An Integral-equation-oriented Vectorized SpMV Algorithm and its Application on CT Imaging Reconstruction

Weicai Ye, Chenghuan Huang, Jiasheng Huang, Jiajun Li, Yao Lu, Ying Jiang*
School of Computer Science and Engineering,
Guangdong Province Key Laboratory of Computational Science,
Sun Yat-sen University, Guangzhou 510275, China
cai_rcy@163.com, entityless@outlook.com, winrg315@126.com,
lijj87@mail2.sysu.edu.cn, {luyao23, jiangy32}@mail.sysu.edu.cn

***Abstract*—Sparse-matrix vector multiplication (SpMV) is a core routine in many applications. Its performance is limited by memory bandwidth, which is for matrix transport between processors and memory, and instruction latency in computations. Vectorized operations (SIMD) can dramatically improve the execution efficiency, but irregular matrices' sparsity pattern is not compatible with the style of SIMD execution. We present a new matrix format, Compressed Sparse Column Vector (CSCV), and a corresponding vectorized SpMV algorithm for matrices arising from integral equations. This SpMV algorithm can inherently suit wide SIMD instructions and reduce the memory bandwidth used. We implement this algorithm for Computed Tomography (CT) imaging reconstructions on both Intel and AMD x86 platforms and compare it with seven state-of-the-art SpMV implementations using different CT imaging matrices. Experimental results show that CSCV can achieve up to 96.9 GFLOP/s in single-precision tests, with speedup 3.70× to MKL and 3.48× to the second place implementation. Furthermore, the implementation of CSCV SpMV is performance portable, which excludes almost all SIMD assemble code and has promising performance with compiler-assisted vectorization. Code Availability: https://github.com/sysu-compsci/cscv**

***Index Terms*—parallel SpMV, integral-operator-oriented vectorization, CT imaging reconstruction**

I. Introduction

SpMV is a core routine utilized in many applications: scientific and engineering computing, economic modeling, graph analysis with social network data, etc. SpMV operations are often executed at high frequency with the same matrix and dominate the performance in these applications. Therefore, accelerating SpMV on various specific hardware is an important task, especially on a single compute node where the Central Processing Unit (CPU) cores share main memory to exchange data.

The SpMV used in Computed Tomography (CT) must be accelerated. CT is widely used to visualize three-dimensional images of human and animal body structures using a computer from a series of plane cross-sectional images made along an axis. CT is typically used in screening for specific disease conditions (i.e., lung cancer). A core procedure in CT is imaging reconstruction to solve an integral equation with projection data. Iterative CT imaging reconstruction methods (i.e., MBIR [10], [12]) are used for better image quality where SpMV is a key process. As the size of projection data increases, the reconstruction speed becomes a bottleneck of improving the image quality of CT.

There are two categories of methods to accelerate SpMV arising from integral equations, like in iterative CT imaging reconstruction. Methods in the first category utilize the specific matrices' sparsity pattern [6], [14], which gains promising speedup in a specified CT imaging reconstruction algorithm. Methods of the second category ignore the specific matrices' sparsity patterns, transform matrices into general storage formats, and optimize SpMV operations with commonly used matrix storage format, i.e., the compressed format Compress Sparse Row (CSR) [1], the Compress Sparse Column (CSC) [15], a block-wise format [4], and the hybrid one [3].

The most critical challenge of the existing methods for accelerating SpMV arising from integral equations is efficiently using wide SIMD instructions, extending from 512- to 2048-bit wide [11]. Because of irregular matrix sparsity patterns, there are requirements for extra memory loading and scattering operations. Furthermore, these requirements become much more intensive when matrix elements are in single-precision. However, processing data in single-precision is a daily used scenario in CT imaging reconstruction where the existing methods do not perform well.

We propose a new matrix format Compressed Sparse Column Vector (CSCV), and a corresponding vectorized SpMV algorithm. CSCV format takes advantage of geometric characteristics of line integral imaging operators. As a result, the CSCV SpMV algorithm can potentially accelerate SpMV in imaging models involving Radon transformation and attenuated X-ray transformation. This type of imaging model is used in CT, PET, SPECT imaging. Furthermore, the CSCV SpMV algorithm inherently not only accommodates wide SIMD instructions but also reduces the memory bandwidth used.

* Correspongding to: Ying Jiang (jiangy32@mail.sysu.edu.cn).

To present the ideas of CSCV, we choose CT imaging reconstruction with parallel beam geometry as an example, which is the most widely used and mature in various CT imaging geometries. We implement our algorithm on both Intel and AMD x86 platforms and compare it with 8 state-of-the-art SpMV implementations using different CT imaging matrices. Experimental results show that CSCV implementations can outperform other competitors and have speedups within $1.05 \sim 3.48$× to the closest one. Furthermore, the implementation of CSCV for SpMV is performance portable without using any processor-specific intrinsics.

The rest of this article is organized as below. We review related works in Section II and show details of challenges in vectorized CSC-based SpMV in Section III. Section IV is devoted to presenting ideas of the CSCV and its SpMV algorithm. Experiment results are presented in section V, and concluding remarks are made in Section VI.

II. Accelerating SpMV on a Single Computer

In this paper, SpMV is defined as follows. Given an $m \times n$ sparse matrix A, an n-element input vector x, and an m-element output vector y. SpMV can be denoted as $y = Ax$, where $y_i = \sum_{j \in \{j : A_{ij} \neq 0\}} A_{ij} \times x_j$. Different from dense matrix vector multiplication, SpMV only invokes the nonzero entries of the matrix A. Therefore, we need to utilize the extra data for indexing the nonzero entries or treat a given sparse matrix as a set of dense sub-matrices.

The performance of SpMV in a single compute node is limited by both memory bandwidth and instruction latency in computations. The memory bandwidth determines the shortest time for transmitting matrix data between processors and memory. When using the full compute node, the peak performance of SpMV is determined by the memory bandwidth. Instruction latency often comes from the float-point instructions and instructions to load and store vector elements. In addition, for SpMV, the latency also comes from additional instructions for indicating the nonzeros. When using a small number of CPU threads, the time consumed by SpMV is dominated by instruction latency [13], where the latency from instructions for indicating the nonzeros is a negligible part.

To reduce the consumption of memory bandwidth in SpMV, two categories of methods using different storage formats have been proposed. Methods of category one take advantage of the sparsity pattern of matrices. ELL in [2] is the one of this sort, designed for matrices arising from solving the elliptic Partial Differential Equation (PDE). Methods of category two ignore the sparsity pattern of matrices and use general storage formats to represent matrices and optimize the SpMV on given formats. Common storage formats of this category include CSR [1], CSC [15], CSR5 [9], SPC5 [3], LAV [16].

The methods in the second category can be divided into three types by matrix representation used. Methods of the first type use compressed formats to represent the matrix, such as CSR, CSC, and CSR5. Among them, CSR is the most basic one. CSR format representation of the matrix contains three arrays: row offset, column index, and value. CSC format is similar to that of CSR, except that CSC format is a column-major layout. The SpMV algorithm for CSC format is shown on Algorithm 1. The CSC and CSR formats have common shortcomings: vector elements associated with nonzero elements in rows/columns are not necessarily stored in adjacent cells in memory, and the memory access of vector elements is indirect. Therefore, SpMV in CSR and CSC formats are not suitable for vectorization. In order to adapt to vectorization, CSR5 proposes an efficient segment summation algorithm based on CSR format. Methods of the second type represent the matrix as a collection of dense sub-matrices, such as [17]. The dense matrix structure is suitable for vectorization operations and can significantly reduce the number of integers used for indication. However, useless zeros are filled into the matrix. As a result, additional memory bandwidth consumption reduces SpMV performance. Methods of the last type combine the advantages of the first two methods and represent the matrix as dense and sparse parts, such as SPC5 and LAV. The SPC5 format represents a matrix as a collection of dense sub-matrices with a fixed size but only stores the nonzero values in the sub-matrices. By using efficient hardware instructions to expand the compressed nonzero values into a complete matrix, the SpMV algorithm using the SPC5 format not only adapts to vectorization operations but also greatly reduces the consumption of memory bandwidth. LAV divides the matrix into a dense part and a sparse part in order to obtain better parallel scalability than CSR in power graph calculations.

There are three difficulties in designing efficient SpMV solutions:

- accommodating wide SIMD instructions like AVX-512,
- fast determination of parameters for block structure,
- optimizing SpMV code without hard coding assemble code to obtain the advantage of low-level SIMD instructions.

The solution to the first difficulty is that the memory access continuity of SpMV must be guaranteed. Otherwise, collecting the unaligned data into the SIMD register and distributing data to the correct target location takes longer than the calculation using the SIMD instruction. Using block-based structures is a way to ensure the continuity of memory access. The source of the second difficulty is that the size of the dense matrix blocks must be optimized one by one for different matrices, as shown in [7], [5]. In order to effectively use vectorized instructions, it is usually necessary to add machine-related SIMD instruction calls to the code or even use assembly code. Without solutions to the third difficulty, the result is that either the code can

Algorithm 1 SpMV for CSC format

```
1: for $i = 0$ to $n - 1$ do
2:     for $j = col\_ptr[i]$ to $col\_ptr[i+1] - 1$ do
3:         $y_{row\_ptr[j]} = x_i * val[j] + y_{row\_ptr[j]}$
4:     end for
5: end for
```

not run on other hardware platforms, or multiple copies of different machine-related code implementations would be prepared.

In order to overcome these difficulties, we propose a permutation algorithm based on integral operators. By rearranging matrix elements and vector elements, a matrix is represented as a collection of dense vectors stored in CSCV format. Therefore, the SpMV process using CSCV matrix format is suitable for wide SIMD operations, and assembly SIMD instructions used in SpMV can be automatically generated by compilers. In addition, the parameter selection in the matrix replacement process does not need to be carried out on a case-by-case basis.

III. Challenges in Vectorized CSC-based SpMV for Integral Equations

We denote SpMV implementations using the CSR format matrices as CSR-based SpMV. CSC-based SpMV and CSCV-based SpMV are denoted in a similar way.

In the case of solving integral equations like CT image reconstruction, CSC-based SpMV has a wider application range than that of CSR-based SpMV. CSC-based SpMV is not only suitable for the traditional ART type algorithms, but also for the Iterative Coordinate Descent (ICD) algorithms (refer to [6] and [14]). CSR-based SpMV does well in ART-type algorithms but is inefficient in ICD algorithms.

In [6] and [14], the authors proposed methods that speed up the specified CT imaging reconstruction algorithms by utilizing the sparsity pattern of the matrix. [14] proposed a block-wise local reordering method to improve locality of reference of vector y in CT imaging reconstruction with parallel beam geometry. [6] presented a method to accelerate SpMV in CT imaging reconstruction with cone-beam geometry. Both methods have moderate vectorization efficiency and were not fully vectorized.

Unlike CSR-based SpMV, CSC-based SpMV implementations require additional reordering of vector y. The reordering hinders the effective vectorization of CSC-based SpMV. For example, Algorithm 2 is a vectorized CSC-based SpMV algorithm, usually introduced by compiler automatic SIMD code generation. In Algorithm 2, additional instructions are needed for vector permutation: gathering and packing elements of vector y (in line 5) and then scattering them back (in line 7). These additional instructions for permutation take much time, even more than that of the SIMD computation step (in line 6).

Algorithm 2 A vectorized CSC-based SpMV

```
Input: a CSC matrix $A$, vector $x$ and $y$, $S_{\text{VVec}}$
Output: vector $y$
1: for each column $i$ in $A$ do
2:     Load $x_i$
3:     for each $S_{\text{VVec}}$ long segment $j$ in column $A_{*,i}$ do
4:         Load $A_{*,i}[j]$ and row index set $r(A_{*,i}[j])$
5:         Gather all elements in $y$ with $id \in r(A_{*,i}[j])$ into a $S_{\text{VVec}}$ long vector $Y_{tmp}$
6:         $Y_{tmp} = x_i * A_{*,i}[j] + Y_{tmp}$
7:         Scatter $Y_{tmp}$ to $y$
8:     end for
9: end for
```

To obtain an efficient vectorized CSC-based SpMV algorithm, we need to balance the consistency of permutation instructions and zero element access rate by finding an optimal global permutation strategy. To reduce the cost induced by vector permutation, we hope that the same set of permutation instructions can be used for as many columns as possible (this is called permutation instruction consistency), and the zero elements in matrix A are accessed in the calculation as few as possible (this is called zero element access rate).

An algorithm with higher permutation instruction consistency means that it is more efficient to gather and scatter data, and an algorithm with a lower zero access rate means that it is more efficient to perform multiplication. However, for the SpMV problem, the high consistency of permutation instructions and low zero element access rate are contradictory.

Finding out an optimal global permutation strategy can be modeled as solving a graph or hypergraph problem. Each node in the graph/hypergraph represents a row or column of the matrix. Each weighted edge in the graph represents a nonzero matrix element with different schemes, while each weighted hyperedge in the hypergraph is a column id set or row id set. In [11], the permutation problem was modeled as a Traveling Salesman Problem (TSP), finding the shortest but most effective route for a person given a list of specific destinations. In [6] and [15], the permutation strategy problem was modeled as a hypergraph edge-based partition problem that divides a set of nodes into several parts by minimizing the cut edge cost. Both TSP and hypergraph partition problems are NP-Complete.

It is very costly to find the optimal global permutation strategy. A feasible solution is to use block-wise local reordering with extra ad hoc permutation. This is the idea we vectorize SpMV for CSC style formats.

IV. Basic CSCV Design

This section will introduce the CSCV matrix format and the idea of its SpMV algorithm. The key to developing the CSCV algorithm is to design local permutation strategies oriented by integral equations. For these purposes, we first demonstrate the concept of CSCV and review the geometric properties of the integral operator in CT imaging reconstruction. Secondly, we construct a local unified reordering strategy for the vector y in each matrix block based on the geometric properties of the integral operator and propose a fully vectorized SpMV algorithm with column-major style. Finally, we reorder the column segments of the matrix in CSCV format and pack them into Vectorized eXecution Groups (*VxGs* for short) to shield memory access latency.

A. CSCV for SpMV

CSCV concept has three components: a matrix storage format (called CSCV format), SpMV algorithms using matrices in CSCV storage format and local permutation strategies. We will discuss these three components briefly as follows.

CSCV format is a vectorized CSC style matrix storage format. In the CSCV format, the nonzeros on the same column are stored in multiple *CSCV*Elements (short for CSCVE). CSCVE is a vector whose length is fixed at S_{VVec}. Matrix elements stored in the same CSCVE have identical column and continuous row subscripts. Therefore, every CSCVE is suitable for SIMD operations. In addition, the data used by CSCVE as row indices will be greatly reduced. When the row subscripts of matrix elements are not the same, only an ad hoc permutation is required.

Algorithm 3 is a fully vectorized CSCV-based SpMV. For different matrix blocks, different permutation strategies are ad hoc applied to vector y. When permutation is completed, there is no gather and scatter operations in loops of SpMV.

There are two local permutation strategies used in CSCV algorithms, which are applied to matrices and vectors, respectively. The design of optimal permutation strategies is based on the geometric properties of the integral operator in CT imaging reconstruction.

B. Geometric Properties of Integral Operators

In this sub-section, we review geometric properties of integral operators in CT imaging reconstruction, which are sources of finding optimal local permutation. These properties are independent of discretization methods for solving integral equations. Therefore, the matrices produced by this type of integral operators preserve similar features for reordering.

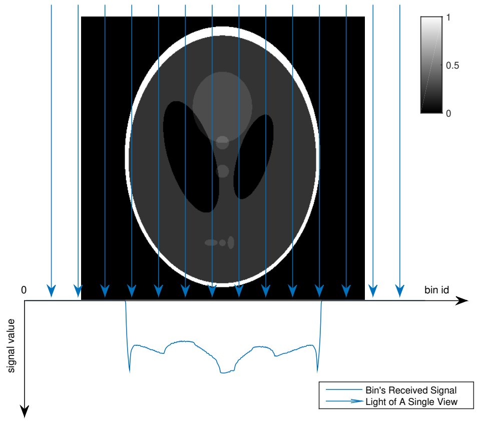
In this paper, we study the SpMV for solving the following integral equation:

$$\int_{-\infty}^{\infty} L\left(o,q\right) u\left(o+tq\right) dt = f\left(o,q\right), \tag{1}$$

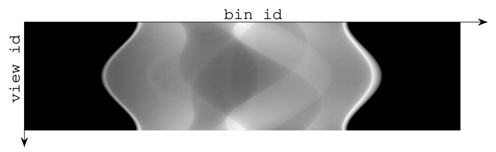
where $u : R^m \mapsto R, f, L : R^{2m} \mapsto R, o \in R^m, t \in R, q \in S^{m-1}$, and S^{m-1} is the unit sphere in R^m. When $L(o,q) = 1$,

Algorithm 3 A fully vectorized CSCV-based SpMV with local ad hoc vector permutation

```
Input: a CSCV matrix $A$, vector $x$ and $y$, $S_{\text{VVec}}$, mapping set $\{\iota_k\}$
Output: vector $y$
 1: for each matrix block $A^k$ in $A$ do
 2:     Reorder $y$ with mapping $\iota_k$ to vector $\tilde{y}$
 3:     for each column $i$ in $A^k$ do
 4:         Load value $x_i$ of vector $x$
 5:         for each $S_{\text{VVec}}$ long segment $j$ in column $A^k_{*,i}$ do
 6:             load vector $V_j$ and the first row index id $q$ related to $V_j$
 7:             load vector $\tilde{y}_q$ from $\tilde{y}$ started with $q$
 8:             $\tilde{y}_q = x_i * V_j + \tilde{y}_q$
 9:         end for
10:     end for
11:     Reorder $\tilde{y}$ with inverse mapping of $\iota_k$ to $y$
12: end for
```



(a) Forward projection of an image in a single view.



(b) Sinogram of image in Figure 1a.

Fig. 1: Forward projection and sinogram in CT imaging.

the above integral equation depicts the processing of the CT imaging. The function u models the unknown attenuation and is reconstructed by available sinogram/projection data f. After discretization, Eq 1 becomes a linear system $Ax = y$, where x is for function u and y is for sinogram f, respectively.

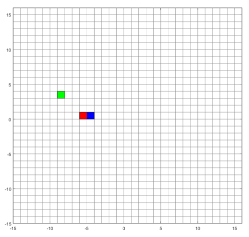
For simplicity, we discuss the case when m is 2, corresponding to the two-dimensional (2-D in brief) CT imaging reconstruction problem with parallel beam geometry, and the discretization function space consist of piecewise constant functions on a square. In parallel beam geometry, q is for different sample views. During the imaging process, X-rays pass through the object and are detected by the bins from the other side of the light source. This process is shown in Figure 1a. A sinogram consists of bins' received signal view by view, as Figure 1b shows. Due to $q \in S$, for different o, the nonzero data set of $f(o, .)$ $R(f, o, *)$ is quite different, shown in Figure 2b. That is the reason why so difficult to find an optimal reordering scheme for vector y satisfying Algorithm 3.

An entry of matrix A describes the contribution of attenuation value of a grid element in the image domain to a bin's signal in a single view. Some separated pixel blocks in the region of the image domain are shown in Figure 2a. Figure 2b shows their traces in the projection domain, marked with color, which is not similar.

The following properties are from Eq 1 in [8].

P1: For any given view, contiguous pixels in a 2-D image are mapped to either contiguous or identified bins by the integral operator. Nevertheless, pixels in a 2-D image are mapped to contiguous or identified bins maybe not be contiguous.

P2: For any given view, the integral operator maps a pixel to a close interval on the bin line.


(a) Pixels in image domain.

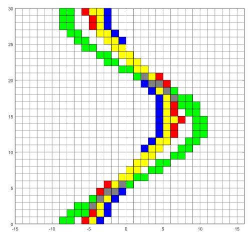

(b) Traces of pixels in projection domain.

Fig. 2: Trajectories of different pixels in the projection domain. In Fig. (b), blue, green, and red trajectories correspond to the unique projection points of different color pixels in Fig. (a), respectively. The yellow trajectory represents the projection points shared by red and blue pixels, and the gray trajectory represents the projection points shared by all three pixels. The trajectory of adjacent red and blue pixels share many similar traces, while non-adjacent pixels also share some identical traces in limited views.

For adjacent pixels, the projection trajectories are either the same or adjacent. As a result, elements in vector x corresponding to adjacent pixels have many identical or adjacent elements of y. Figure 2 shows this result, like the red and blue pixels. The green pixel is not contiguous with the blue one, shown in Figure 2a; however, there exists a view interval in Figure 2b, the trajectory of green and blue joint together.

In [14], the author constructs a Block Transpose Buffer (BTB in brief) to take advantage of the first part of P1 and P2 to improve the data locality of SpMV. It transposes the subset of vector y from bin-major ordering to view-major ordering. However, using BTB can not eliminate the permutation operations of vector y in Algorithm 2. As a result, the SpMV in [14] can not be fully vectorized.

P3: For any column of the matrix arising from an integral operator, the nnz is similar.

P3 is the base for balancing workload for thread-level parallelism. This property is wildly used in both accelerating methods on CPUs, GPUs, and Clusters.

C. A Local Permutation Algorithm for CSCV

We propose an Integral Operator Based Local Reordering (short for IOBLR) as a local permutation strategy. In CT reconstruction, the integral operator projects a pixel in the reconstructed image into a bunch of curves, and the curves corresponding to adjacent pixels are piecewise parallel. The core idea of the IOBLR algorithm is to treat all projection data trajectories of a group of pixels as a bunch of piecewise parallel curves and then transform the spatial coordinates of projection data into the offsets of parallel curves and the coordinates on curves. Then, the corresponding matrix's nonzeros on a parallel segment form a CSCVE.

IOBLR rearranges the projection data according to the projection trajectories, which is different from the matrix-independent coordinate axis rotation local arrangement algorithm used by BTB. Therefore, IOBLR can achieve fully vectorized SpMV calculation, while BTB only improves the locality of memory access.

TABLE I: Information of the sample matrix block

Full image size	25 * 25
Number of Bins	38
Delta Angle	4°
Image Block Range	Row: [5, 9], Col: [5, 9]
Block Start Angle	32°
S_{VVec}	8
S_{VxG}	2

IOBLR in the CSCV algorithm theoretically supports different CT imaging geometries because IOBLR is based on the geometry properties P1, P2, and P3 of line integral imaging operators. These properties ensure the trajectories of pixel projection are still piecewise parallel, and the local reordering strategy IOBLR is effective, even for different CT geometries or other medical imaging methods (such as SPECT and PET). Without loss of generality, we choose parallel beam geometric CT imaging as an example to show our design.

All following examples in this section will use a tiny sub-matrix block described in Table I. In current CSCV implementations, the number of views in the matrix block equals S_{VVec}.

Figure 3 shows the memory layouts of three columns in the example matrix block before and after conversion to CSCV format. Before converting to CSCV format, the nonzeros of each column corresponds to the points on the projection trajectory of one pixel, which are represented in blue lattices. The nonzeros in a column are stored in a bin-major layout. After conversion, nonzeros recorded in CSCV format are represented as grids on the red parallel polylines. After the conversion, the newly added padding zeros are represented by yellow lattices. Each CSCVE corresponds to an arrow polyline segment. The nonzero values in a CSCVE are stored in an IOBLR-major layout where the elements along the direction of the arrow are contiguous in memory. The shapes of parallel polylines are determined by the curve of the minimum bin number of the reference pixel, and the reference pixel is determined by the center point of the pixel block.

Figure 4 shows examples of zero padding required for SIMD operations under different vector y layouts. The bin-major layout is typical in CT imaging reconstruction while the view-major layout is used in BTB. The IOBLR is used for CSCV. Let us take $S_{\text{VVec}} = 8$ as an example. Obviously, the IOBLR is the best.

The efficiency of an IOBLR algorithm is measured by the zero-padding rate of a matrix. The zero-padding rate of a given matrix is defined as $R_{\text{nnzE}} := nnz\left(\tilde{A}\right)/nnz\left(A\right) - 1$, where $nnz\left(\tilde{A}\right)$ is the number of nonzeros of matrix A in CSCV format, and $nnz\left(A\right)$ is the original one. The higher the zero-padding rate, the lower the efficiency of an IOBLR algorithm. The statistics of all pixels in the example block are shown in Figure 5.

There are two factors that affect efficiency: the size of image block (short for S_{ImgB}) and the length of CSCVE (S_{VVec}). A larger S_{ImgB} always results in a larger zero-padding rate.

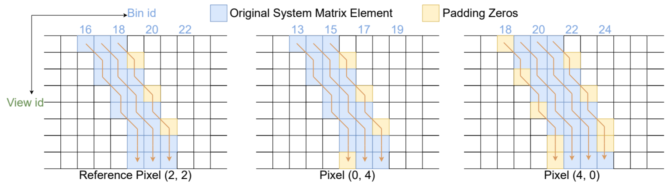


Fig. 3: The memory layout of CSCVEs of different pixels with an IOBLR, shown by projection trajectories. A blue and a yellow lattice indicate a nonzero and a zero in the system matrix, respectively; colored lattices in an arrow polyline form a CSCVE. The polyline shows the trajectory of the reference pixel in IOBLR.

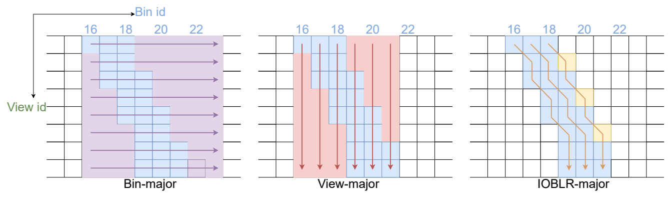


Fig. 4: SIMD-efficiency comparison under the different layout of vector y. There are three layouts: bin-major, view-major, and IOBLR-major, shown by different color arrow lines. SIMD-efficiency is defined by the area of the intersection of arrow lines and the blue grids, which represent nonzeros in the matrix. When S_{VVec} is 8, the range of area of bin-major, view-major and IOBLR-major layouts are 3, $2 \sim 6$ and $7 \sim 8$, respectively.

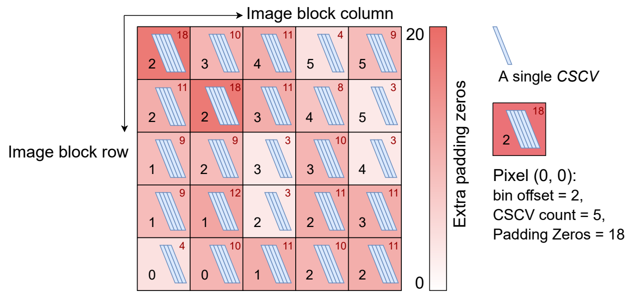
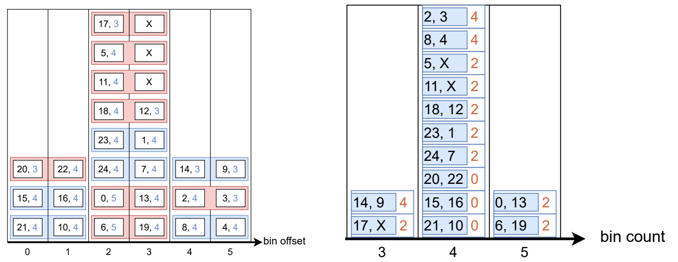


Fig. 5: Distribution of zero-padding, *CSCVE* numbers and offsets of parallel curves (short for bin offset) brought by different reference pixels. The darker color, the larger number of padding zeros.

The same is true for S_{VVec}. However, a larger S_{VVec} would reduce the amount of index data. The experimental part will introduce the joint effect of S_{ImgB} and S_{VVec} on the zero-padding rate. The zero-padding rate is mostly about 25%–45% in our experiments.

D. Design of VxG

We propose VxG to compress the size of index data further and reduce the latency of memory access. A VxG concatenates CSCVEs sharing the same projection data and ensures that the subscripts of the vector y accessed in different CSCVEs are continuous. The length of a VxG is fixed. After using



(a) Order pixels by bin Offset, and construct *VxG* (*VxGs* are marked red if extra padding zeros are introduced in this step)

(b) Order *VxGs* by bin count

Fig. 6: Constructing and Ordering *VxG*

VxGs in CSCV format, the index data volume of a matrix is 0.25 times that before use and 0.03 times that of CSC format. Moreover, using multiple VxGs improves the temporal locality of accessing memory and increases the number of instructions in the innermost loop of SpMV, which helps compilers to rearrange the instruction pipeline and shield memory access latency.

Figure 6 shows an construction process of VxGs which are obtained by sorting CSCVEs twice, one is through the offsets of parallel curves and the other is through the counter. The left sub-figure in Figure 6 describes that CSCVEs are sorted by bin offsets of parallel curves, and VxGs are constructed. Each box represents a CSCVE. The number in the box means "(offset, count)". If additional padding zeros are introduced during this process, the VxG is marked red. The right figure shows the procedure of sorting VxGs by bin count and building the final result of VxGs.

E. Other Implementation Details

Padding-zeros removing. The implementation of CSCV has two branches: CSCV-Z without removing padding zeros and CSCV-M with removing padding zeros. A mask-based technique and a vector expansion technique are used to remove padding zeros in CSCV-M, whose concept is the same as that of SPC5 [3].

On Intel platforms, CSCV-M uses the hardware *vexpand* instructions in AVX-512 for vector expansion; on other platforms, vector expansion is implemented by software code denoted as *soft-vexpand* in CSCV-M. After removing padding zeros, approximately 30% of the memory bandwidth is saved. However, vector expansion operations introduce a large amount of additional instruction latency.

Multi-threaded implementation In a compute node, there is only one CSCV process. In order to overcome the performance degradation of false sharing among CPU cores, each thread has its own local copy of vector y. After SpMV calculation of each thread is completed, CSCV process sums up vector y copies globally with multi-threads. In addition, we use block partitioning for vector x and row partitioning for the matrix, which can reduce the size of each copy of vector y that each thread needs to save. The entire matrix will create

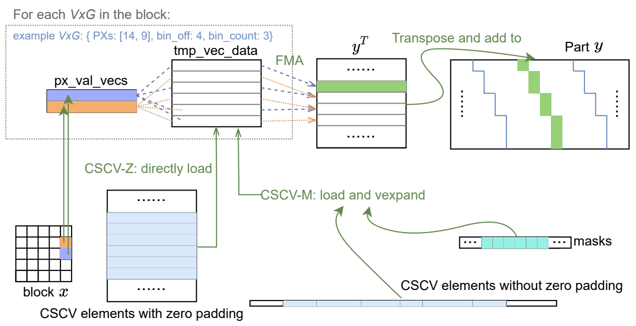

For each *VxG* in the block:
example *VxG*: { PXs: [14, 9], bin_off: 4, bin_count: 3}
tmp_vec_data
px_val_vecs
FMA
y^T
Transpose and add to
Part y
CSCV-Z: directly load
CSCV-M: load and vexpand
masks
block x
CSCV elements without zero padding
CSCV elements with zero padding

Fig. 7: The whole process of CSCV-based SpMV

fixed-size blocks according to the granular configuration, and it is guaranteed that each thread will get at least one block.

Vectorization. In order to improve the portability of the source code, implementations of the CSCV SpMV algorithm use the compiler-assistant automatic vectorization to enable SIMD floating-point computation except for the *vexpand* instruction instead of intrinsic function calls for using SIMD Instructions. Therefore, only S_{VVec} and S_{VxG} required by CSCV need to be set, and then high-efficiency vectorization floating-point computation can be realized through compilation. The specific results will be shown in the chapter on performance evaluation.

Process. The whole process of SpMV using CSCV format is divided into matrix format conversion before calculation, local ad hoc reordering, and fully vectorized SpMV calculation in the calculation process, as shown in Figure 7.

V. Evaluation

A. General Setup

We choose a dual-socket Intel Xeon Gold 6130 (short for SKL) system and a dual-socket AMD EPYC 7452 (short for Zen2) system for the performance evaluation. Both SKL and Zen2 CPUs are classic cache-based x86 multicore processors, supporting 512-bit wide (AVX-512) and 256-bit wide (AVX-256) SIMD instructions at most, respectively. The peak read-only memory bandwidths (M_{PBw} for short) of SKL and Zen2 systems are 202.8GB/s and 236.43GB/s, respectively. The value of bandwidth was obtained by the Intel MLC benchmark suite. Both SKL and Zen2 platforms run CentOS Linux Enterprise Server 7.6. On the SKL platform, the hyper-threading is turned on.

We compare CSCV implementations, CSCV-Z and CSCV-M, against other SpMV implementations, including SPC5, CVR, Merge, CSR5, VHCC, ESB, and Intel MKL. To compile SpMV implementations, we use Intel compiler 19.1.3.304 and GCC compiler 10.2 on SKL and Zen2 systems, respectively. Intel MKL provides SpMV implementations for CSR and CSC formats, denoted as MKL-CSR and MKL-CSC, respectively. On both SKL and Zen2 platforms, Intel MKL runtime is provided by Intel compiler 19.1.3.304. All SpMV implementations are compiled with *"-xHost"* and *"-march=native"* flags

TABLE II: Information of the matrix datasets

reconstructed img size	num bin	num view	delta angle	nnz	x size	y size
512×512	730	240	0.75°	166148730	262144	175200
768×768	1096	480	0.375°	747032208	589824	526080
1024×1024	1460	480	0.375°	1328114108	1048576	700800
2048×2048	2920	160	0.1875°	1750179564	4194304	467200

on SKL platform and Zen2 platform, respectively. On all platform, implementations are compiled with the flag *"-O3"*.

All tests on the SKL platform are run under the environment variable *"KMP_AFFINITY=scatter,granularity=fine"* to configure the thread affinity of OpenMP. On Zen2 platform, *"KMP_AFFINITY=scatter,granularity=tile"* is used. Only one process with multi-threads is used in tests. All implementations are tuned up to the optimal, including using the best kernel of SPC5, the best scheduling of ESB, etc. Some implementations can not complete all tests. ESB, VHCC, CSR5, and CVR only provide double-precision SpMV kernels and can not finish single-precision tests. SPC5 and VHCC only work on Intel platforms due to using platform-related instructions. VHCC does not support running in one and two threads. We use VHCC code in the CVR public repository because the *tzcnti* instruction in the original VHCC code is only available on Intel Knights Corner platform.

B. Datasets

To evaluate the SpMV performance of $y = Ax$, we chose matrices from integral equations in CT imaging reconstruction whose geometric setting is parallel beam. The three small matrices are used for clinical CT, and the latter two matrices are used for micro CT, with 1024×1024 used for both clinical and micro cases. To fit in the memory size of the computing node, a matrix that uses limited angles to reconstruct a 2048×2048 image was selected for tests. The information of used matrices, including CT geometric setting parameters, is shown in Table II. The memory size of these matrices is apparently larger than the size of L3 caches of each CPU.

The matrices used for tests are in both single and double precision. SpMV with single-precision matrices is a daily used routine in clinical medicine images. Furthermore, it is more challenging to optimize single-precision SpMV because it is more difficult to saturate the memory bandwidth in single precision, and double-precision SpMV can easily achieve higher memory bandwidth usage than that of single-precision SpMV. However, not all SpMV implementations support single-precision SpMV.

C. Test methods

Performance is measured by the minimum SpMV execution time recorded with at least 100 SpMV iterations under the best specific configuration. The minimum execution time is advantageous over the average and the median in avoiding random time overhead like threads' fork-join and memory allocation operations.

The performance data reported include the execution time, the performance of floating-point units in GigaFLOP per Second (GFLOP/s in brief), and the effective memory bandwidth usage ratio in SpMV. The performance of floating-point units of a system is defined as:

$$F(A, p) = 2 \times nnz(A)/T(p),$$

where $nnz(A)$ is the number of nonzeros of matrix A and $T(p)$ is the execution time of SpMV with p threads. The effective memory bandwidth usage ratio is defined as:

$$R_{\text{EM}} = (M(A) + M(x) + M(y)) \,/\, (T(p) \times M_{\text{PBw}}) \,,$$

where M_{PBw} is the read only memory bandwidth. $M(A)$, $M(x)$ and $M(y)$ are the total memory requirements of matrix A, vectors x and y, respectively. $M(A)$ can be various in different implementations. The sum of $M(A)$, $M(x)$ and $M(y)$ is denoted as M_{Rit}, indicating the minimum memory need to be read per iteration of $y = Ax$, generally called as memory requirements in the following subsections.

D. Selecting Parameters for CSCV

In order to select the optimal combinations of S_{VxG}, S_{VVec} and S_{ImgB} for CSCV implementations, we analyze the relationship between performance and parameters. Here, we evaluate the memory requirements and performance capabilities of CSCV-Z and CSCV-M. R_{nnzE} is used to characterize both additional memory requirements and useless floating-point operations introduced by a given combination of parameters. The value in GFLOP/s measures floating-point operation capabilities. The larger value in GFLOP/s, the better performance.

We choose the single-precision matrix to reconstruct images of the size 1024×1024 pixels for fast performance trend analysis and parameter selection. Due to the geometrical nature of the line integral operator, the parameter selection required for CSCV implementations does not need to be done on a case-by-case basis. The single-precision matrix is used because its performance is more sensitive to the choice of parameter combinations, especially when using multiple threads.

The distribution of R_{nnzE} and memory requirements of both CSCV-Z and CSCV-M are shown in Figure 8. As shown in this figure, R_{nnzE} increases as S_{VxG}, S_{VVec} and S_{ImgB} increase. When S_{ImgB} is large, the correlation between R_{nnzE} and S_{VxG} becomes weak, indicating that it is easier to form *VxG* without introducing extra nonzeros. The larger the S_{VVec}, the more obvious rising trend of R_{nnzE} is usually. As both S_{VVec} and S_{ImgB} increase simultaneously, the memory requirement increases significantly because the trajectory similarity between different pixels decreases as the average distance from the reference pixel increases. Therefore, the number of padding zeros introduced with IOBLR will increase.

The memory requirements of CSCV-Z and CSCV-M are quite different. Under the same parameters, the memory requirement of CSCV-M is significantly lower than that of CSCV-Z. When removing padding-zeros, the memory requirement of CSCV-Z is proportional to R_{nnzE}, while the memory requirement of CSCV-M has little to do with S_{VxG} and S_{ImgB}. In addition, when S_{VVec} changes from 4 to 8, the memory required by CSCV-M is reduced because the effective number of bits per mask byte doubles.

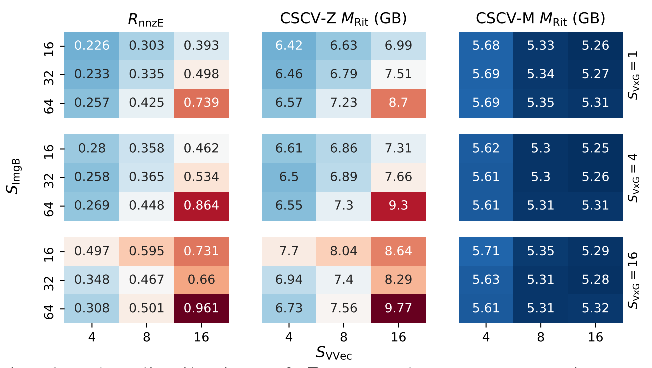


Fig. 8: The distribution of R_{nnzE} and memory requirements of CSCV-Z and CSCV-M about combinations of S_{VVec}, S_{ImgB} and S_{VxG}.

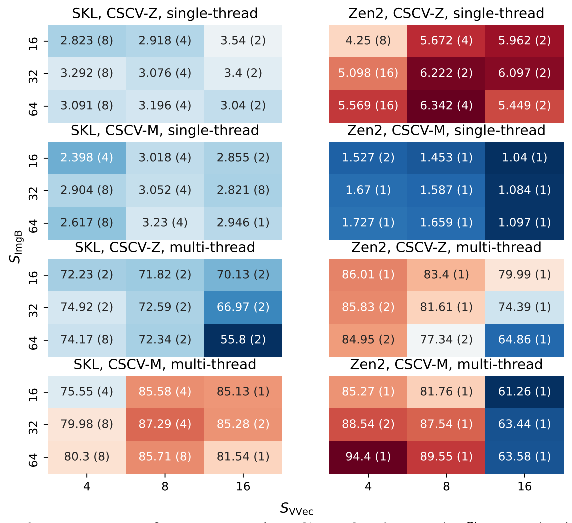


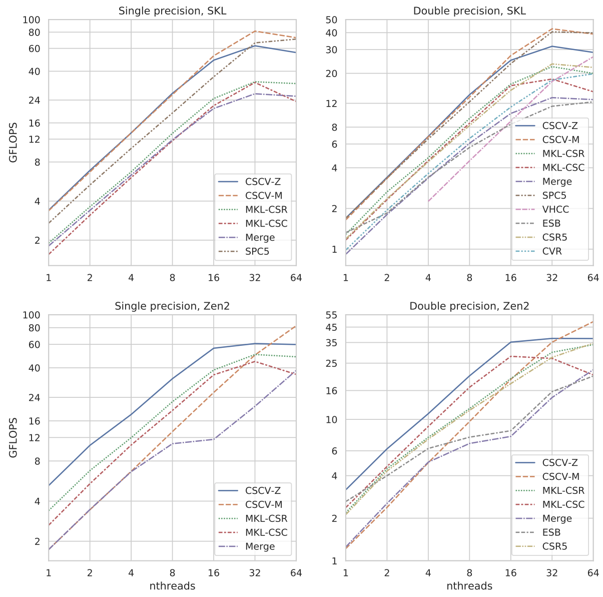
Fig. 9: Best performance in GFLOP/s and S_{VxG} choices of CSCV implementations for different S_{VVec} and S_{ImgB}

TABLE III: The parameter combination used in parallel tests for CSCV implementations and the R_{nnzE} in storage formats

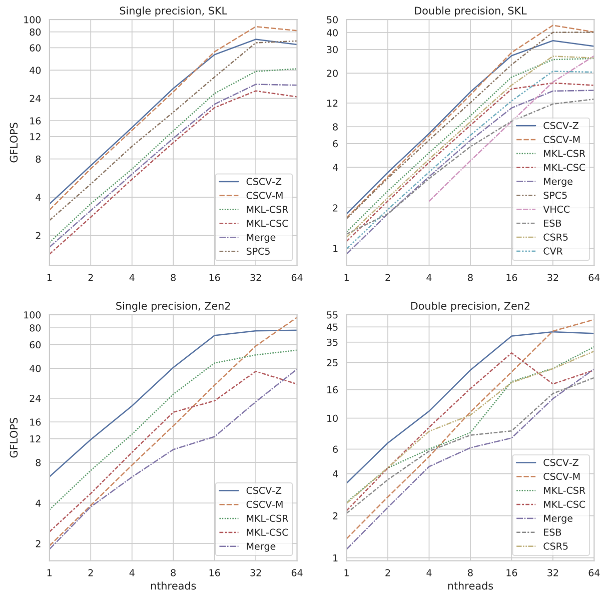
platform	implementation	precision	S_{ImgB}	S_{VVec}	S_{VxG}	R_{nnzE}
SKL	CSCV-Z	single	16	16	2	0.417
		double	16	16	2	0.417
	CSCV-M	single	32	8	4	0.365
		double	16	16	2	0.417
Zen2	CSCV-Z	single	64	8	4	0.448
		double	32	8	2	0.345
	CSCV-M	single	64	4	1	0.257
		double	16	8	1	0.303

Figure 9 demonstrates the performance distribution in GFLOP/s with different S_{ImgB} and S_{VVec} combinations. The best performance combination is selected from S_{VxG} in the range of 1 to 16. The selected S_{VxG} value is shown in the parentheses and to the right of the GFLOP/s value.

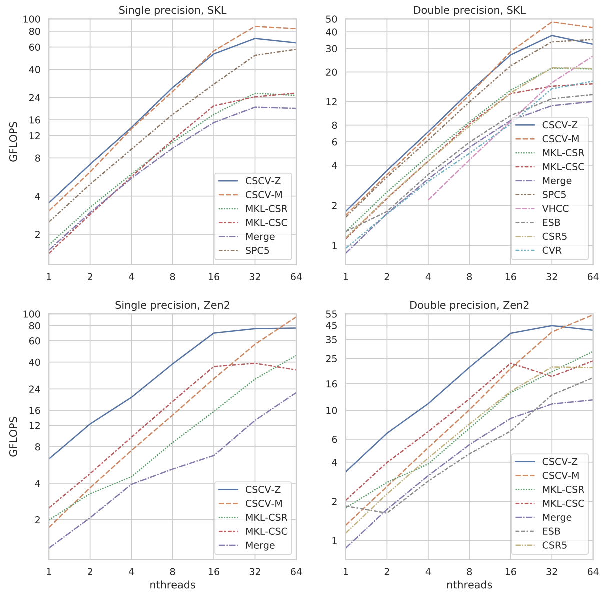
Performance measured in GFLOP/s varies across platforms. The single-threaded performance of CSCV-Z on the Zen2 platform is almost twice that of the SKL platform, indicating the single-core performance of the Zen2 platform is stronger than that of the SKL platform. However, in the single-threaded cases of CSCV-M, the performance of the Zen2 platform is about half of that of the SKL platform, revealing the high instruction overhead of *soft-vexpand*. Thanks to the reduction,



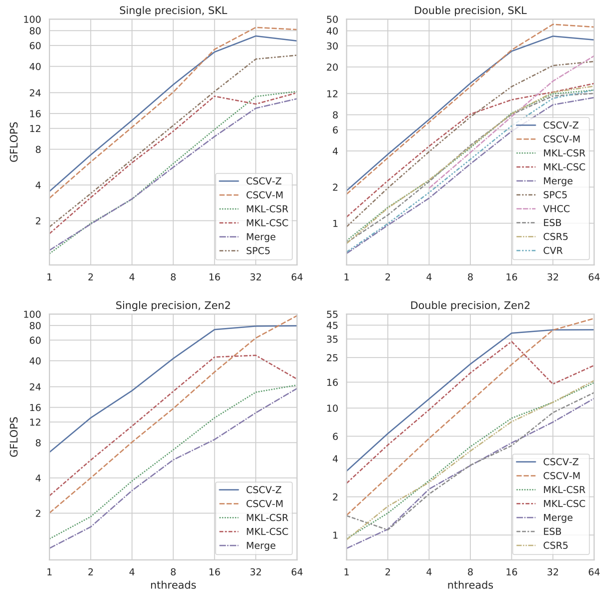
(a) Reconstrued image size = 512×512



(b) Reconstrued image size = 768×768



(c) Reconstrued image size = 1024×1024



(d) Reconstrued image size = 2048×2048

Fig. 10: Scalability of different SpMV implementations in GFLOP/s

resulting in less expense of both loading x and reducing y^T in blocks. For the multi-threaded performance of CSCV-Z on Zen2, the smallest S_{ImgB} and S_{VVec} obtain the best for its low instruction count and latency as well as aligned memory access, CSCV-Z provided the best performance in single-threaded cases. Nevertheless, on both SKL and Zen2 platforms, CSCV-M obtains the best multi-threaded performance, indicating that the bottleneck has shifted from the computing power of cores to memory bandwidth.

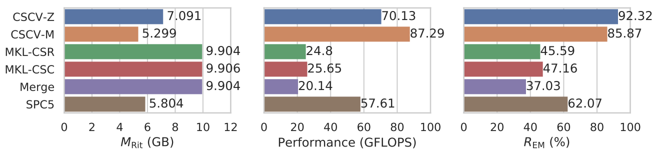
When S_{ImgB} is big, the number of blocks will be reduced, resulting in less expense of both loading x and reducing y^T in blocks. For multi-threaded performance of CSCV-Z on Zen2, the smallest S_{ImgB} and S_{VVec} obtain the best performance, which shows that the performance of CSCV-Z is directed related to R_{nnzE} or M_{Rit} and is bounded by memory bandwidth since the computing power excesses. For CSCV-M on Zen2, the bottleneck is still on the computing power; thus the best performance is obtained when S_{ImgB} is 64; with S_{VVec} as 4, 8, and 16, value of performance are 94.4, 89.55 and 63.58 GFLOP/s, with R_{nnzE} as 0.257, 0.425 and 0.739, revealing the correlation of R_{nnzE} and the instruction expense of CSCV-M.

Choices of S_{VxG} depend on the platform used. Since using *VxG* has a positive impact on instruction pipelining, the best S_{VxG} choices are usually greater than 1 in tests on SKL platform and the tests of single-threaded CSCV-Z on Zen2 platform. The best S_{VxG} in all tests of CSCV-M on Zen2 platform is 1; although the bottleneck CSCV-M on Zen2 is instruction latency, *soft-vexpand* has already filled up the inner loop of SpMV with enough instructions.

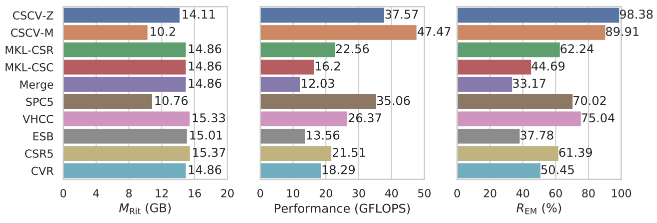
As a summary, Table III lists the parameter combinations used in parallel performance tests on different platforms with other implementations. The principle of our choice of combination is to get the best single-threaded performance for CSCV-Z and the best multi-threaded performance for CSCV-M.

E. Parallel Performance

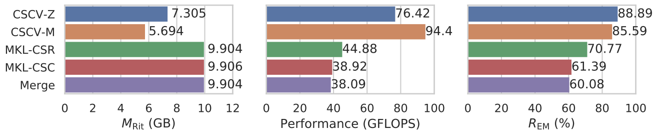
Figure 10 shows distribution functions of computing performance with respect to the number of threads and reflects the scalability of different implementations. On the SKL platform, almost all implementations are linearly scalable in the range of $1 \sim 16$ threads. Only VHCC could visibly improve performance when using 64 threads. Therefore, VHCC would be instruction latency bounded and take advantage of hyperthreading. When the number of threads is in the range of $1 \sim 8$, CSCV-Z is the best. In other cases, CSCV-M is the most prominent. On the Zen2 platform, CSCV-Z achieves the best performance when the number of threads does not exceed 32; when 64 threads are used, the performance of CSCV-M is better than that of CSCV-Z. From single thread to 64 threads on the Zen2 platform, CSCV-M is nearly linearly scalable,



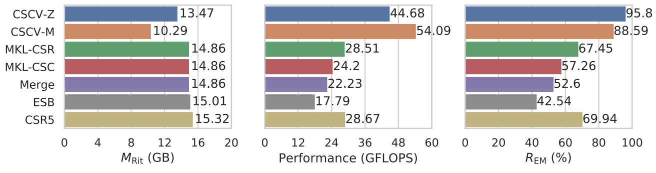
(a) Single precision, SKL



(b) Double precision, SKL



(c) Single precision, Zen2



(d) Double precision, Zen2

Fig. 11: Memory requirements, best performance in GFLOP/s and memory bandwidth usage ratio of different SpMV implementations when reconstrued image size = 1024×1024

indicating that the instruction overhead of *soft-vexpand* can be tackled with enough computing power.

CSCV-M, CSCV-Z, and SPC5 are superior to other implementations on the SKL platform, while CSCV-M, CSCV-Z, and MKL-CSR are among the top 3 with the highest performance on the Zen2 platform. In single-threaded tests, the performance of CSCV-Z can reach 6.63 GFLOP/s on the Zen2 platform, which is 2.35 times faster than the second place. Table IV lists the best computing performance of each implementation on the same platform in multi-threaded tests. The best performance of CSCV-M for processing double-precision data on SKL and Zen2 platforms is 45.3 and 53.8 GFLOP/s, respectively. The best speedups in these tests on SKL and Zen2 platforms over MKL-CSR are 3.00 and 2.34, respectively. In processing single-precision data tests, both platforms' best performance doubles, at most reaching 96.9 GFLOP/s. Those would be very helpful for the daily use of CT reconstruction. Furthermore, compared to the best performance of the second place, CSCV-M is about $1.05 \sim 3.48$ (resp. $1.06 \sim 2.30$) times faster in single-precision (resp. double-precision) procedure. And compared to MKL-CSR, CSCV-M is about $1.89 \sim 3.70$ (resp. $2.22 \sim 3.61$) times faster in single-precision (resp. double-precision) procedure.

As the size of the matrix increases, the performance advantage of CSCV implementations is further magnified. For example, the performance of CSCV-M in single-precision tests has risen from 82.8 GFLOP/s for small matrix tests to 96.9 GFLOP/s for large matrices on the Zen2 platform. This is the credit of IOBLR and zero padding removal technology. In contrast, the performance of most other comparison frameworks has slightly degraded.

To profile the multi-threaded performance of CSCV, we use the tests in reconstructing images of size of 1024×1024 to analyze the relationship among memory requirements, the best performance, and the corresponding effectiveness on memory bandwidth usage of different implementations. All data are shown in Figure 11. Compared to CSCV-M and SPC5, other frameworks have significantly higher memory requirements, and their performance is bounded by the memory bandwidth.

We have summarized two reasons for the difference in performance from data in Figure 11. Reason 1: When memory requirements are similar, effective bandwidth usage ratios determine performance. Therefore, the performance of CSCV-M on the SKL platform is better than that of SPC5. Reason 2: When the effective memory bandwidth usage ratios are similar, memory requirements are the bottleneck. This is why the performance of CSCV-M was always better than CSCV-Z even though CSCV-Z achieved 98.4% of M_{PBw}.

F. Summary of Experiments

In summary, CSCV implementations have an outstanding performance in all tests and show significant advantages in breaking the performance bounds of SpMV: CSCV-M and CSCV-Z would be suitable for cases of memory-bound and latency-bound, respectively.

TABLE IV: The best performance (in GFLOP/s) of each implementation with all 4 matrices' tests on a platform. In each sub-column, the best value is marked in bond style, and the second one is in italic style.

platform	precision	implementations	avg. perf.	max. perf.
SKL	single	CSCV-Z	*68.74*	*72.1*
		CSCV-M	**85.48**	**87.98**
		MKL-CSR	31.16	40.99
		MKL-CSC	27.55	32.75
		Merge	24.81	30.93
		SPC5	61.46	70.71
	double	CSCV-Z	*35.05*	37.57
		CSCV-M	**45.19**	**47.47**
		MKL-CSR	20.59	25.72
		MKL-CSC	16.48	18.15
		Merge	12.82	14.89
		SPC5	34.52	*40.54*
		VHCC	26.13	26.88
		ESB	12.68	13.56
		CSR5	21.39	26.72
		CVR	17.62	20.66
Zen2	single	CSCV-Z	*73.36*	*79.47*
		CSCV-M	**92.44**	**96.93**
		MKL-CSR	43.75	54.57
		MKL-CSC	41.56	44.63
		Merge	30.84	39.49
	double	CSCV-Z	*41.25*	*44.68*
		CSCV-M	**51.24**	**54.09**
		MKL-CSR	27.62	33.79
		MKL-CSC	28.66	33.45
		Merge	17.23	22.49
		ESB	17.7	20.27
		CSR5	25.69	34.63

The GFLOP/s performance of CSCV-based SpMV is portable. Although vectorization of CSCV is enabled by compiler-assisted, the performance of CSCV-M on the SKL platform is significantly better than that of the manually tuned SPC5 assembly code kernels, which is particularly prominent in single-precision tests. Moreover, the parameter selection of CSCV does not need to be carried out on a case-by-case basis.

VI. Conclusions and Future Works

In this paper, we proposed an integral-equation-oriented vectorized CSCV-based SpMV algorithm and implemented it on CT imaging reconstruction. The performance of CSCV implementations was at most 96.9 GFLOP/s in tests and portable on different x86 platforms with advantage on other SpMV implementations.

We will implement CSCV on $x = A^T y$ in CT backward projection. Besides implementing CSCV for matrices from CT imaging reconstruction with different geometries and other applications like SPECT and PET, we will also migrate CSCV onto other vectorization-enabled hardware platforms, like Sunway Taihulight, GPUs, and ARM CPUs.

VII. ACKNOWLEDGEMENTS

This work is partially supported by by Key-Area Research and Development Program of Guangdong Province (No.2021B0101190003), by National Science and Technology Major Project of the Ministry of Science and Technology of China Grant No.2016YFB0200602, by the Science and Technology Program of Guangzhou, China (No.201804020053), by NSFC under Grant 12126610 and Grant 81971691, by Guangdong Province Key Laboratory of Computational Science at the Sun Yat-sen University (2020B1212060032).

References

[1] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and Van der Vorst, H. *Templates for the solution of linear systems: building blocks for iterative methods.* SIAM, 1994.

[2] Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on cuda. Tech. rep., Citeseer, 2008.

[3] Bramas, B., and Kus, P. Computing the sparse matrix vector product using block-based kernels without zero padding on processors with AVX-512 instructions. *CoRR abs/1801.01134* (2018).

[4] Buatois, L., Caumon, G., and Lévy, B. Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems 24*, 3 (2009), 205–223.

[5] Buluç, A., Fineman, J. T., Frigo, M., Gilbert, J. R., and Leiserson, C. E. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures* (2009), pp. 233–244.

[6] Chen, P., Wahib, M., Wang, X., Takizawa, S., Hirofuchi, T., Ogawa, H., and Matsuoka, S. Performance portable back-projection algorithms on cpus: Agnostic data locality and vectorization optimizations. *CoRR abs/2104.13248* (2021).

[7] Choi, J. W., Singh, A., and Vuduc, R. W. Model-driven autotuning of sparse matrix-vector multiply on gpus. *ACM sigplan notices 45*, 5 (2010), 115–126.

[8] Helgason, S. *Integral Geometry and Radon Transforms.* Integral Geometry and Radon Transforms, 2011.

[9] Liu, W., and Vinter, B. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (New York, NY, USA, 2015), ICS '15, Association for Computing Machinery, p. 339–350.

[10] Sauer, K., and Bouman, C. A local update strategy for iterative reconstruction from projections. *IEEE Transactions on Signal Processing 41*, 2 (1993), 534–548.

[11] Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., Horsnell, M., Magklis, G., Martinez, A., Premillieu, N., et al. The arm scalable vector extension. *IEEE micro 37*, 2 (2017), 26–39.

[12] Thibault, J.-B., Sauer, K. D., Bouman, C. A., and Hsieh, J. A three-dimensional statistical approach to improved image quality for multislice helical ct. *Medical physics 34*, 11 (2007), 4526–4544.

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

[14] Wang, X., Sabne, A., Sakdhnagool, P., Kisner, S. J., Bouman, C. A., and Midkiff, S. P. Massively parallel 3d image reconstruction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2017), pp. 1–12.

[15] White, J. B., and Sadayappan, P. On improving the performance of sparse matrix-vector multiplication. In *Proceedings Fourth International Conference on High-Performance Computing* (1997), IEEE, pp. 66–71.

[16] Yesil, S., Heidarshenas, A., Morrison, A., and Torrellas, J. Speeding up spmv for power-law graph analytics by enhancing locality & vectorization. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2020), IEEE Computer Society, pp. 1219–1233.

[17] Yzelman, A. Generalised vectorisation for sparse matrix: vector multiplication. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms* (2015), pp. 1–8.