods for sparse linear systems. In: SIAM (2003)
31. Sun, Q., Zhang, C., Wu, C., Zhang, J., Li, L.: Bandwidth reduced parallel SpMV on the SW26010 many-core platform. In: Proceedings of the 47th International Conference on Parallel Processing, pp. 1–10 (2018)
32. Tiwari, A., Kumar, V., Mitra, G.: High performance and energy optimal parallel programming on CPU and DSP based MPSOC. Ph.D. thesis, Ph. D. dissertation, IIIT-Delhi (2018)
33. Wang, Y., et al.: Advancing DSP into HPC, AI, and beyond: challenges, mechanisms, and future directions. CCF Trans. High Perform. Comput. **3**, 114–125 (2021)
34. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1–12 (2007)
35. Xie, B., Zhan, J., Liu, X., Gao, W., Jia, Z., He, X., Zhang, L.: CVR: efficient vectorization of SPMV on x86 processors. In: IEEE CGO (2018)
36. Xu, H., Zhu, X., Wang, Q., Liu, J.: Efficiently executing sparse matrix-matrix multiplication on general purpose digital single processor. In: 2022 IEEE 24th International Conferenct on High Performance Computing & Communications, pp. 1–8. IEEE (2022)
37. Yang, B., Gu, S., Gu, T.X., Zheng, C., Liu, X.P.: Parallel multicore CSB format and its sparse matrix vector multiplication. In: Advances in Linear Algebra & Matrix Theory, vol. 2014 (2014)
38. Yin, S., Wang, Q., Hao, R., Zhou, T., Mei, S., Liu, J.: Optimizing irregular-shaped matrix-matrix multiplication on multi-core DSPs. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER), pp. 451–461. IEEE (2022)
39. Zhang, Y., et al.: Memory-aware optimization for sequences of sparse matrix-vector multiplications. In: 37th IEEE International Parallel & Distributed Processing Symposium (IPDPS), IEEE (2023)
40. Zhang, Y., Li, S., Yan, S., Zhou, H.: A cross-platform SpMV framework on many-core architectures. ACM Trans. Archit. Code Optim. (TACO) **13**(4), 1–25 (2016)
41. Zhou, H., Fan, X., Zhao, L.: Optimizations on sparse matrix-vector multiplication based on CUDA. Comput. Meas. Control **18**(8) (2010)