Optimizing SpMV on Heterogeneous Multi-Core DSPs through Improved Locality and Vectorization

Deshun Bi
National University of Defense Technology
China
bds123@nudt.edu.cn

Shengguo Li
National University of Defense Technology
China
nudtlsg@nudt.edu.cn

Dezun Dong
National University of Defense Technology
China
dong@nudt.edu.cn

Peng Zhang
National University of Defense Technology
China
zhangpeng13a@nudt.edu.cn

Jianbin Fang
National University of Defense Technology
China
j.fang@nudt.edu.cn

ABSTRACT

The sparse matrix-vector multiplication (SpMV) is widely used in large-scale scientific computing and engineering. However, optimizing SpMV for high-performance digital signal processors (DSPs) has received limited attention. We present HALAV, a method to accelerate SpMV on CPU-DSP heterogeneous platforms, using the FT-M7032 DSP platform as a case study. HALAV partitions the input matrix into 'dense' and 'sparse' parts through column reordering. For the dense part, HALAV automatically selects storage formats optimized for vectorization to run on the DSP. At the same time, it offloads the sparse component to be processed by the CPU using the standard CSR algorithm. We evaluate our approach on the FT-M7032 platform and an Intel Xeon CPU. Experimental results show that our techniques achieve average speedups of 2.09× and 1.66× over the competing baselines on the FT-M7032 and the Xeon platform, respectively.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; • **Computing methodologies** → **Shared memory algorithms**; **Vector / streaming algorithms**.

KEYWORDS

SpMV, Heterogeneous Multi-Core DSP, Vectorization, Locality, Performance Optimization

ACM Reference Format:
Deshun Bi, Shengguo Li, Dezun Dong, Peng Zhang, and Jianbin Fang. 2024. Optimizing SpMV on Heterogeneous Multi-Core DSPs through Improved Locality and Vectorization. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673061

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPP '24, August 12–15, 2024, Gotland, Sweden
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1793-2/24/08.
https://doi.org/10.1145/3673038.3673061

1 INTRODUCTION

Sparse matrix-vector multiplication (SpMV)[1] is found in a variety of applications [7]. Despite its widespread use, optimizing SpMV on parallel architectures remains challenging due to its irregular indirect data accessing pattern [13]. With the increasing importance of power and energy consumption in HPC systems, high-performance digital signal processors (DSPs) are being integrated into HPC systems [6, 19, 21]. To the best of our knowledge, there has been limited research on optimizing general SpMV for these emerging multi-core DSP platforms [4, 5].

This paper aims to bridge the gap of optimizing SpMV on DSPs by taking the FT-M7032 CPU-DSP heterogeneous platform as a case study [1, 2]. The FT-M7032 DSP implements a sophisticated VLIW instruction set with SIMD units and software-managed on-chip scratchpad memory features which are widely used in other DSP architectures. Like other DSPs, memory management on FT-M7032 is bare metal, requiring the software to carefully utilize the memory space. As a result, optimization techniques developed for the FT-M7032 DSP can be applied to a wide range of other DSPs [4, 5, 21, 24].

Heterogeneous multi-core DSPs typically consist of two types of cores: a general-purpose CPU and a multi-core DSP. Taking full advantage of heterogeneous architectures is important for SpMV performance, but there is still little research in this area. The two processor cores' distinct computational and memory access capabilities make it non-trivial to utilize the processing units.

The DSP side offers extensive on-chip memory and robust vector processing capabilities, outperforming the CPU side. If the task is not divided properly, this architectural difference usually causes a load imbalance, and how to use the rich storage and computation resources on the DSP is still a problem. In addition, the highly irregular nature of sparse matrices makes it more difficult to develop efficient SpMV on heterogeneous processors.

To address the above challenges, we first capture the performance characteristics of the processor cores by running micro-benchmarks and then propose HALAV[2]. HALAV first divides the matrix into relatively 'dense' and 'sparse' parts using efficient column reordering

[1] An SpMV operation, $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$, multiplies a sparse matrix $\mathbf{A}$ of size $m \times n$ by a dense vector x of size n, and then produces a dense vector y of size m.
[2] HALAV =Heterogeneity and Locality-Aware Vectorization

algorithms. It then automatically selects vectorization-friendly storage formats according to the distribution of the elements in the dense part. With the diverse storage resources on the DSP, the dense part can achieve a high locality of memory access by using column blocking. The CPU side processes the sparse part using the standard CSR algorithm. The DSP and CPU sides are executed simultaneously to improve SpMV's computational efficiency. We evaluate our approach on FT-M7032, and experimental results show that compared with the SpMV implementation on the CPU cores of FT-M7032, HALAV using one cluster achieves up to 2.09× speedup. We also compare HALAV with the novel locality-aware vectorization (LAV) method [25] on FT-M7032, where HALAV showcases significant performance advantages. To fully exploit the performance of FT-M7032, we implement an MPI-level parallel SpMV, where each DSP cluster runs an MPI process. We utilize METIS [8] for load balancing across the four clusters. HALAV using four DSP clusters is competitive with the highly optimized MKL on a 12-core Intel Xeon E5-2692 processor, with an average speedup of 1.66×.

The main contributions of this paper are as follows:

- We propose a new partitioning scheme for heterogeneous multi-core DSPs, which partitions the matrix into dense and sparse parts through efficient column reordering.
- We propose HALAV, a new SpMV method for heterogeneous architecture. HALAV achieves high vectorization efficiency for dense parts through the adaptive selection of vectorization strategies. The sparse part using the CSR format is executed in parallel with the dense part.
- Experiments show that our approach achieves an average speedup of 1.66× over Intel's highly-optimized MKL library with low preprocessing overhead and good scalability.

2 BACKGROUND

2.1 FT-M7032 Heterogeneous Processor

The FT-M7032 is a CPU-DSP heterogeneous computing platform. It has 16 ARMv8 CPU cores, which run the operating system, and 4 multi-core DSP clusters. As shown in Figure 1(a), the FT-M7032 platform's CPU cores form the general-purpose area (GP), while the 4 clusters form the acceleration area (ACC). Each cluster has 8 DSP cores, a global shared memory (GSM), and off-chip DDR memory. The DSP cores operate at 1.8 GHz, compared to the 2.0 GHz of the CPU cores. The four DSP clusters can operate independently, with each cluster having access to its own private GSM. The CPU cores in GP can access the entire DDR space across different DSP clusters. The GSM is shared by the eight DSP cores within each cluster, which is 6 MB and can store about 780,000 double-precision elements. Data transfers between DSP clusters are managed by CPUs.

Figure 1(b) shows that the DSP core is a VLIW-based architecture. It comprises an instruction function unit (IFU), a scalar processing unit (SPU), a vector processing unit (VPU), and a DMA engine. The IFU can issue at most 11 instructions per cycle, comprising five scalar instructions and six vector instructions. The SPU, which includes an SPE (Scalar Processing Element) and an SM (Scalar Memory) with size of 64KB, is responsible for instruction flow control and scalar computations. The VPU has 768KB AM (Array Memory) and 16 VPEs (Vector Processing Elements), which work in

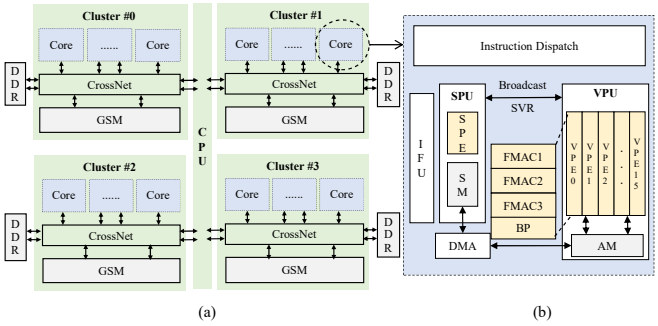


Figure 1: The overview of the FT-M7032 architecture (a) and the micro-architecture of a DSP core in FT-M7032 (b).

a SIMD fashion and can perform 1024 bits (16 FP64 data) at a time. To transfer data between the SPU and VPU, broadcast instruction and scalar-vector register (SVR) are utilized. Finally, the DMA engine is used to transfer data between different levels of memory, including main memory, GSM, and SM/AM.

2.2 SpMV with Vectorization and Locality

We review existing methods to inspire our designs, such as SELL [17], Sell-c-σ [10], ESB [15], and LAV [25] techniques, which are frequently used to implement SpMV vectorization and locality.

SELL. The SELL (Sliced ELLpack) format is a modified version of the ELL [9] format. It divides the sparse matrix into row slices of size c and each slice is stored in the ELL format. All rows in a slice are processed together with the same vector instructions, so if the rows in a slice have different numbers of nonzeros, they are padded to the same length. The slice size c is typically given by the width of the machine's vector unit and determines the degree of padding. When the number of elements in each row varies greatly, SELL will introduce more zero padding.

Sell-c-σ. Sell-c-σ initially groups σ consecutive rows of the matrix. Within each group, rows are reordered based on the descending order of their nonzero elements. This process aims to ensure that each set of c consecutive rows possesses a similar number of nonzeros, thereby reducing the zero padding.

ESB. ESB introduces two additional optimizations over Sell-c-σ: column blocking and the bit array. By setting the column block size to the cache size, the cache locality of the input vector is effectively improved. Using the bit array can reduce the required memory bandwidth, but it technically does not increase SIMD efficiency because ESB still performs arithmetic on padding zeros [15]. When the variance of the row lengths of the matrix is too large, ESB can hinder the efficiency of the overall vectorization.

LAV. LAV uses column reordering to improve the efficiency of vectorization. It reorders the columns based on decreasing numbers of nonzeros, this is called column frequency sorting (CFS) [25]. With CFS, columns with more nonzeros are processed in close temporal succession, reusing input vector elements. If the matrix is large, input vector elements are evicted from the LLC before being reused. Therefore, LAV partitions the matrix into column blocks after CFS. LAV divides the column block into dense and sparse parts using the fraction T (e.g., 80%), which denotes the percentage of nonzeros

in the dense part. LAV applies the global row reordering to each column block in the dense part, known as row frequency sorting (RFS) [25]. LAV first performs SpMV vectorization on the chunks (i.e. slices) of dense part and then processes the sparse part using the CSR algorithm.

3 ANALYSIS OF SPMV APPROACHES WITH HETEROGENEOUS DSPS

We find that on modern multi-core DSPs, the existing techniques sometimes are not faster than a simple CSR implementation on the CPU. In this section, we analyze this behavior on a representative FT-M7032 DSP processor in three ways: ***zero-padding minimization*** [10], ***column reordering*** [25], and ***blocking*** [15, 25].

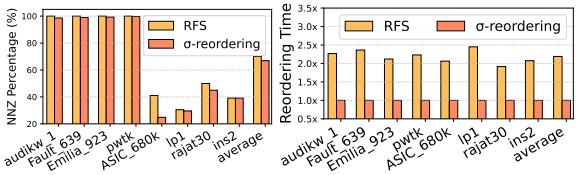
3.1 Architecture Analysis

Figure 1 shows that DSP memory architecture has three main levels: an off-chip DDR, an on-chip GSM shared by DSP cores, and AM/SM private to a DSP core. The `values` array of nonzeros and vector x must be loaded into AM before performing vectorization computation on VPUs. GSM is high-speed on-chip memory serving as a shared LLC cache, which can cache vector x to improve locality. However, GSM's limited memory means that it can't hold the entire vector x if x exceeds GSM's capacity. To perform SpMV calculations for arbitrary matrices using GSM, we employ column ***blocking***. The cache locality of the input vector corresponding to each column block is improved by transferring it into GSM. The DMA provides two data transfer modes: point-to-point mode (dma_p2p) and super-gather mode (dma_sg) [14]. While dma_p2p is a non-blocking direct transfer, dma_sg specializes in gathering operations on x.

3.2 Zero-padding Minimization

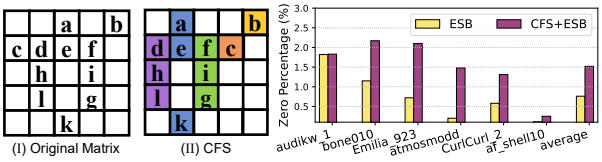
To improve the efficiency of SIMD vectorization, zero padding should be reduced as much as possible. The SELL format uses row slicing to reduce zero padding, and the sell-c-σ and LAV use row reordering to reduce it further. The sell-c-σ uses local reordering in the σ range, while the LAV uses a global row reordering (RFS). In matrices for power-law graphs, the number of nonzeros in both rows and columns is highly imbalanced. RFS can effectively reduce the zero padding for such matrices, but it may be harmful to sparse matrices used in scientific computing. This is because all rows are reordered, which typically results in a loss of spatial and temporal locality for the input vector, as nearby rows tend to have similar nonzero patterns [15, 26].

Figure 2 shows the comparison results of RFS and σ reordering, with c set to 16 and σ set to 512. Figure 2(a) shows the percentage of **nonzeros** in SELL after different row reordering, and Figure 2(b) shows the overhead of RFS relative to σ reordering. We use the coefficient of variation [23] (CV = Standard Deviation / Mean Value) to measure the difference in row lengths of sparse matrices. The first four matrices have smaller CVs and the last four are larger in Figure 2. As shown in Figure 2(a): when CV is small, indicating a balanced number of elements per row, RFS exhibits only slight competitiveness over σ reordering. When CV is large, indicating an imbalance in the number of elements per row, RFS introduces significant zero padding. However, as shown in Figure 2(b), the



(a) (b)

Figure 2: Comparison of row reordering methods. Figure 2(a) shows the percentage of nonzeros in SELL after different row reordering, and Figure 2(b) shows the overhead of RFS relative to σ reordering.



(a) (b)

Figure 3: Impact of CFS on zero-padding. Figure 3(a) illustrates the CFS and Figure 3(b) shows its effect on ESB's zero padding.

overhead of RFS is several times that of local reordering. Thus, RFS does not offer significant advantages over σ local reordering.

Although the article [26] claims that Sell-c-σ effectively accelerates matrices in SuiteSparse, Figure 4 in the same article reveals that Sell-c-σ is not always the fastest method. Similarly, Figure 2(a) shows that for matrices with larger CVs, Sell-c-σ still produces significant zero padding, which affects SpMV performance. The inadequacy of these current methods motivates us to search for new vectorization methods.

3.3 Column Reordering

We evaluate the impact of column reordering on SpMV performance. Column reordering is often used in conjunction with column ***blocking***. This can concentrate nonzeros in the first few column blocks to increase locality and vectorization efficiency. However, our findings indicate that CFS does not always improve SpMV performance. Using the ESB format with two column blocks as an example, Figure 3(b) illustrates the **zero** percentage in ESB and indicates that CFS can lead to increased computational tasks for certain matrices. This is because CFS breaks the original locality in the matrix, causing the zero padding to increase instead of decrease.

LAV stores the sparse part of the matrix in CSR after CFS and ***blocking***, reducing the zero padding in the sparse part without reducing it in the dense part. It is worth noting that LAV has no heterogeneous parallelism for dense and sparse parts stored using different formats. The dense part is processed first before the sparse part [25], which reduces the performance of SpMV to some extent.

3.4 Mask

ESB and LAV both use the mask load instruction to reduce bandwidth requirements. However, the slice still needs to pad zero to achieve computational alignment of SIMD lanes for the vectorization [15, 25]. Therefore, the mask technique does not reduce zero padding to improve vectorization efficiency, and introduce additional control logic. Since many matrices in scientific computing

have a balanced distribution of nonzeros to rows [26], there is no need to use additional masks for such matrices. The FT-M7032 currently lacks the mask load interface, so our work does not use the masking technique. If the number of rows in a matrix is not a multiple of the number of SIMD lanes, zero padding is usually required to implement vectorization for the last slice. LAV computes the row_mask [25] to identify zero rows. For this case, we propose a new matrix partitioning scheme that avoids zero padding and row_mask calculation, as described in Section 4.2.

4 HALAV: HETEROGENEITY AND LOCALITY-AWARE VECTORIZATION

The analysis above highlights the shortcomings of existing vectorization methods in addressing heterogeneous processors, posing challenges in task distribution and vectorization efficiency. Drawing from these insights, we introduce HALAV and delve into the detailed design concept of HALAV for a single DSP cluster.

4.1 Design Overview

HALAV is a new approach that speeds up SpMV on heterogeneous multi-core DSPs by enhancing locality and vectorization. HALAV proposes a new partitioning of heterogeneity-aware. It divides the matrix into dense and sparse parts using a lightweight and efficient preprocessing step. The DSP in the cluster and the CPU in GP compute the dense and sparse parts, respectively. The dense part automatically selects vectorization-friendly formats based on the distribution of nonzeros, which effectively improves the computational efficiency of SIMD lanes. The sparse part uses the CSR algorithm without additional zero padding. The sparse and dense parts are executed simultaneously, efficiently exploiting the performance of heterogeneous processors.

4.2 New Partition for Heterogeneity-Aware

To establish the foundation for our partition design, we conduct a basic micro benchmark to test the performance of the DSP and CPU side. We implement SpMV vectorization on the DSP side in the ESB, because it uses column blocking, which can utilize GSM to cache x. The CPU side is implemented using CSR. Figure 4 shows that the performance on the DSP side is much higher than on the CPU side, due to the DSP obvious vectorization advantage. The performance difference of these two types of cores motivates us to find a new partitioning. We propose a heterogeneity-aware new partitioning by implementing *an efficient CFS* and *two-level partitioning*: preliminary and heterogeneous partitions.

CFS concentrates the densest columns to the left, which reduces zero padding on the DSP side and boosts vectorization efficiency. To achieve CFS of a matrix, it is necessary to create a `sortedCols` array to record the sorted column index. The most time-consuming part in CFS is to query the sorted new column index based on the old column index, and the traditional query process as shown in Lines 15 to 19 of Algorithm 1. In the worst case, the complexity of this linear search for a nonzero column index is O(N). When the matrix is large, it will take up more time. To address this problem, we propose MapCFS, an efficient CFS based on map arrays.

MapCFS. MapCFS accelerates the query by constructing a map array, `colIndexMap` (Lines 1 to 3 in Algorithm 1). The map array

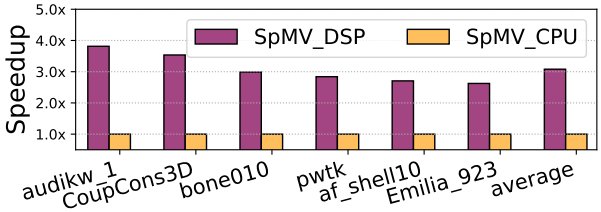


Figure 4: DSP SpMV performance relative to CPU. The DSP side uses 8 DSP cores (one cluster) for parallel computing, and the CPU uses 4 cores corresponding to one DSP cluster.

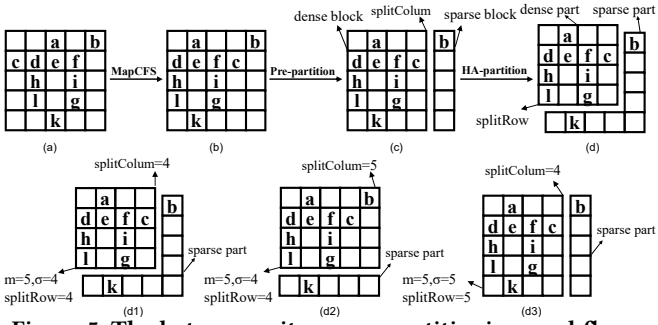


Figure 5: The heterogeneity-aware partitioning workflow.

Algorithm 1: MapCFS Algorithm

```
   Input: matrix A, sortedCols.
   Output: newColIndices, newValues.
1  for k = 0 to A.numCols do
2  |  colIndexMap[sortedCols[k].colIndex] = k
3  end
4  for i = 0 to A.numRows do
5  |  rowStart = A.row_ptr[i], rowEnd = A.row_ptr[i + 1]
6  |  for j = rowStart to rowEnd do
7  |  |  oldColIndex = A.colIndices[j]
8  |  |  newColIndex = colIndexMap[oldColIndex]
9  |  |  // Update column index.
10 |  |  newColIndices[j] = newColIndex
11 |  |  newValues[j] = A.values[j]
12 |  end
13 end
14 //Appendix: Traditional method of querying the new column index.
15 for k = 0 to A.numCols do
16 |  if sortedCols[k].colIndex == oldColIndex then
17 |  |  newColIndex = k, break
18 |  end
19 end
```

stores the map from the old column index to the new column index, i.e. the old index finds the new index (Line 8). This method avoids expensive linear searches for each nonzero and can reduce the time complexity to O(1). In addition, MapCFS works directly with the CSR format (Lines 4 to 13). Since CSR is the default format in many sparse matrix applications, MapCFS has high applicability and low preprocessing overhead.

Preliminary partition. Blocking takes the output of MapCFS and divides the matrix into dense block and sparse block. The dense block contains column blocks with dense columns. The standard for selecting the number of column blocks (N_d) that make up the dense block is to pre-balance the DSP and CPU task divisions,

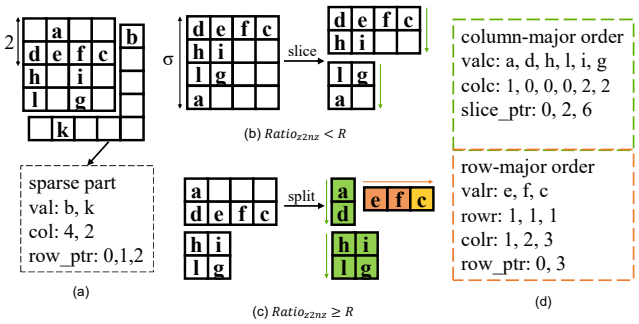


Figure 6: The layout of the dense and sparse part in HALAV.

and we will discuss it in Section 5.2.3. After using the blocking strategy, we implement the initial partitioning of the matrix for the heterogeneous processors and output a "`splitColumn`" index. `splitColumn` is a column that distinguishes between dense and sparse blocks. The column blocks in dense block will use GSM caching to ensure the locality of the data.

Heterogeneous partition. Based on the results of the preliminary partition, we further divide the dense block. We compare the number of rows in the matrix with σ. If m is not a multiple of σ, we take $m - m\%^3\sigma$ as the "`splitRow`" index. We use `splitRow` to split the dense block into the dense part above, and the remaining part together with the sparse block form the sparse part. This ensures that the number of rows in the dense part is a multiple of the number of SIMD lanes, and avoids using row_mask to mark empty rows.

Figure 5 shows an example of the HALAV partition. The first step performs MapCFS on A to attain Figure 5(b). Note that the input vector x is also permuted according to the new order in `sortedCols`. After MapCFS, most of the nonzeros are now in the first few columns. The next step divides A into column blocks. The number of columns per block, `blocklen`, is the size of GSM in practice. The `splitColumn`, the multiple of blocklen, is chosen to partition matrix into dense block and sparse block. The dense block consists of the first $\frac{splitColumn}{blocklen}$ column blocks, which contain the majority of the nonzeros. The sparse block contains the remaining columns. Figure 5(c) shows the dense and sparse blocks of A with `blocklen` = 4 and `splitColumn` = 4, where a single column block is sufficient to form the dense block. Finally, by calculating `splitRow`, the dense block is split into the dense part above, and the remaining part together with the sparse block form the sparse part, as shown in Figure 5(d). Figure 5(d1-3) shows that when `splitRow` and `splitColumn` take different values, the sparse part can correspond to different shapes. This allows dynamic adaptation of partitioning styles based on matrix dimensional properties.

4.3 HALAV Matrix Representation

HALAV uses different representations for dense and sparse parts to facilitate subsequent SpMV parallel execution. To utilize the vector components of DSP, we store the dense part in a format that is easy to vectorize. For the sparse part, we adopt CSR format.

[3]Here, the % represents the modulo operation.

Algorithm 2: Search for `regularlen` in the irregular slice

Input: $Srow$, $slicelen$, and $rowlen_slice$.
Output: $regularlen$.

```
regularlen = slicelen, presize = slicelen * Srow * nnzr_size
for len = slicelen − 1 to 0 do
    nnzi_num = 0
    for i = 0 to Srow do
        if rowlen_slice[i] ≥ len then
            nnzi_num+ = rowlen_slice[i] − len
        end
    end
    nextsize = nnzi_num * nnzi_size + len * Srow * nnzr_size
    if nextsize >= presize then
        regularlen = len + 1, break
    end
    presize = nextsize
end
```

4.3.1 The dense part representation. There are two commonly used strategies for vectorization: in-row vectorization of row-major order and cross-row vectorization of column-major order [11]. In-row vectorization processes same-row values using SIMD units. If the row length is not multiple of SIMD width, the outliers are computed separately. The cross-row vectorization uses SIMD lanes to process adjacent rows in the same slice simultaneously and does not require any additional operations for reduction and outliers. When adjacent rows in a matrix have similar lengths, cross-row vectorization is more efficient, and all formats in Section 2.2 adopt cross-row vectorization. However, cross-row vectorization tends to underperform when row lengths vary widely, as excessive padding zeros lead to performance penalties.

Therefore, the format in Section 2.2 is not always efficient when the distribution of row lengths is uneven. This inspired us to explore new vectorization formats. After heterogeneous partition, HALAV uses $Ratio_{z2nz} = \frac{DenseZero}{DenseNNZ}$ to measure the storage efficiency when dense part uses cross-row vectorization with σ row reordering. Here, $DenseNNZ$ and $DenseZero$ represent the number of nonzeros and the zero-padding in the dense part. Note that obtaining $DenseNNZ$ and $DenseZero$ does not require physical reordering of rows. HALAV can compute these values by simply counting the nonzeros in each row. HALAV uses $Ratio_{z2nz}$ to automatically select vectorization strategies for the dense part.

If $Ratio_{z2nz}$ is less than the threshold R, it indicates that cross-row vectorization is more efficient and there is no need to consider in-row vectorization. In this case, HALAV first performs continuous σ row reorder on each column block and then divides each block into slices. Figure 6(b) continues the example of Figure 5(d) and shows the block divided into two slices. We use two rows per slice because we assume two SIMD lanes. Afterward, vectorization calculation is performed on each slice using column-major order.

If $Ratio_{z2nz}$ is greater than or equal to the threshold R, it means that the sparse matrix has rows of "special" length, which results in more zero-padding. For example, the second row in Figure 6(c). This significantly affects the efficiency of cross-row vectorization. HALAV introduces in-row vectorization to deal with such rows. Firstly, HALAV directly divides each block into slices, eliminating row reordering since it does not effectively reduce zero padding. Secondly, HALAV uses CV to evaluate the row length distribution of each slice. If the CV is less than 0.2 [18], the distribution is balanced,

indicating that the slice is more regular and stored in column-major order (see the second slice in Figure 6(c)). Otherwise, the row length distribution is unbalanced, thus this slice is irregular (see the first slice in Figure 6(c)). In this case, HALAV first finds the more regular part of the irregular slice and stores it using column-major order. The remaining irregular part is stored using row-major order. Note the regular part of a slice can be the whole slice (CV < 0.2) or a part of the slice (CV >= 0.2). Finally, HALAV appends the regular parts of each slice together and stores them in a column-major order data structure. The irregular parts are appended together and stored in a row-major order data structure. This is shown in the green and orange boxes in Figure 6(d).

HALAV finds the length of the regular part (`regularlen`) in the irregular slice using Algorithm 2. `Srow` is the slice size, similar to the c in SELL, and `rowlen_slice` array stores the number of nonzeros of each row in the slice. For irregular slice, if the `regularlen` is large, such as equal to the original length of the slice (`slicelen`), it will result in more zero padding. If the `regularlen` is small, too many nonzeros will be split into irregular parts. Because irregular parts use in-row vectorization, there will be additional processing overhead. Therefore, Algorithm 2 tries to preserve the regular parts of the slice as much as possible, while reducing zero padding. HALAV stores the regular part in column-major order, only storing column indexes and values. Each nonzero occupies a storage size of: $sizeof(double) + sizeof(int)$, which we use `nnzr_size` to represent. The irregular part needs to store the row and column indexes and values of nonzeros, occupying a space of: $sizeof(double) + 2 * sizeof(int)$. We use `nnzi_size` to represent. It is easy to see that each element of the irregular part requires more storage space. Only columns with more zeros are divided into irregular parts, resulting in a lower storage overhead for irregular slice. Therefore, Algorithm 2 searches for the most appropriate `regularlen` based on the slice storage cost. Line 1 initializes the `regularlen` to the `slicelen`. Lines 3 to 9 calculate the slice storage size (`nextsize`) for each length, including both regular and irregular parts. Searching one by one until the nonzeros are added to the irregular parts can no longer reduce the slice storage size (Lines 10 to 12). The `regularlen` that balances zero padding and in-row vectorization overhead is found.

Figure 6(c) shows the HALAV processing when $Ratio_{z2nz}$ is greater than the threshold R. For the regular part, the cross-row vector computation is taken directly. For the irregular part, the in-row vector computation is taken and the computed result is accumulated at the corresponding position of the vector y according to the index of the current row. If a row length of irregular part is not a multiple of the number of SIMD lanes, the remaining elements in each row are spliced. A vector computation is performed on the spliced data and the computed result is scattered to the corresponding position of vector y according to the row index of each element. If the remaining data is not sufficient for the number of SIMD lanes, the scalar computation is directly performed. For instance, in Figure 6(c), the yellow nonzero `C` represents such a case. HALAV dynamically adjusts the vectorization strategy according to the distribution of matrix elements, which effectively reduces zero-padding and improves the vectorization efficiency.

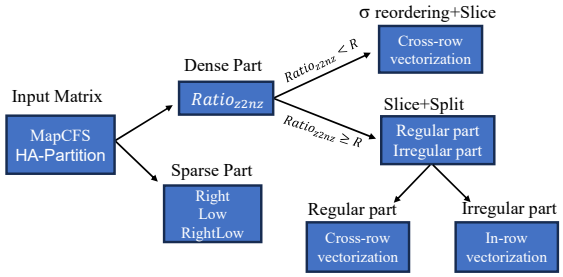


Figure 7: The HALAV workflow.

4.3.2 The sparse part representation. As shown in Figure 5(d1-3), the sparse part includes three types: RightLow, Low, and Right. If m is not a multiple of σ, a Low type is generated. When the number of columns in the matrix is greater than 780,000 (see Section 2.1), a Right type may occur. When both situations exist simultaneously, the sparse part is the RightLow type. Different types of sparse parts correspond to different numbers of rows. The number of rows for the Right and RightLow types is the same as the original matrix. However, for the Low type, there are only $m\%\sigma$ rows. HALAV designs a unified CSR-based SpMV algorithm for sparse parts, which can efficiently execute SpMV using OpenMP. To ensure that the different types are calculated correctly, a starting row needs to be recorded. The starting row for the Right and RightLow types is 0, while for the Low type is $m - m\%\sigma$.

4.4 HALAV's Workflow

Figure 7 shows the HALAV workflow. Firstly, HALAV uses MapCFS to preprocess the input matrix, Afterward, the heterogeneity-aware partition is used to divide the matrix into dense and sparse parts. The sparse part may contain three types, and HALAV uses a unified CSR algorithm for calculation on CPU. HALAV calculates $Ratio_{z2nz}$ of the dense part for comparison with the threshold R. If it is less than the R, HALAV reorders the consecutive σ rows for each column block and uses cross-row vectorization on each slice. If it is greater than the R, HALAV performs row slicing on dense parts and splits each irregular slice. The regular part is calculated using cross-row vectorization, while the irregular part uses in-row vectorization. The dense and sparse parts execute simultaneously to enhance SpMV performance.

4.5 SpMV Algorithm with HALAV

In this section, we introduce the implementation of HALAV's SpMV and use optimization strategies such as: *reducing startup overhead* and *thread-level load balancing* to improve SpMV performance.

4.5.1 Reducing startup overhead. Transferring the `values` and vector x involves calls to dma_p2p and dma_sg (see Section 3.1). A transfer process involves startup and transfer overhead. The transfer overhead remains constant once the data volume is fixed, but excessive calls can significantly increase the startup overhead [2]. To perform SpMV on the AM (768KB), we need to partition it into at least three parts to store x, `A.values`, and the calculated y. Based on the size of AM, we can figure out that the maximum length of `value` and x each transmission is: $\frac{768*1024*8}{(64*3)} = 32768$, which is

about $30 * 1024$. When using in-row vectorization, it is necessary to scatter the calculated vector result to the corresponding position of y based on the nonzeros row index. We use the SVR register to complete the scatter process on SM (64KB). SM needs to store the row index and the calculated y. So, the maximum transfer length of row indexes is $\frac{64*1024*8}{(64+32)} = 5461$, about $40 * 128$.

4.5.2 SpMV_HaLAV. HaLAV's SpMV algorithm consists of two parts: cross-row vectorization and in-row vectorization, which are used to handle regular and irregular parts of slices, respectively. Algorithm 3 demonstrates the SpMV_HaLAV algorithm. To make it concise, scalar calculations in the irregular parts are not shown here. It works by operating on one column block at a time (Line 2), processing slices of the block in thread-parallel. The vector x is transferred from the DDR to the GSM using the dma_p2p before the SpMV operation (Line 3). After moving the vector x, the indexes of the start and end slices that each thread is responsible for processing are obtained (Lines 5 and 6). After that, each thread processes its slices (including regular and irregular parts) in parallel.

Lines 8 to 23 show the cross-row vectorization algorithm. First, we use dma_p2p and dma_sg to load matrix values and vector x into AM (Lines 11 and 12). Then, the transmitted data is converted into columns, these columns of length `Srow` come from multiple slices (Line 13). `Srow` refers to the slice size and is set to the number of SIMD lanes in the DSP, which is 16 (`vec_len`). Since the length of the i-th slice can be calculated using $\frac{(slice_ptr[i+1]-slice_ptr[i])}{Srow}$. The lengths of each slice are recorded in advance in the `slicelen` array. Afterward, the vectorization of the column-major order is computed for each slice. `slice_id` is the index of the current slice (Line 13). `precol` and `curcol` are the cumulative number of columns from the start slice to the previous slice and the current slice respectively (Line 14). Lines 16 to 18 show the computation process for the current slice, where 16 double-type elements can be calculated by VPU each time. When the calculation is complete, the results are written back to DDR (Line 22).

Lines 25 to 44 show the in-row vectorization algorithm. Line 25 gets the start and end row numbers that each thread is responsible for. Lines 29 and 30 transfer matrix values and vector x of irregular parts to AM. And convert the data length into the number of vector calculations (Line 32). Then we transfer the row index to SM and calculate the corresponding number of vectors (Lines 33 and 36). Lines 38 to 41 show the calculation of the `currVecNum` vectors and scatter the results to SM. When the calculation is complete, the results are stored back in DDR (Line 43).

4.5.3 Thread-level load balancing. To allow the DSP cores to achieve better thread-level parallelism, we propose a load-balancing strategy with the granularity of slices. Since the dense part stored in HaLAV format consists of multiple slices, distributing the slices across the DSP cores according to the number of slices can result in a serious load imbalance. To solve this problem, we propose a linear scanning allocation (LSA) algorithm, which can make the slices computed by each core have an approximate number of elements.

Algorithm 4 describes the LSA algorithm. To make it more concise, only the regular slice is considered here. Lines 1 to 19 iterate through each column block in turn. Lines 3 and 4 indicate the `start` and `end` positions of the current column block in `slice_ptr`.

Algorithm 3: SpMV_HaLAV Algorithm

```
   Input: matrix A, vector xv, Srow, and coreNum.
   Output: vector yv.
1  Function SpMV_HaLAV (A, xv, Srow, coreNum)
2      for bi = 0 to numblocks do
3          dma_p2p: transfer xv[bi] from DDR to GSM
4          // tid indicates the thread index. tid ∈ (0,...,coreNum-1).
5          start = thread_ptr[tid + bi * coreNum]
6          stop = thread_ptr[(tid + 1) + bi * coreNum]
7          // Cross-row vectorization handles the regular part in slices
8          for sli = start to stop in parrall do
9              off = A.slice_ptr[sli]
10             // The transmission length is amlen: 30 × 1024.
11             dma_p2p: transfer A.valc[off] from DDR to AM
12             dma_sg: transfer xv from GSM to AM by A.colc[off]
13             slice_id = sli, colnum = amlen/Srow
14             precol = 0, curcol = slicelen[slice_id]
15             while curcol <= colnum do
16                 for j = precol to curcol do
17                     vec += vec_muli(val_am[j], xv_am[j])
18                 end
19                 yv_am[slice_id − sli] = vec
20                 precol = curcol, curcol += slicelen[++slice_id]
21             end
22             dma_p2p: transfer &yv[sli*Srow] from AM to DDR
23         end
24         // In-row vectorization handles the irregular part in slices
25         rowstart=start × Srow, rowstop=stop × Srow
26         for rowid = rowstart to rowstop in parrall do
27             off = A.row_ptr[rowid]
28             // The transmission length is amlen: 30 × 1024.
29             dma_p2p: transfer A.valr[off] from DDR to AM
30             dma_sg: transfer xv from GSM to AM by A.colr[off]
31             //vecNum: number of vector calculations
32             vecNum = amlen/vec_len
33             rowoff = 0, currVecNum = smlen/vec_len
34             for j = 0 to vecNum do
35                 // The transmission length is smlen: 40 × 128.
36                 dma_p2p: transfer A.rowr[rowoff] from DDR to SM
37                 rowoff += smlen
38                 for k = 0 to currVecNum do
39                     vec +=vec_muli(val_am[j + k], xv_am[j + k])
40                     scatter(vec, rowr_sm[k * vec_len], yv_sm)
41                 end
42             end
43             dma_p2p: transfer yv from SM to DDR
44         end
45     end
46 end
```

`cblocknnz` represents the number of elements in the current column block, while `corennz` is the average number of elements that each thread ideally processes (Lines 5 and 6). Lines 7 to 17 iterate through each slice in the current column block to find the starting slice for each thread. The `bound` indicates the starting boundary of the thread's processing range under ideal load conditions [2] (Line 8). Lines 9 to 16 find the slice index closest to the starting boundary and this slice as the starting slice for the current thread. Line 15 saves the slice index to an array and continues to find the starting slice for the next thread. Line 18 records the end position of the last thread's processing range. The `thread_ptr` array records the index of the starting slice processed by each thread, and the starting slice of the next thread is the ending slice of the current thread. After that, each thread gets the range of slices to process by accessing the `thread_ptr` array (Lines 5 and 6 in Algorithm 3). In this way, each

Algorithm 4: LSA Algorithm

Input: matrix A, $coreNum$.
Output: $thread_ptr$.

```
for bi = 0 to numblocks do
    // rslice: number of slices in each column block.
    start = A.slice_ptr[bi * rslice]
    end = A.slice_ptr[(bi + 1) * rslice]
    cblocknnz = end − start
    corennz = cblocknnz/coreNum, tid = 0
    for sli = 0 to rslice do
        bound = start + tid * corennz, preslice = bi × rslice
        if A.slice_ptr[preslice + sli] ≥ bound then
            d1 = |A.slice_ptr[preslice + sli] − bound|
            d2 = |A.slice_ptr[preslice + sli − 1] − bound|
            if d1 ≥ d2 then
                sli − −
            end
            thread_ptr[tid + bi × (coreNum + 1)] = sli, tid + +
        end
    end
    thread_ptr[coreNum + bi × (coreNum + 1)] = rslice
end
```

thread processes a similar number of nonzero elements to achieve load balancing, effectively improving the SpMV performance.

4.6 Four-Clusters HALAV Design

This section shows how to apply the HALAV algorithm to four clusters. Since the FT-M7032 are distributed architectures, the initial step involves matrix partitioning. A common method partitions rows into equal-sized row blocks, assigning one to each cluster. While simple, this approach often leads to load imbalance. To generate a more balanced load between clusters, we need to perform an equal partitioning of the matrix nonzeros.

We use graph partitioning-based reordering, which has been proven to be the best for SpMV performance among different reordering methods [20]. Graph partitioning can be used to define ordering by partitioning a matrix into a given number of parts, and then grouping rows and columns according to assigned parts. We design a partitioning method based on vertex weights using METIS [8], to balance the number of nonzeros within the parts. This is achieved by weighting each vertex in the graph by the number of nonzeros in the corresponding row. To better execute SpMV on the divided parts, we implement an MPI-level parallel SpMV, with each DSP cluster running an MPI process. The MPI processes on each cluster call the SpMV_HALAV algorithm described earlier.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

Table 1 lists the matrices from the SuiteSparse Collection [3] used in our evaluation. These matrices vary in size and sparsity, which were extracted from different application areas and can represent a wide range of inputs for SpMV in HPC applications. We benchmark HALAV against traditional CSR SpMV on GP area CPU cores and further evaluate its performance by implementing vectorized ESB, and LAV on the FT-M7032. Since Sell-c-σ can be seen as a subset of ESB, it was not selected. In addition, to better analyze matrices with large $Ratio_{z2nz}$, we also considered the SpMV performance of In-row vectorization (In-row in Figure 8(a)). The CPU cores in the GP area run the Linux kernel version 5.4.0, and the CPU codes are compiled with GCC version 9.3.0 and "-O2 -lpthread -fopenmp" options. For the performance of the four clusters, we compare HALAV with the highly optimized MKL library on a 12-core Intel Xeon E5-2692 processor. The E5-2692 processor runs at 2.20GHz and has 64KB of L1 cache per core, 256KB of L2 cache per core, and 30MB of L3 cache shared by 12 cores. The MKL (version 2018.0.0) codes on E5-2692 are compiled using the Intel compiler ICC version 18.0.0 with the options "-O3 -ffast-math -qopenmp -march=native".

Table 1: Input matrices used in our evaluation

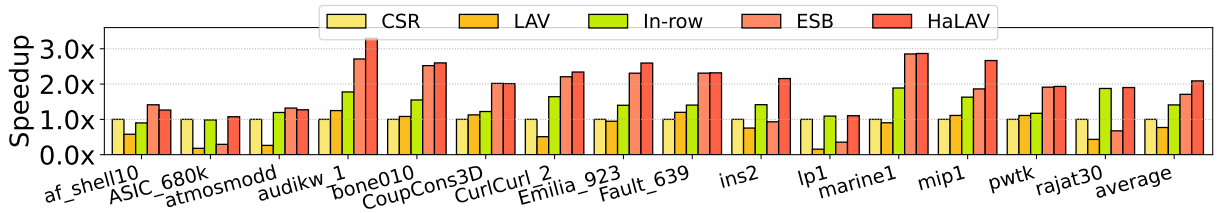
ID	Input	Rows(N)	#nnz	#nnz/N	$Ratio_{z2nz}$
1	af_shell10	1.51M	52.67M	34.93	0.22%
2	ASIC_680k	0.68M	3.87M	5.67	302.39%
3	atmosmodd	1.27M	8.81M	6.94	1.21%
4	audikw_1	0.94M	77.65M	82.28	1.57%
5	bone010	0.99M	71.67M	72.63	1.64%
6	CoupCons3D	0.42M	22.32M	53.56	0.45%
7	CurlCurl_2	0.81M	8.92M	11.06	0.97%
8	Emilia_923	0.92M	41.01M	44.42	1.50%
9	Fault_639	0.64M	28.61M	44.79	1.05%
10	ins2	0.31M	2.75M	8.89	156.21%
11	lp1	0.53M	1.64M	3.08	238.00%
12	marine1	0.40M	6.23M	15.55	0.83%
13	mip1	0.07M	10.35M	155.77	17.94%
14	pwtk	0.22M	11.63M	53.39	0.27%
15	rajat30	0.64M	6.18M	9.59	122.29%

The In-row, ESB, and the dense parts of LAV and HALAV in Figure 8(a) utilize eight DSP cores, thus a single DSP cluster. The sparse parts of LAV, HALAV, and the CSR run on the 16 CPU cores in the GP area. To better utilize the on-chip cache, the column block length `blocklen` is set to the size of GSM (780,000 i.e. 0.78M). The slice size in ESB, LAV, and HALAV is 16, which is the vector length of FT-M7032. In our method, we experimentally set σ to 512 and R to 15% based on our findings that these two values generally yield better performance. Since the article [25] did not provide a criterion for choosing T in LAV, only setting T at 80%, we chose the partition with the highest LAV performance to make the comparison fairer. We also use optimization techniques such as reducing startup overhead and thread-level load balancing for In-row, LAV, and ESB.

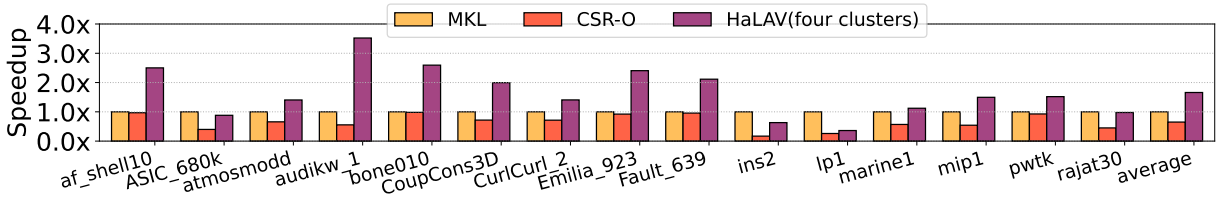
5.2 Performance Evaluation

5.2.1 Performance of HALAV. In this section, we assess the performance of HALAV on a single DSP cluster by utilizing all the matrices provided in Table 1. The Figure 8(a) compares the speedup of In-row, ESB, LAV, and HALAV over CSR. HALAV delivers an average speedup of 2.09× over standard CSR. In-row and ESB also provide significant average speedups of 1.41× and 1.71× respectively. However, the performance of LAV is inferior to CSR. This is because LAV is mainly applied to skew matrices of dense law graphs, and the speedup is not significant for matrices on SuiteSparse. And LAV uses RFS, the sorting range is too large, so the computation results cannot be scattered on the chip, which also causes overhead. In addition, the sequential execution of dense and sparse parts in LAV reduces the SpMV efficiency.

Matrices with more balanced row length distributions have less zero-padding and ESB typically have higher performance, such as `audikw_1` and `marine1`. In-row performance is higher for irregular matrices such as `ins2` and `rajat30`. HALAV combines the



(a) Speedup comparison of the different methods over CSR.



(b) Speedup comparison of MKL vs SpMV_HaLAV using four clusters.

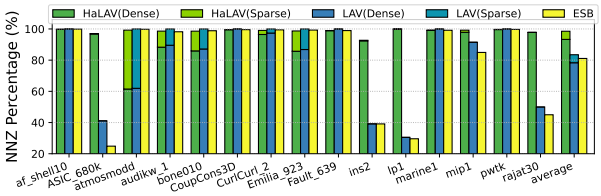
Figure 8: Performance comparison of SpMV_HaLAV with other SpMV methods.

features of both formats and provides better acceleration, especially `audikw_1`. For the matrices `atmosmodd` and `af_shell10`, HaLAV is not the fastest. This is because these two matrices have good locality, making HaLAV's optimization methods less effective. For matrices with dimensions less than 0.78M, such as `CoupCons3D`, `Fault_639`, `marine1`, `pwtk`, the corresponding vector x can be transferred directly to the GSM cache. Since there is no column blocking, their sparse part is only of type Low and contains fewer nonzeros. Therefore, the performance of HaLAV's heterogeneous parallelism is insignificantly improved compared to ESB. However, HaLAV has significant speedup for matrices with large dimensions that require column blocking, e.g., `audikw_1`, `Emilia_923`.

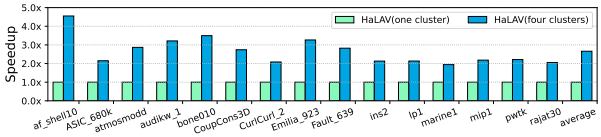
5.2.2 Performance of FT-M7032. We compare the performance of HaLAV using four DSP clusters of FT-M7032 with the highly optimized MKL on an Intel processor. HaLAV (four clusters) uses four MPI processes, with each MPI process running HaLAV on one DSP cluster and 4 CPU cores. And the MKL uses the E5-2692 processor's 12 threads for parallel execution. Figure 8(b) shows that HaLAV (four clusters) has a significant acceleration compared to MKL for many matrices, with an average speedup of 1.66×. Especially for matrices `audikw_1`, `bone010`, and `af_shell10`, the acceleration is more pronounced. The 16 CPU cores in the GP region are distributed across 8 NUMA nodes, and accessing data across NUMA nodes can cause significant performance degradation. CSR-O uses *numactl* to interleave memory allocation on all NUMA nodes to improve CSR performance and achieves an average speedup of 0.65× compared to the MKL.

Figure 9(b) tests the scalability of HaLAV. HaLAV has an average speedup of 2.66× when using four clusters compared to one cluster. Note that the speedup of `af_shell10` is greater than 4. This is because the dense part of each cluster has a different $Ratio_{z2nz}$. Therefore, each cluster can dynamically select the most appropriate HaLAV kernel, which effectively improves the scalability.

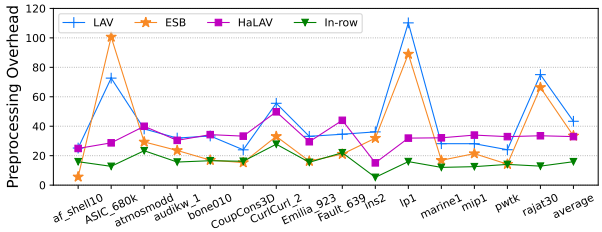
5.2.3 Evaluation of the parameter N_d. We use S_{d2c} to represent the speedup of the DSP side compared to the CPU side, which can be measured by using micro-benchmarks. If the task is divided between DSP and CPU according to $S_{d2c} : 1$, HaLAV has the highest parallel efficiency, with a time consumption proportional to $\frac{NNZ}{S_{d2c}+1}$. However, nonzeros are stored in column blocks, making it difficult to achieve optimal task allocation. We try to find the near-optimal division. If the DSP side calculates all the tasks, the time consumption is proportional to $\frac{NNZ}{S_{d2c}}$. If the task is divided between DSP and CPU according to $S_{d2c} - 1 : 1$, the CPU becomes the performance bottleneck. At this point, the parallel computation time is proportional to $\frac{NNZ}{S_{d2c}}$, which is equal to the computation time on the DSP side only. Therefore, the CPU computation task cannot exceed $\frac{NNZ}{S_{d2c}}$, otherwise, it will affect the parallel efficiency.



(a) The proportion of nonzeros in different formats.



(b) Scalability of HaLAV.



(c) Comparison of the preprocessing time.

Figure 9: Analysis of HaLAV.

Specifically, N_d can be calculated as:

$$\left(\mathrm{NNZ} - \sum_{i=1}^{N_\mathrm{d}} NNZ_{blocki}\right) < \frac{\mathrm{NNZ}}{S_{d2c}}, \tag{1}$$

thus,

$$\sum_{i=1}^{N_\mathrm{d}} NNZ_{blocki} > \mathrm{NNZ} * \frac{S_{d2c} - 1}{S_{d2c}}. \tag{2}$$

We can select a relatively small N_d that satisfies the Formula (2) as the number of column blocks of dense block in Section 4.2, where NNZ_{blocki} represents the nonzeros of the i-th column block. The S_{d2c} of HALAV in Figure 8(a) is the speedup of ESB compared to CSR. In practice, HALAV typically divides a matrix into only a few column blocks. A simple way to determine the optimal N_d is by trial and error.

5.2.4 Percentage of nonzeros. Figure 9(a) shows the percentage of nonzeros in different formats. In-row is not considered here because it has no zero padding. Overall, HALAV is the most effective in reducing zero padding, followed by LAV. For regular matrices, LAV has a slightly higher percentage of nonzeros, this is because LAV uses RFS. However, for irregular matrices such as `ASIC_680k`, `ins2`, `lp1`, `rajat30`, the advantage of HALAV in reducing zero padding is more significant. Figure 9(a) also breaks down the nonzeros' share of the dense and sparse parts in HALAV and LAV. It can be seen that for most matrices, the dense part of both formats contains at least 80% nonzeros. This is because the sparse part takes data from the DDR for each computation, and if there are too many elements it will affect the SpMV overall efficiency.

5.2.5 Preprocessing overhead. We use $\frac{Time_{pre}}{Time_{cpu}}$ to measure the preprocessing overhead. $Time_{pre}$ is the preprocessing time and $Time_{cpu}$ is the execution time of CSR. Figure 9(c) shows that LAV preprocessing is the most expensive, costing on average 43× CSR runtime. In-row has the lowest overhead of 16×. HALAV is also light, requiring only 32× runtime. We find that for matrices with large $Ratio_{z2nz}$, such as `ASIC_680k`, `lp1`, and `rajat30`, the preprocessing overhead of ESB and LAV is high. This is because ESB and the dense parts of LAV introduce more zero padding. HALAV maintains a low preprocessing overhead for all matrices. Note that format conversion can be done offline and the SpMV kernel is often called many times, so the one-time preprocessing overhead is usually negligible.

6 RELATED WORK

In this section, we briefly introduce some current optimization methods for multi-core DSPs. Gao et al. [5] developed an SpMV approach utilizing on-chip scratchpads for the TI Keystone II DSP, achieving a performance improvement of about 50%. Liu et al. [16] designed a two-level DMA double-buffer optimization method to achieve overlapping computation and data access, which effectively improves GEMM performance. Wang et al. [22] proposed a structured grid-based SpMV algorithm on M-DSP, achieving a bandwidth utilization of about 72%. Bi et al. [1, 2] developed efficient SpMV kernels for banded matrices and block sparse matrices on DSP by exploiting the characteristics of structural matrices. However, this method is not suitable for general SpMV operations because general sparse matrices don't have such structural properties.

Heterogeneous processors integrate two different types of cores. Using such architectures can improve SpMV performance, but research in this area is limited. Key challenges arise from the architectural differences between the core types. Li et al. [12] proposed a parallel algorithm called HASpMV for asymmetric multi-core processors (AMPs). By testing different core characteristics, HASpMV achieved good cache locality and load balancing across different core types. In this paper, we propose HALAV, which can efficiently improve the SpMV performance on heterogeneous multi-core DSPs by enhancing locality and vectorization.

7 CONCLUSION

In this paper, we present our experience in optimizing SpMV on heterogeneous multi-core DSPs, using the FT-M7032 platform as a case study. To enhance the efficiency of SpMV on heterogeneous architectures, we propose HALAV, a new SpMV method for heterogeneous architecture. HALAV partitions the input matrix into a dense and a sparse part. The dense part automatically selects vectorization-friendly storage formats and exploits data locality. The sparse part is processed using the standard CSR algorithm. We evaluate our approach on FT-M7032, and experimental results show that HALAV achieves average speedups of 2.09× over the competing baselines in the GP area. When utilizing all four DSP clusters, HALAV demonstrates superior performance compared to the Intel highly optimized MKL library, and the average speedup can be up to 1.66×. However, we also find that HALAV is not the fastest method for the `atmosmodd` and `af_shell10` matrices in Figure 8(a). In the future, we will analyze the characteristics of such matrices to further improve the SpMV performance of HALAV.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments and Weifeng Liu for his valuable suggestions on this article. This work was supported in part by the National Key R&D Program of China under grant agreement 2023YFB3001702, the National Science Foundation of China (NSFC) under grant agreements 61902411, 62032023, 12002382, 11275269, 42104078, U22B2005, and the Excellent Youth Foundation of Hunan Province under grant agreement 2021JJ10050. Deshun Bi and Shengguo Li contributed equally to this work and Dezun Dong is the corresponding author.

REFERENCES

[1] Deshun Bi, Shengguo Li, Yichen Zhang, Xiaojian Yang, and Dezun Dong. 2023. Efficiently Running SpMV on Multi-core DSPs for Banded Matrix. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 201–220.

[2] Deshun Bi, Xiaowen Tian, Shengguo Li, and Dezun Dong. 2023. Efficiently Running SpMV on Multi-Core DSPs for Block Sparse Matrix. In *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 1912–1919.

[3] T. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25.

[4] Yang Gao and Jason D Bakos. 2013. Sparse matrix-vector multiply on the texas instruments c6678 digital signal processor. In *2013 IEEE 24th ASAP*. IEEE, 168–174.

[5] Yang Gao, Fan Zhang, and Jason D. Bakos. 2014. Sparse matrix-vector multiply on the Keystone II Digital Signal Processor. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2014.7040985

[6] Francisco D Igual, Murtaza Ali, Arnon Friedmann, Eric Stotzer, Timothy Wentz, and Robert A van de Geijn. 2012. 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*. IEEE, 1–11.

[7] Eun-Jin Im and Katherine Yelick. 2000. Optimization of sparse matrix kernels for data mining. In *submitted to First SIAM Conf. on Data Mining*.

[8] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[9] David R Kincaid, Thomas C Oppe, and David M Young. 1989. *ITPACKV 2D user's guide*. Technical Report. Texas Univ., Austin, TX (USA). Center for Numerical Analysis.

[10] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM Journal on Scientific Computing* 36, 5 (2014), C401–C423.

[11] Chenyang Li, Tian Xia, Wenzhe Zhao, Nanning Zheng, and Pengju Ren. 2021. SpV8: Pursuing optimal vectorization and regular computation pattern in SpMV. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 661–666.

[12] Wenxuan Li, Helin Cheng, Zhengyang Lu, Yuechen Lu, and Weifeng Liu. 2023. HASpMV: Heterogeneity-Aware Sparse Matrix-Vector Multiplication on Modern Asymmetric Multicore Processors. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 209–220.

[13] Changxi Liu, Biwei Xie, Xin Liu, Wei Xue, Hailong Yang, and Xu Liu. 2018. Towards efficient SpMV on Sunway manycore architectures. In *Proceedings of the 2018 International Conference on Supercomputing*. 363–373.

[14] Sheng Liu, Yasong Cao, and Shuwei Sun. 2022. Mapping and Optimization Method of SpMV on Multi-DSP Accelerator. *Electronics* 11, 22 (2022), 3699.

[15] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 273–282.

[16] Zhong Liu and Xi Tian. 2018. Vectorization of matrix multiplication for multi-core vector processors. *Chin. J. Comput* 41, 10 (2018), 2251–2264.

[17] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*. Springer, 111–125.

[18] Yuyao Niu, Lu Zhengyang, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *35th IPDPS*. IEEE, 68–78.

[19] Abhiprayah Tiwari, Vivek Kumar, and Gaurav Mitra. 2018. *High performance and energy optimal parallel programming on CPU and DSP based MPSOC*. Ph. D. Dissertation. Ph. D. dissertation, IIIT-Delhi.

[20] James D Trotter, Sinan Ekmekçibaşı, Johannes Langguth, Tugba Torun, Emre Düzakın, Aleksandar Ilic, and Didem Unat. 2023. Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[21] Yaohua Wang, Chen Li, Chang Liu, Sheng Liu, Yuanwu Lei, Jian Zhang, Yang Zhang, and Yang Guo. 2021. Advancing DSP into HPC, AI, and beyond: challenges, mechanisms, and future directions. *CCF Transactions on High Performance Computing* 3 (2021), 114–125.

[22] Yang Wang, Jie Liu, Xiaoxiong Zhu, Qingyang Zhang, Shengguo Li, and Qinglin Wang. 2023. Improving Structured Grid-Based Sparse Matrix-Vector Multiplication and Gauss–Seidel Iteration on GPDSP. *Applied Sciences* 13, 15 (2023), 8952.

[23] Biwei Xie, Zhen Jia, and Yungang Bao. 2019. Benchmarking SpMV Methods on Many-Core Platforms. In *Benchmarking, Measuring, and Optimizing: First BenchCouncil International Symposium, Bench 2018, Seattle, WA, USA, December 10-13, 2018, Revised Selected Papers 1*. Springer, 233–247.

[24] Chao Yang, Shuming Chen, Jian Zhang, Zhao Lv, and Zhi Wang. 2019. A Novel DSP Architecture for Scientific Computing and Deep Learning. *IEEE Access* 7 (2019), 36413–36425. https://doi.org/10.1109/ACCESS.2019.2905302

[25] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2020. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[26] Serif Yesil, Azin Heidarshenas, Adam Morrison, and Josep Torrellas. 2023. WISE: Predicting the Performance of Sparse Matrix Vector Multiplication with Machine Learning. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 329–341.