Information Sciences 523 (2020) 279–295

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

tpSpMV: A two-phase large-scale sparse matrix-vector multiplication kernel for manycore architectures

Yuedan Chen[a,b], Guoqing Xiao[a,b], Fan Wu[a,b], Zhuo Tang[a,b], Keqin Li[a,b,c,*]

[a] College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, Hunan, China
[b] National Supercomputing Center in Changsha, Changsha 410082, Hunan, China
[c] Department of Computer Science, State University of New York, New Paltz, New York 12561, USA

ARTICLE INFO

Article history:
Received 23 November 2019
Revised 7 March 2020
Accepted 11 March 2020
Available online 14 March 2020

Keywords:
CSR
Manycore
Parallelization
Sparse matrix-vector multiplication (SpMV)
SW26010

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is one of the important subroutines in numerical linear algebras widely used in lots of large-scale applications. Accelerating SpMV on multicore and manycore architectures based on Compressed Sparse Row (CSR) format via row-wise parallelization is one of the most popular directions. However, there are three main challenges in optimizing parallel CSR-based SpMV: (a) limited local memory of each computing unit can be overwhelmed by assignments to long rows of large-scale sparse matrices; (b) irregular accesses to the input vector result in expensive memory access latency; (c) sparse data structure leads to low bandwidth usage. This paper proposes a two-phase large-scale SpMV, called `tpSpMV`, based on the memory structure and computing architecture of multicore and manycore architectures to alleviate the three main difficulties. First, we propose the two-phase parallel execution technique for `tpSpMV` that performs parallel CSR-based SpMV into two separate phases to overcome the computational scale limitation. Second, we respectively propose the adaptive partitioning methods and parallelization designs using the local memory caching technique for the two phases to exploit the architectural advantages of the high-performance computing platforms and alleviate the problem of high memory access latency. Third, we design several optimizations, such as data reduction, aligned memory accessing, and pipeline technique, to improve bandwidth usage and optimize `tpSpMV`'s performance. Experimental results on SW26010 CPUs of the Sunway TaihuLight supercomputer prove that `tpSpMV` achieves up to 28.61 speedups and yields the performance improvement of 13.16% over the state-of-the-art work on average.

1. Introduction

In multicore and manycore era, many accelerators, such as Field Programmable Gate Array (FPGA) [1,2], x86 CPU [3], Intel Xeon Phi [4], SW26010 CPU [5,6], and General Purpose Graphics Processing Unit (GPGPU) [7–10], have been used widely in various fields for its characteristics of high-performance computational capacity. However, the large number of computing units posts a tricky challenge in meeting memory bandwidth requirements, especially for the memory-bound kernels such as sparse matrix-vector multiplication (SpMV).

* Corresponding author at: College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, Hunan, China.
E-mail addresses: chenyuedan@hnu.edu.cn (Y. Chen), xiaoguoqing@hnu.edu.cn (G. Xiao), ztang@hnu.edu.cn (Z. Tang), lik@newpaltz.edu (K. Li).

https://doi.org/10.1016/j.ins.2020.03.020
0020-0255/

a	b		c	d	
e		f	g	h	i
j			k	l	m
			n	o	
p		q	r		

Fig. 1. A sparse matrix *A*.

SpMV is an elementary and indispensable operation in many large-scale applications, such as solving sparse linear systems [11–13], electronic structure computations [14,15], graph computations [16–18], etc. It dominates the performance of involved applications. SpMV's expression is

$$y = A \times x, \tag{1}$$

where A is an input sparse matrix containing m rows, n columns, and nnz nonzeros, x is an input dense vector with n elements, and y is a result vector with m elements. The sparsity of A gives rise to irregular data access patterns and difficulty of exploiting data locality of SpMV.

CSR (compressed sparse row) [19] is one of the most popular sparse matrix storage formats that compresses the storage space. It uses three arrays to compress the sparse matrix A in Eq. (1): an integer array "$\boldsymbol{Pr}[m+1]$" recording pointers to the start and end positions of each row, an integer array "$\boldsymbol{Col}[nnz]$" recording column indices of nonzeros, and a floating-point array "$\boldsymbol{Val}[nnz]$" recording numerical values of nonzeros. We take a sparse matrix A, as shown in Fig. 1, as an example, where $m = 6$, $n = 6$, and $nnz = 18$. The three CSR arrays of A are presented as follows:

- $\boldsymbol{Pr}[7] = \{0, 4, 4, 9, 13, 15, 18\}$;
- $\boldsymbol{Col}[18] = \{0, 1, 3, 4, 0, 2, 3, 4, 5, 0, 3, 4, 5, 3, 4, 0, 2, 3\}$;
- $\boldsymbol{Val}[18] = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r\}$.

Additionally, the implementations and performance of SpMV vary according to the input sparse matrix's formats. The algorithm of CSR-based SpMV is shown in Algorithm 1.

Algorithm 1 The CSR-based SpMV.

Require: Three CSR arrays of A: $\boldsymbol{Pr}[m+1]$, $\boldsymbol{Col}[nnz]$, and $\boldsymbol{Val}[nnz]$;
The array of x: $\boldsymbol{x}[n]$;
The parameters of SpMV: m, n, and nnz.
Ensure: The array of y: $\boldsymbol{y}[m]$.
1: **for** each row r_i of A **do**
2: **for** each nonzero a of the row r_i **do**
3: $\boldsymbol{y}[i] = \boldsymbol{y}[i] + \boldsymbol{Val}[a] \times \boldsymbol{x}[\boldsymbol{Col}[a]]$;
4: **end for**
5: **end for**
6: **return** $\boldsymbol{y}[m]$.

Although CSR-based SpMV exposes straightforward row-wise parallelization, the performance of parallel CSR-based SpMV is mainly subject to data-dependent performance degradation caused by ***(a) the irregular row length of the input matrix A, (b) irregular data access patterns of SpMV***, and ***(c) sparse data structure***. First, as for large-scale input sparse matrices, there may be rows that are too long for the local memory of each computing core, resulting in the computational scale of parallel CSR-based SpMV is limited by the long rows and limited local memory. Second, the irregular data access patterns of SpMV result in inefficient memory accesses and large memory access latency on manycore architectures. Third, redundant data swapping and uncoalesced data accesses due to the sparse data structure lead to low bandwidth usage of manycore processors.

As noted, this paper designs `tpSpMV` to alleviate the difficulties of optimizing parallel CSR-based SpMV on manycore architectures. Our contributions mainly include:

- We propose a two-phase parallel execution technique that separates the parallel CSR-based SpMV into two phases: the partial CSR-based SpMV phase and accumulation phase, to alleviate the limitation of computing scale.

- We design the adaptive partitioning strategy and parallelization scheme for the two phases of tpSpMV by using local memory caching technique to leverage hardware advantages and reduce memory access latency, respectively.
- We further design several optimization techniques, i.e., data reduction, aligned memory accessing, and pipeline processing, to improve bandwidth utilization and optimize communication of tpSpMV.
- We evaluate tpSpMV on SW26010 CPU of the Sunway TaihuLight supercomputer [20]. The evaluation results prove that tpSpMV achieves up to 28.61 speedups and yields the performance improvement of 13.16% over the state-of-the-art work on average.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 details the features of SW26010 CPU. Section 4 presents parallelization design of tpSpMV. Section 5 demonstrates the communication optimization for tpSpMV. Section 6 evaluates performance of tpSpMV. Section 7 finally concludes this paper and gives our future work.

2. Related work

SpMV's performance in large-scale applications is limited arising from irregular row length and overloading datasets. Therefore, some research works are explored to alleviate this constrain for SpMV acceleration. Sadi et al. [21] present an algorithm co-optimized custom shared memory hardware accelerator for SpMV to overcome the problem of datasets exceeding the on-chip fast storage. Xiao et al. [22] design an auto-tuning four-way sparse matrix partitioning method based on a statistical model of matrix structure description for SpMV to fit in the on-chip storage. Merrill and Garland [23] propose an equitable multi-partitioning for sparse matrices to ensure that each thread can handle its assignment. Greathouse and Daga [24] design combination CSR-Adaptive SpMV that dynamically determines whether to execute a set of rows or with the traditional CSR-based SpMV based on the row length of the input matrix. ahSpMV [25] chooses a proper threshold for the hybrid (HYB) storage format based SpMV parallelization on heterogeneous manycore architectures to overcome the problem of irregular row length of sparse matrices and achieve better SpMV performance.

In addition, it has been widely observed that the irregularity of SpMV is a well-known challenge that limits SpMVâs parallelism. For this reason, a lot of works have been done to improve data locality and bandwidth utilization. Xie et al. [3] aim at both vectorization efficiency and locality and low preprocessing overhead by presenting the compressed vectorization-oriented sparse row (CVR) format for SpMV. Zhang et al. [26] devise Blocked Compressed Common Coordinate (BCCOO) format that reduces the memory footprint of SpMV to solve the bandwidth challenge, and use vertical partitioning strategy to achieve better data locality. Liu and Vinter [27] propose CSR5 (Compressed Sparse Row 5) that is insensitive to the sparsity structure of the input matrix on various parallel computing platforms. Ashari et al. [28] present an adaptive SpMV algorithm that combines rows into groups and adjusts the requested resources based on the number of nonzeros in rows to improve coalescing. BASMAT [29] optimizes SpMV by predicting the major performance bottleneck (bandwidth bound, memory latency bound, or thread imbalance) of an input matrix according to its sparsity features. Karsavuran et al. [30] identify five quality criteria that refer to the trade-off between the reuse of input matrices and parallel write. Elafrou et al. [31] propose an SpMV optimizer that applies suitable optimizations to tackle the performance bottlenecks of the input matrix. Yang et al. [32] partition sparse matrices using a probability mass function to obtain better data locality.

3. Features of the SW26010 manycore architecture

Fig. 2 shows the computing architecture and memory structure of SW26010 CPU.

The computing architecture of SW26010 provides multi-level parallelism. Each SW26010 chip contains four core groups (CGs), which provides the first level of parallelism. Moreover, one CG contains a management processing element (MPE) core and 64 computing processing element (CPE) cores. The MPE performs not only computations but also pre-processing, task assignment, etc. The 64 CPEs, arranged in 8 rows and 8 columns, perform parallel computing kernels, which provide the second level of parallelism. In addition, each CPE has a 256-bit vector unit, which provides another level of parallelism.

Programmers can develop the first level of parallelism among CGs by utilizing Message Passing Interface (MPI). As for developing the second level of parallelism among CPEs within a CG, the Sunway system provides a customized light-weight library, named *Athread*. Therefore, a CG is an MPI process and a CPE is a thread. In addition, the vector unit of each CPE can be utilized by vectorization, where the vectorization size is 4.

As for the cache-less memory hierarchy of SW26010, the MPE and 64 CPEs of each CG can access an 8GB DDR3 memory. An MPE has a 32KB L1 instruction cache and a 256KB L2 data cache. A CPE has a 64KB scratchpad memory, called local data memory (LDM), rather than data caches. The difference between the scratchpad memory and data cache is that the scratchpad memory is software-controlled, while the cache is hardware-controlled.

There are two approaches for data swapping between the memory and LDM: one is efficiently performed by via Direct Memory Access (DMA) and the other one is performed via Gload/Gstore with high latency. DMA prefers to transmit large chunks of data, while Gload/Gstore is suitable to discretely access small data.

Each CG has the peak performance of 765 GFlops and the maximum theoretical bandwidth of 34 GB/s. There are two key points to fully leverage the computing resources of SW26010 CPUs:

- **CG and CPE**: the multiple-level parallelism, i.e., the parallelism among CGs, the parallelism among CPEs within each CG should be carefully developed for computing kernels.

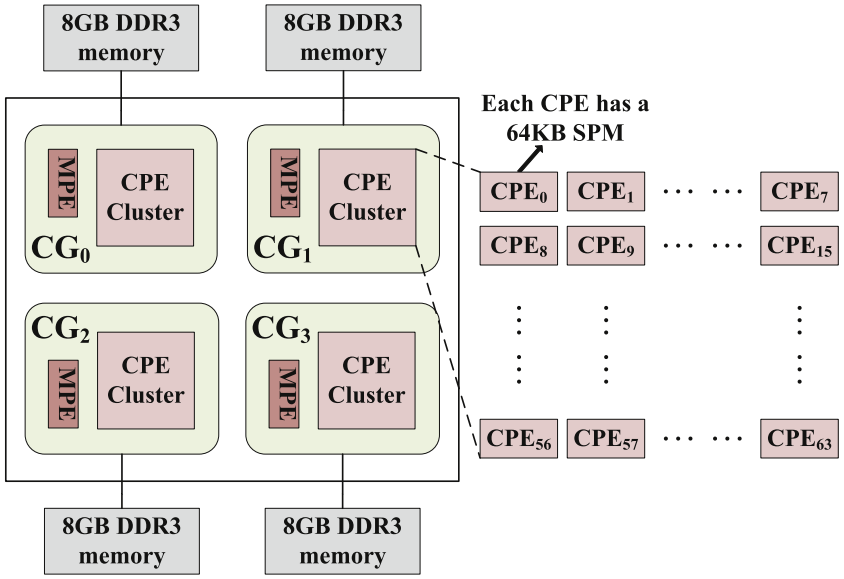

Fig. 2. The architecture of the SW26010 CPU.

- **LDM**: the limited memory of LDM demands a delicate design to make full use of the advantages.

4. Parallelization design of `tpSpMV`

Define the input sparse matrix A of `tpSpMV` to be a CSR-stored sparse matrix containing m rows, n columns, and nnz nonzeros. Define θ to be the number of CGs, and δ to be the number of CPEs of each CG. This paper proposes the parallelization design for `tpSpMV` on SW26010 CPUs that using the following techniques to optimize parallel CSR-based SpMV.

Local memory caching. To support efficient memory access pattern, the input vector data of x is distributed and cached in the local data memory, i.e., LDM on each CPE, so that each CPE performs calculations on all the cached data of x and corresponding distributed data of the input matrix A, which improves the data locality and data transmission performance between main memory and LDM.

Two-phase parallel execution. To support `tpSpMV` for large-scale data, the sparse matrix A is blocked into fine-grained submatrices, each of them has a suitable size for LDM. Therefore, based on the local memory caching of x, the results obtained from CPEs may be not the final result y. This paper proposes two-phase parallel execution to execute parallel CSR-based SpMV into two phases: (1) the parallel partial CSR-based SpMV phase; (2) the parallel accumulation phase. The first phase executes the parallel CSR-based SpMV based on the fine-grained data partitioning, and the second phase executes further accumulation to generate the final result.

4.1. Parallel partial CSR-based SpMV phase

4.1.1. The adaptive partitioning

We design an adaptive partitioning strategy for the partial SpMV phase on the SW26010 architecture to fully leverage the multiple-level parallelism and fully exploit the limited storage of LDM.

There are three layers of the adaptive partitioning for the partial CSR-based SpMV phase:

- ***Layer 1:***
  To leverage the first level parallelism among θ CGs, A is partitioned by rows into θ *blockA*s based on the number of rows and CGs. Each *blockA* contains m/θ rows and n columns of A.
  Each CG is assigned a *blockA* and the input vector x. The ith CG reads the $\lceil \frac{i\times m}{\theta} \rceil$th row through the $\lceil \frac{(i+1)\times m}{\theta} \rceil$th row of A to load the *blockA*, where $i \in \{0, 1, 2, \ldots, \theta - 1\}$. The results on each CG are θ segments, denoted as y'_{seg}, and each y'_{seg} contains m/θ elements.
- ***Layer 2:***
  To leverage the second level parallelism among δ CPEs within each CG, the *blockA* on each CG is partitioned by columns into δ *tileA*s based on the number of nonzeros and CPEs. Each *tileA* contains m/θ rows and n' columns. In addition, each *tileA* on the CG has roughly equal number of nonzeros.
  x on each CG is correspondingly partitioned into δ segments, denoted as x_{seg}. Each x_{seg} contains n' elements.
  As shown in Fig. 3, there is an array $\boldsymbol{Pc}[\delta + 1]$ on each CG that stores the positions of each *tileA* in *blockA*. Each CPE of a CG is assigned a *tileA* and the corresponding x_{seg} according to array $\boldsymbol{Pc}$. The jth CPE of each CG reads the

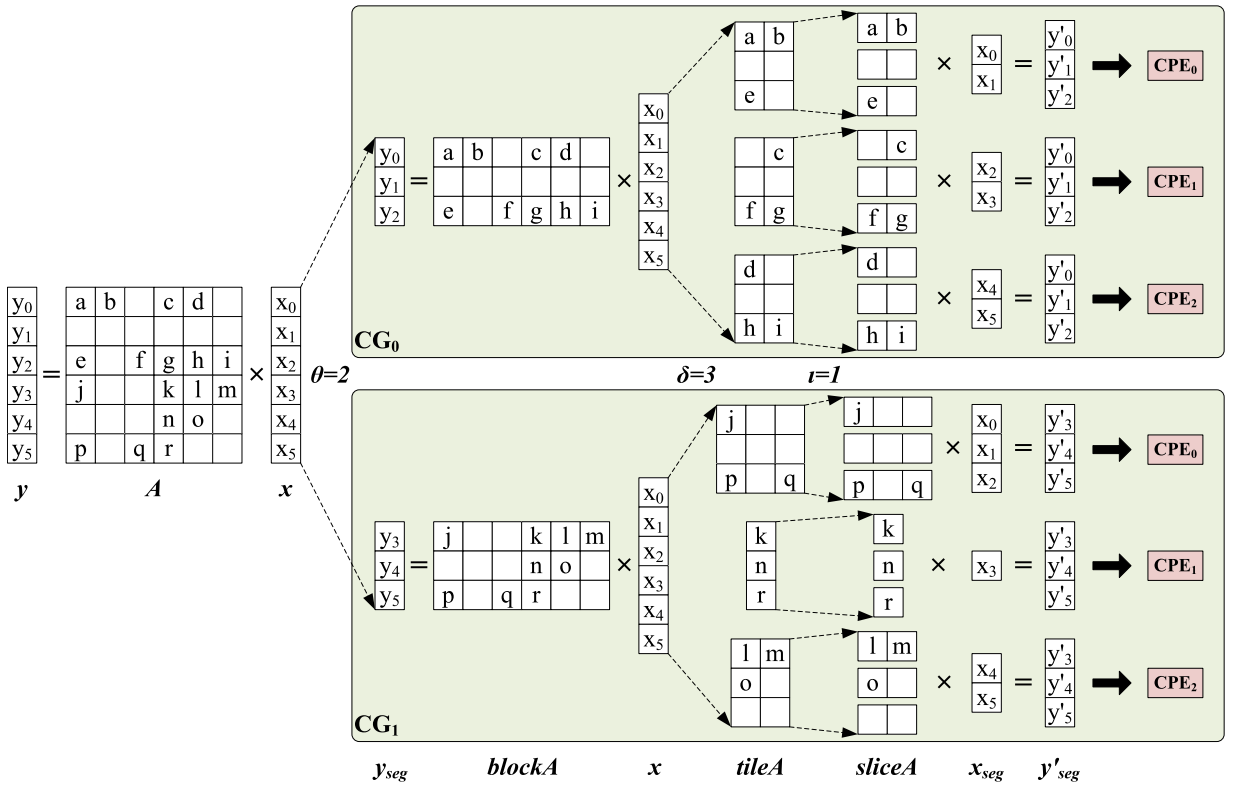

Fig. 3. Parallel partial CSR-based SpMV phase of `tpSpMV` on SW26010.

Pc[j]th column through the ***Pc***[$j+1$]th column of *blockA* to load the *tileA*, where $i \in \{0, 1, 2, \ldots, \delta - 1\}$. The result is a segment, denoted as y'_{seg}, which contains m/θ elements. Each CG will receive δ y'_{seg}s from the δ CPEs, and the δ y'_{seg}s will be further accumulated to the y_{seg} in the second phase.

- ***Layer 3:***
  To leverage the limited size of LDM, the *tileA* on each CPE is further divided by rows into $m/(\theta \times \iota)$ fine-grained *sliceA*s. Each *sliceA* contains ι rows.
  The LDM of each CPE loads a *sliceA* (ι rows) of the *tileA* each time and caches the x_{seg} for computations. The result each time on the CPE is a segment containing ι elements of the y'_{seg}.
  More importantly, the total size of the *sliceA*, the x_{seg}, and the result segment must adapt to LDM.

4.1.2. The parallelization model

Based on the adaptive partitioning method, we design the parallelization model for the partial CSR-based SpMV phase.

All the *sliceA*s of the *tileA* on each CPE are multiplied by the corresponding x_{seg}. According to the local memory caching technique, therefore, each x_{seg} is cached in the corresponding LDM via DMA at first. Each CPE executes $m/(\theta \times \iota)$ computing rounds in total. Each computing round loads a *sliceA* into the LDM for performing computations with the cached x_{seg}. There are three steps of each computing round on each CPE, as follows:

- ***Step 1:***
  The CPE loads the three arrays of a *sliceA* into LDM via DMA;
- ***Step 2:***
  The CPE performs computations on the loaded *sliceA* and the cached x_{seg};
- ***Step 3:***
  The CPE sends the result, a segment with ι elements of the y'_{seg}, back to main memory via DMA.

Fig. 3 presents an example of parallel partial SpMV phase of `tpSpMV` on SW26010. Algorithms 2 and 3 demonstrate the parallel partial SpMV phase on SW26010, where Algorithm 2 describes `tpSpMV` on an MPE, and Algorithm 3 describes the parallel partial CSR-based SpMV phase on a CPE. Algorithm 3 is called by Algorithm 2.

Algorithm 2 tpSpMV on the MPE of each CG.

Require: A, x, m, n, θ, and δ.
Ensure: *tileY*.
1: Read the *blockA* and x of the CG and hold it in the main memory;
2: //initiate CPE threads of the CG
3: athread_init();
4: //set the number of CPE threads engaged in the CG
5: athread_set_num_threads(θ);
6: //call Algorithm 3 on the θ CPEs to perform the partial CSR-based SpMV part
7: athread_spawn();
8: //deallocate the θ CPEs
9: athread_join();
10: //invoke Algorithm 4 on the θ CPEs to perform the accumulation part
11: athread_spawn();
12: //deallocate the θ CPEs
13: athread_join();
14: //annull the θ CPEs
15: athread_halt();
16: **return** *tileY*.

Algorithm 3 The partial CSR-based SpMV phase of tpSpMV on a CPE.

Require: m, n, θ, δ, and ι;
Three CSR arrays of the *blockA*: $\boldsymbol{Pr}[m/\theta+1]$, $\boldsymbol{Col}[nnz]$, and $\boldsymbol{Val}[nnz]$;
The array of x: $\boldsymbol{x}[n]$.
Ensure: The array of y'_{seg}: $\boldsymbol{y'_{seg}}[m/\theta]$.
1: Allocate memory in LDM ($\boldsymbol{x_{seg}}[n']$) for the array of x_{seg} on the CPE;
2: //load the array of x_{seg} in the LDM
3: *DMA_get*($\boldsymbol{x}, n', \boldsymbol{x_{seg}}$);
4: **for** each *sliceA* in the *tileA* of the *blockA* **do**
5: //load the CSR arrays of the *sliceA* in the LDM
6: *DMA_get*($\boldsymbol{Pr}, \iota, \cdots$);
7: Calculate the number of nonzeros of the *sliceA* (nnz(*sliceA*)) according to $\boldsymbol{Pr}$;
8: **if** $nnz(sliceA) \neq 0$ **then**
9: *DMA_get*($\boldsymbol{Col}, nnz(sliceA), \cdots$);
10: *DMA_get*($\boldsymbol{Val}, nnz(sliceA), \cdots$);
11: Perform SpMV on the *sliceA* and x_{seg}, and the result is a segment with ι elements of the y'_{seg};
12: //return the result segment back to the main memory
13: *DMA_put*($\cdots, \iota, \boldsymbol{y'_{seg}}$);
14: **end if**
15: **end for**
16: **return** $\boldsymbol{y'_{seg}}[m/\theta]$.

4.2. Parallel accumulation phase

The result segments got from CPEs of each CG, i.e., δ y'_{seg}s, should be further accumulated into the final result segment of y, denoted ad y_{seg}. In addition, the accumulation operations can be efficient using parallel techniques, which is what our parallel accumulation phase of tpSpMV does.

4.2.1. The adaptive partitioning

We design the adaptive partitioning method for the parallel accumulation phase on SW26010 as well. There are δ result segments, denoted as y'_{seg}s, of the partial CSR-based SpMV phase on each CG, and there are $\theta \times \delta$ y'_{seg}s in total. Before the accumulation phase, we merge the δ y'_{seg}s on each CG into a matrix *blockY* containing m/θ rows and δ columns.

As for each CG, there are two main layers of the adaptive partitioning:

- ***Layer 1:***
  To leverage the parallelism among δ CPEs within the CG, the *blockY* is partitioned into δ *tileY*s by rows, where each *tileY* contains $m/(\theta \times \delta)$ rows and δ columns.

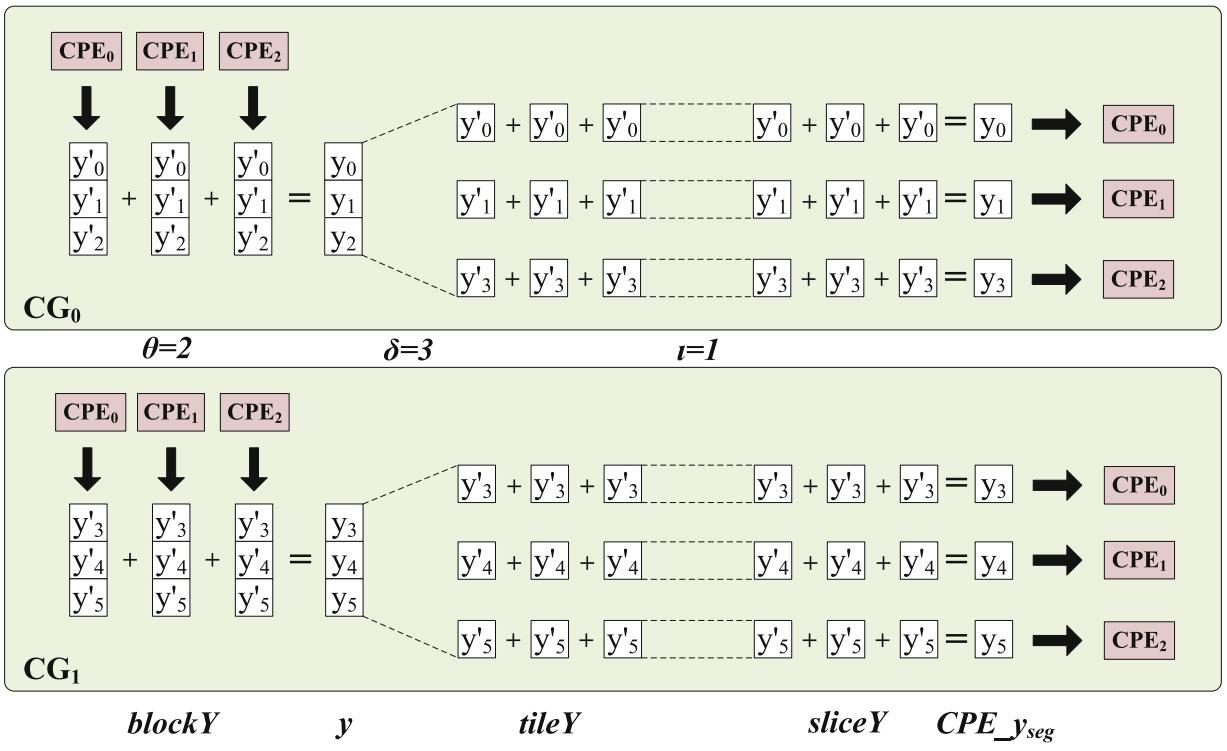

Fig. 4. Parallel accumulation phase of `tpSpMV` on SW26010.

Each CPE is assigned a *tileY*. The *j*th CPE of each CG reads the $\lceil \frac{j \times m}{\theta \times \delta} \rceil$th row through the $\lceil \frac{(i+1) \times m}{\theta \times \delta} \rceil$th row of the *blockY* to load the *tileY*, where $j \in \{0, 1, 2, \ldots, \delta - 1\}$. The accumulation result on all the δ columns of the *tileY* is a segment of y_{seg}, denoted as CPE_y_{seg}, which contains $m/(\theta \times \delta)$ elements.

- ***Layer 2:***
  To leverage the limited size of LDM, the *tileY* on each CPE is further divided by rows into $m/(\theta \times \delta \times \iota)$ *sliceY*s. Each *sliceY* contains ι rows and δ columns.
  Each CPE loads a *sliceY* (ι rows) of the *tileY* each time. The accumulation result on all the δ columns of a *sliceY* is a segment of CPE_y_{seg} that contains ι elements.
  More importantly, the total size of each *sliceY* and the corresponding result segment must adapt to the LDM.

4.2.2. The parallelization model

Based on the adaptive partitioning method, we further design the parallelization model for the accumulation phase on SW26010.

Each CPE only loads a *sliceY* each time for computations. Therefore, each CPE executes $m/(\theta \times \delta \times \iota)$ computing rounds in total. There are three steps of each computing round of the parallel accumulation phase, as follows:

- ***Step 1:***
  Each CPE loads a *sliceY* into the LDM via DMA;
- ***Step 2:***
  Each CPE performs accumulation operations on all the δ columns of the *sliceA*. The result is a segment of CPE_y_{seg}, denoted as $CPE\_y'_{seg}$, which contains ι elements;
- ***Step 3:***
  Each CPE sends the result segment $CPE\_y'_{seg}$ back to main memory via DMA.

Fig. 4 presents an example of the parallel accumulation phase of `tpSpMV` on SW26010. Algorithms 2 and 4 describe the parallel accumulation phase of `tpSpMV` on SW26010, where Algorithm 4 describes the parallel accumulation phase of `tpSpMV` on a CPE. In addition, Algorithm 4 is called by Algorithm 2.

As for the parallel partial CSR-SpMV phase, the ***Layer 1*** partitioning step decides the computational loads for accumulation phase. To guarantee load balance for parallel accumulation phase, each *blockA* has m/θ rows. In addition, the ***Layer 2*** partitioning step partitions *blockA* based on the number of nonzeros and CPEs to guarantee load balance among CPEs within each CG. As for the parallel accumulation phase, each *blockY* is dense. Therefore, the ***Layer 1*** partitioning step that guarantees each *tileY* has $m/(\theta \times \delta)$ rows can ensure load balance among CPEs.

Algorithm 4 The accumulation phase of tpSpMV on a CPE.

Require: m, n, θ, δ, and ι;
The array of the *blockY*: ***blockY***$[m \times n/\theta]$.
Ensure: The array of CPE_y_{seg}: ***CPE_y_seg***$[m/(\theta \times \delta)]$.
1: **for** each *sliceY* in the *tileY* of the *blockY* **do**
2: //load the array of the *sliceY* in the LDM
3: *DMA_get* (***blockY***, $\iota \times \delta, \cdots$);
4: Perform accumulation on the δ columns of the *sliceA*, and the result is a segment with ι elements of the CPE_y_{seg};
5: //return the result segment back to the main memory
6: *DMA_put* ($\cdots, \iota$, ***CPE_y_seg***);
7: **end for**
8: **return** ***CPE_y_seg***$[m/(\theta \times \delta)]$.

5. Communication optimization for tpSpMV

The communication amount and communication overhead of tpSpMV increase as the data scale increases, resulting in the communication overhead may become the main bottleneck of the whole kernel. The communication of our tpSpMV corresponds to the intra-node communications on Sunway system, i.e., the data swapping between main memory and LDM. Therefore, the emphasises of communication optimization are on reducing amount of transmission data and increasing the transmission bandwidth of DMA on each CG. We propose the following schemes to optimize DMA transmission.

5.1. Data reduction

As shown in Fig. 3, there are empty rows in A and empty columns in *blockA*s in tpSpMV. Empty rows of A result in redundant transmission data of y'_{seg} and the CSR array ***Pr*** in the partial CSR-based SpMV phase, and *blockY*s and CPE_y_{seg} in the accumulation phase. Empty columns of *blockA*s result in redundant transmission data of x in the partial CSR-based SpMV phase.

Therefore, the data reduction technique eliminates the zeros in input matrix A to reduce the amount of transmission data. Fig. 5 shows the partial CSR-based SpMV using data reduction technique. As shown in Fig. 5, the data reduction technique eliminates the empty rows of A before performing two-phase execution to reduce the redundant transmission data of y'_{seg} and the CSR array ***Pr***. In addition, the technique reduces the empty columns of *blockA*s and corresponding elements in x after the ***Layer 1*** partitioning to reduce the redundant transmission data of x.

5.2. Aligned memory accessing

The DMA transmission performance can be improved when each data transmission chunk is aligned to a 128-byte boundary in main memory. Each DMA transmission accesses the memory that aligns to a 128-byte boundary. As for transferring a 128-byte data chunk, for example, two DMA transmissions are actually required for a total of 256 bytes if the data chunk is not aligned to a 128-byte boundary in memory. However, only one DMA transmission is actually required for a total of 128 bytes if the data chunk is aligned to a 128-byte boundary in memory.

The aligned memory accessing in the proposed large-scale SpMV is hard to guarantee due to the irregular positions of nonzeros of sparse matrices. Therefore, we design the optimization technique that guarantees the aligned memory accessing in large-scale SpMV. Define M_i as the memory footprint of an integer and M_f as the memory footprint of a floating-point number on the platform.

As for the partial CSR-based SpMV phase, each CPE accesses the three CSR arrays of a *sliceA* each time. Therefore, we propose the padding technique that pads each *sliceA* with zeros to ensure that the number of nonzeros. The padded zeros of each *sliceA* are a multiple of both $\frac{128}{M_i}$ and $\frac{128}{M_f}$, which guarantee the aligned memory accessing to CSR arrays ***Col*** and ***Val***. In addition, there are ι integers of ***Pr*** transferred from main memory to LDM. To guarantee the aligned memory accessing to ***Pr***, we set the value of ι to be a multiple of $\frac{128}{M_i}$. Moreover, there are ι floating-point numbers of $\boldsymbol{y'_{seg}}$ transferred from LDM to main memory. To guarantee the aligned memory accessing to $\boldsymbol{y'_{seg}}$, we set the value of ι to be a multiple of $\frac{128}{M_f}$.

As for the parallel accumulation phase, each CPE accesses a *sliceY* with $\iota \times \delta$ floating-point numbers each time. To guarantee the aligned memory accessing to each *blockY*, we set the value of $\iota \times \delta$ to be a multiple of $\frac{128}{M_f}$. In addition, each CPE transfers ι floating-point numbers of ***CPE_y_seg*** returned from LDM to main memory. Therefore, we set the value of ι to be a multiple of $\frac{128}{M_f}$ to guarantee the aligned memory accessing to ***CPE_y_seg***.

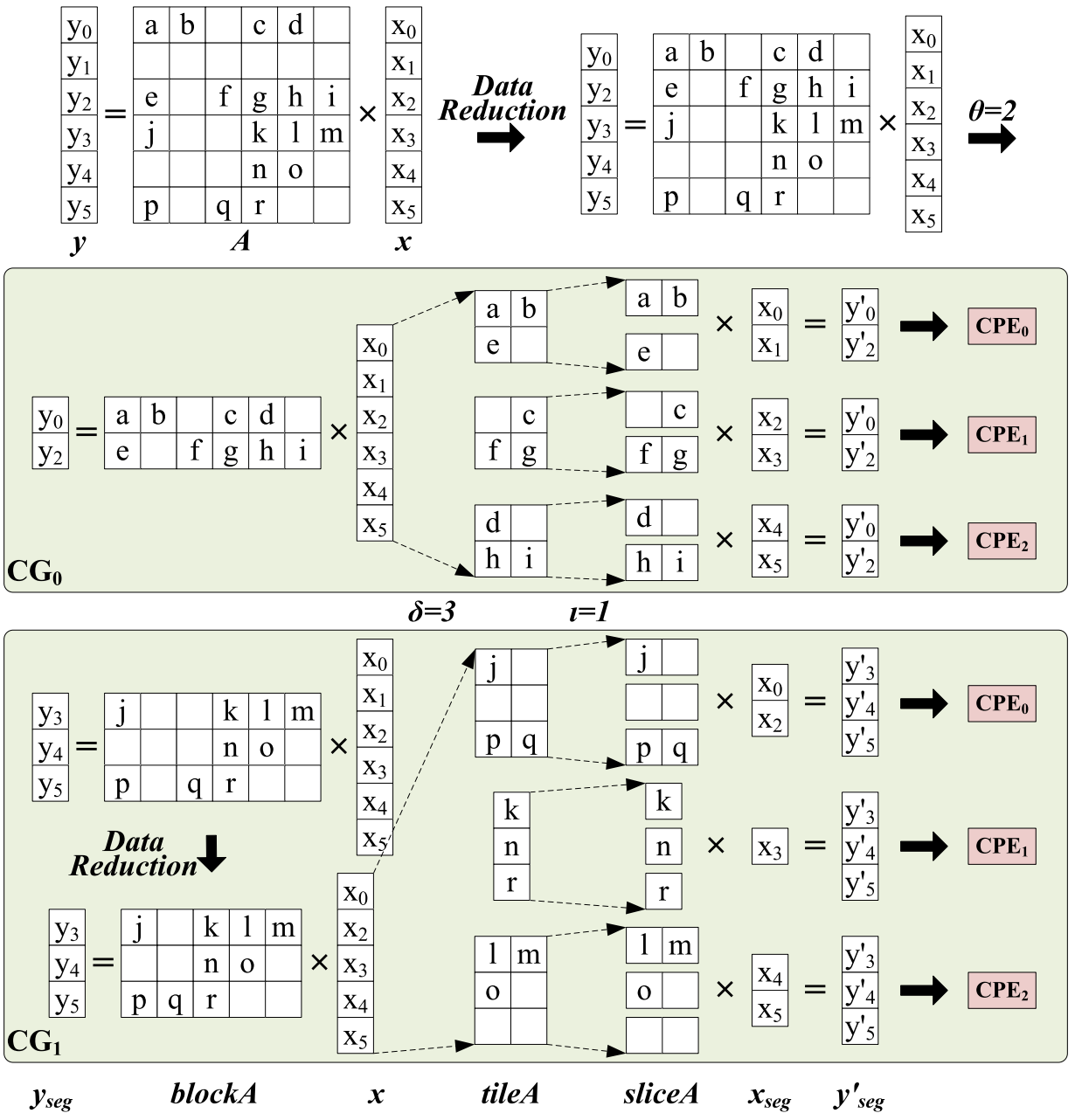

Fig. 5. Data reduction technique for the partial CSR-SpMV phase on SW26010.

5.3. Pipelining

As for the partial CSR-SpMV phase, each CPE firstly caches the x_{seg} in LDM, and then successively loads a *sliceA* (***step 1***), executes computations (***step 2***), and sends the y'_{seg} back (***step 3***). The three steps in each computing round are executed in sequential. In addition, the partial CSR-SpMV phase is executed on each CPE till the CPE completes computations on the entire *tileA* and sends the results back. As for the accumulation phase, each CPE successively loads a *sliceY* (***step 1***), executes computations (***step 2***), and sends the *CPE_*y_{seg} back (***step 3***). The three steps in each computing round are executed in sequential. In addition, the accumulation phase is executed on each CPE till the CPE completes computations on the entire *tileY* and sends the results back.

We use the pipeline technique to create parallelism among the data loading step (***step 1***) in previous computing rounds, computation execution step (***step 2***) in current computing rounds, and results returning step (***step 3***) in next computing rounds. As shown in the timeline of Fig. 6, the performance of `tpSpMV` can be improved using the pipeline technique.

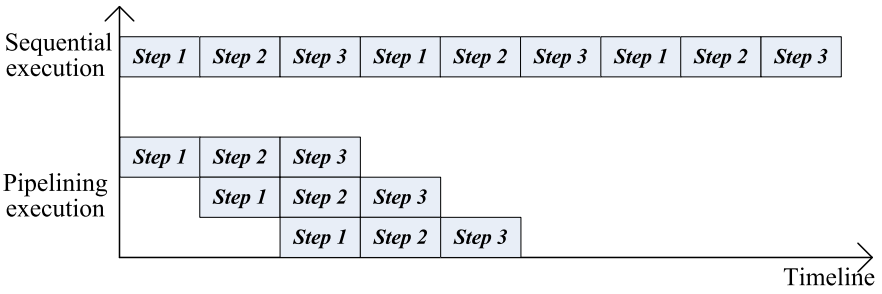

Fig. 6. Pipeline technique.

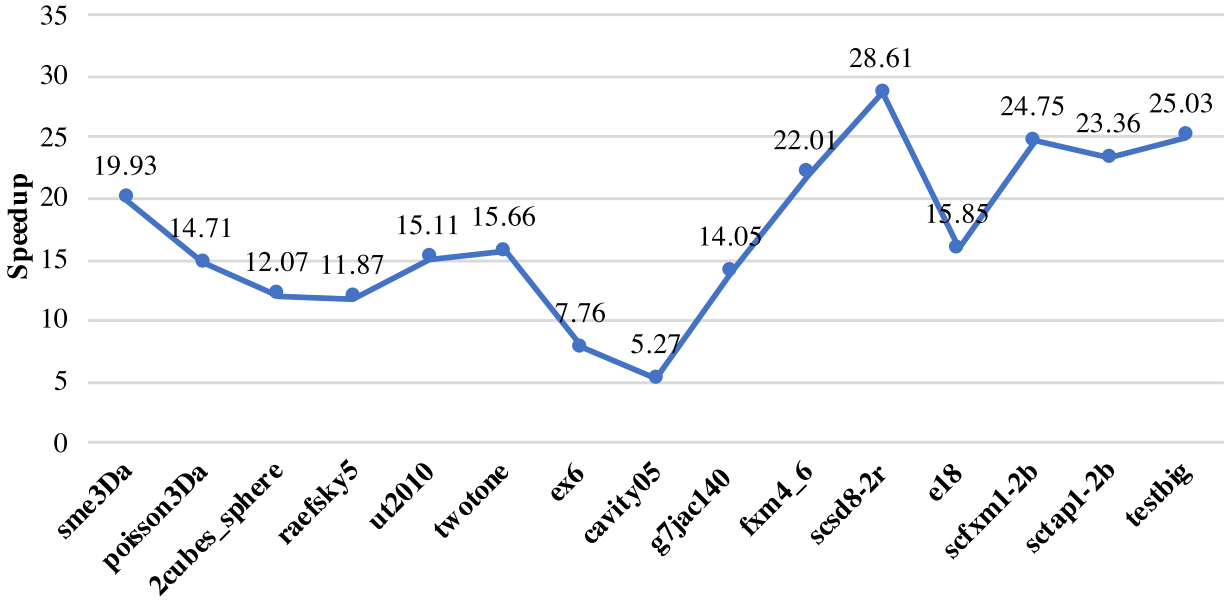

Fig. 7. Speedup of `tpSpMV` on a CG with 64 CPEs.

6. Performance evaluation

6.1. Experimental setup

`tpSpMV` is implemented by C language. This paper tests `tpSpMV` on SW26010 CPUs of the Sunway TaihuLight supercomputer. The frequency of each MPE and CPE of an SW26010 CPU is 1.45GHz. We first test the scalability of `tpSpMV` on CPEs within a single CG. Then, we test the scalability of `tpSpMV` on eight CGs.

We choose 15 sparse matrices, as shown in Table 1, that are frequently used in some related works [33], where m in Table 1 means the number of rows of the sparse matrix, n means the number of columns, and nnz means the number of nonzeros.

6.2. Experimental results and analysis

Fig. 7 presents the speedups of `tpSpMV` achieved on a CG using 64 CPEs. The speedup is the ratio of `tpSpMV`'s parallel running time executed on 64 CPEs within a CG to the sequential running time executed on an MPE. On average, `tpSpMV` achieves the speedup of 17.12 (min: 5.05, max: 28.61) on the 15 matrices on a CG using 64 CPEs, as shown in Fig. 7. We notice that `tpSpMV` obtains the lowest speedups on *ex6* and *cavity05*, the smallest matrices of all the test matrices, indicating that `tpSpMV` performs better on large-scale matrices than small-scale matrices.

Fig. 8 shows GFlops of `tpSpMV` achieved on a CG as the number of CPEs changes, where GFlops is calculated based on the ratio of the double number of nonzeros of the sparse matrix to the execution time of `tpSpMV`. `tpSpMV` on a CG achieves the highest GFlops when all the 64 CPEs are used. In addition, the growth rate of GFlops on a CG slows down as the number of CPEs increases, since multiple CPEs working in the CG experience contention for computing resources.

Table 1
The sparse matrices.

Name	*m*	*n*	*nnz*
sme3Da	12504	12504	874887
poisson3Da	13514	13514	352762
2cubes_sphere	101492	101492	874378
raefsky5	6316	6316	168658
ut2010	115406	115406	111052
twotone	120750	120750	1224224
ex6	1651	1651	49533
cavity05	1182	1182	32747
g7jac140	41490	41490	565956
fxm4_6	47185	22400	265442
scsd8-2r	60550	8650	190210
e18	38602	24617	156466
scfxm1-2b	33047	19036	111052
sctap1-2b	33858	15390	99454
testbig	31223	17613	61639

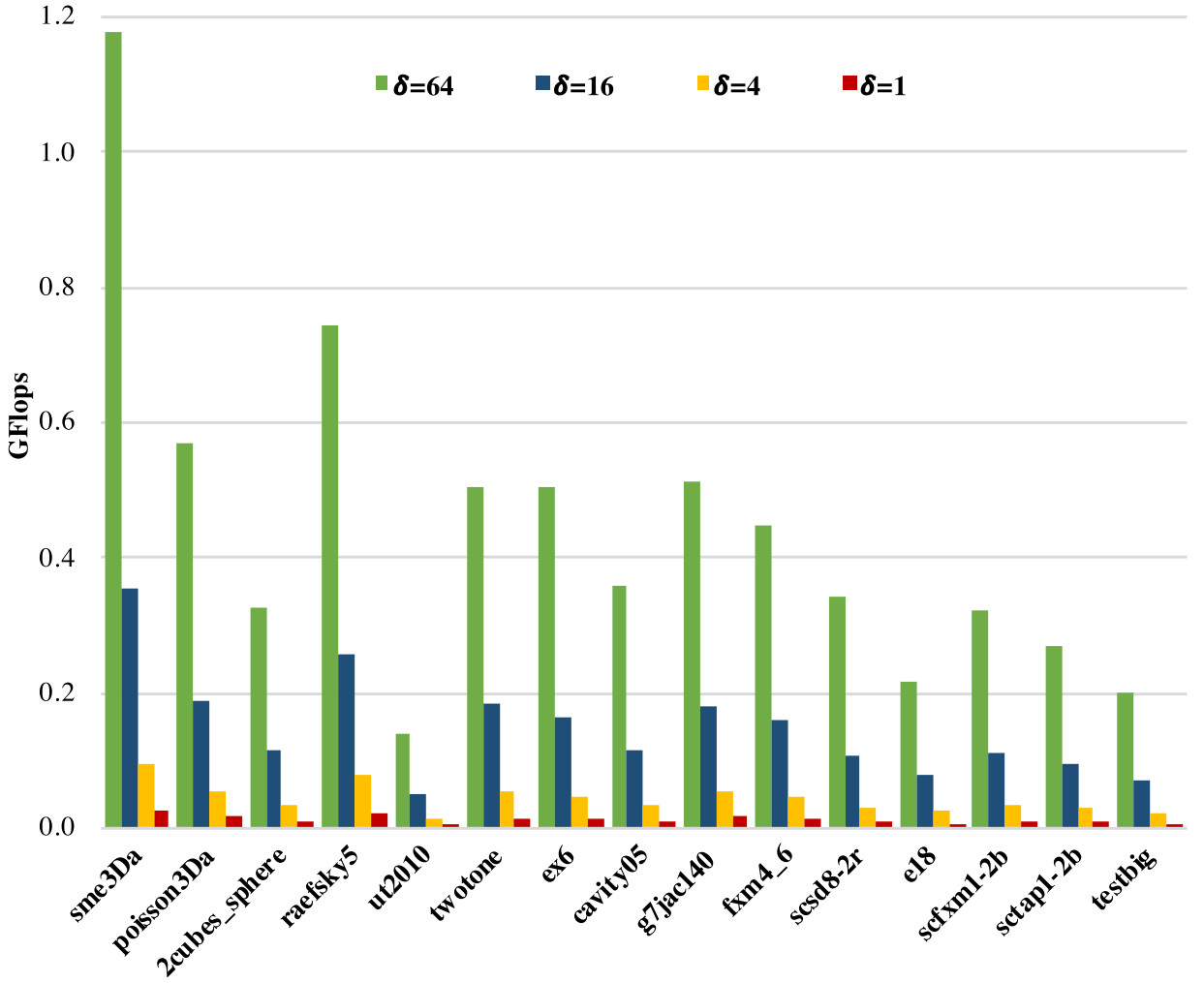

Fig. 8. Performance of `tpSpMV` on a CG varying the number of CPEs.

Fig. 9 shows GFlops of `tpSpMV` achieved on the Sunway varying the number of CGs. The GFlops obtained by `tpSpMV` improves with the increasing of the number of CGs. `tpSpMV` yields good scalability on eight CGs. Moreover, as the number of CGs increases, *blockA*s become sparser and the number of empty rows and columns of *blockA*s increases, which leads to the data reduction technique playing an increasingly important role in performance optimization of `tpSpMV`. Therefore, the data reduction technique avoids the performance degradation caused by sparsity of sparse matrices.

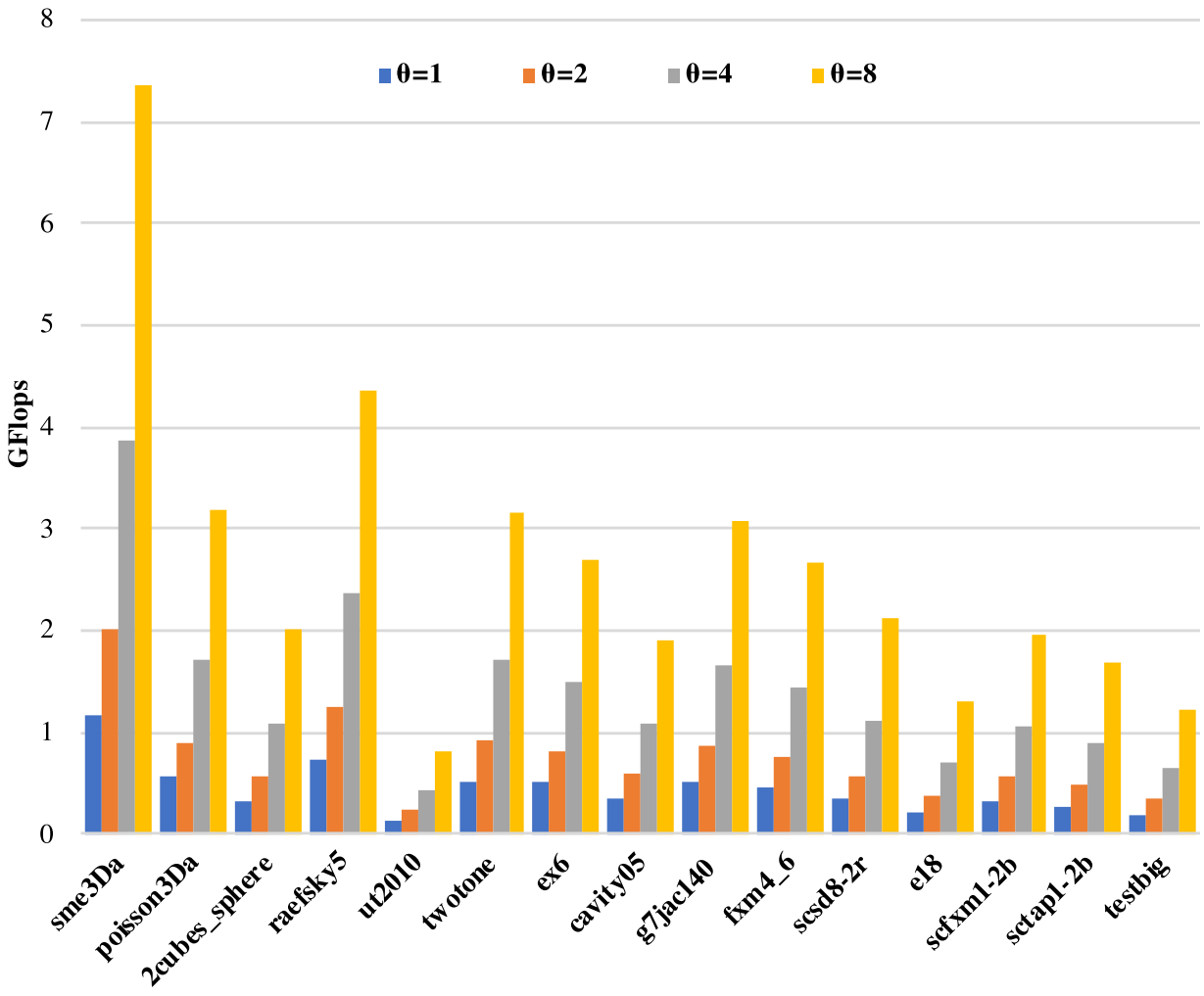

Fig. 9. Scalability of tpSpMV on CGs.

Fig. 10 presents performance of the three optimization techniques. We compare performance of tpSpMV with that without optimization techniques. tpSpMV gains 6.29% (min: 3.57%, max: 9.57%) performance improvement on average by using the proposed optimization techniques. Figs. 11 and 12 show the optimization effects for the partial SpMV phase and accumulation phase, respectively. On average, by using optimizations, the partial SpMV phase and accumulation phase of tpSpMV achieve performance improvements of 7.60% (min: 3.95%, max: 10.74%) and 3.98% (min: 2.11%, max: 6.90%), respectively. The reason why the accumulation phase yields less performance improvements than the partial SpMV phase is that each *blockY* is dense and the size of each *sliceY* is a multiple of $64 \times M_f$ when 64 CPEs are used in each CG, which causes that the aligned memory accessing techniques yield no performance gain for the accumulation phase.

We further present performance contributions to tpSpMV from each of the three optimization techniques, i.e., data reduction, aligned memory accessing, and pipelining, as shown in Fig. 13. Label "tpSpMV Without Optimization" presents the execution time of tpSpMV without any optimization. Label "Data Reduction" shows the execution time of tpSpMV using the data reduction. Label "Aligned Memory Accessing" presents the execution time of tpSpMV using the data reduction and aligned memory accessing techniques. Label "Pipelining" shows the execution time of tpSpMV using all the three techniques. By comparing label "tpSpMV Without Optimization" and label "Data Reduction", the optimization effect of data reduction is 0.31% on average (min: 0.00%, max: 4.63%). By comparing label "Data Reduction" and label "Aligned Memory Accessing", the optimization effect of align memory accessing is 3.21% on average (min: 1.48%, max: 5.50%). By comparing label "Aligned Memory Accessing" and label "Pipelining", the optimization effect of pipelining is 2.88% on average (min: 1.25%, max: 5.96%). The data reduction optimization seems not effective, because all the tested sparse matrices except *ut2010* have no empty rows and columns, and tpSpMV gains performance improvement from data reduction only on *ut2010* (4.63%).

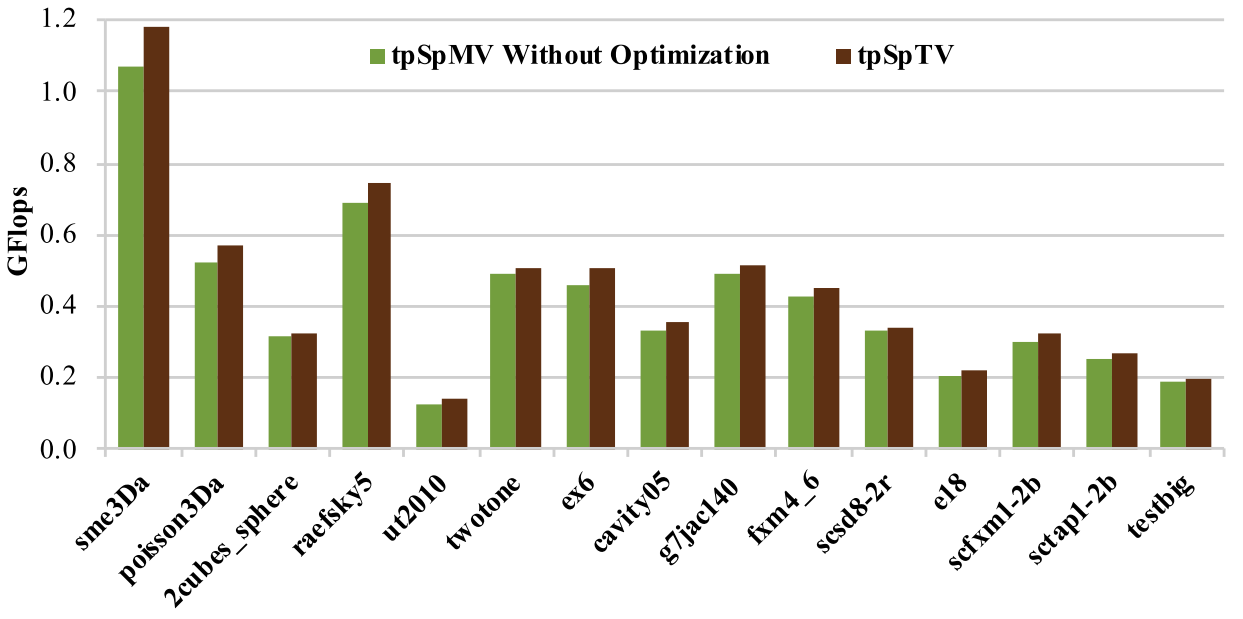

Fig. 10. Effects of the communication optimization techniques for `tpSpMV`.

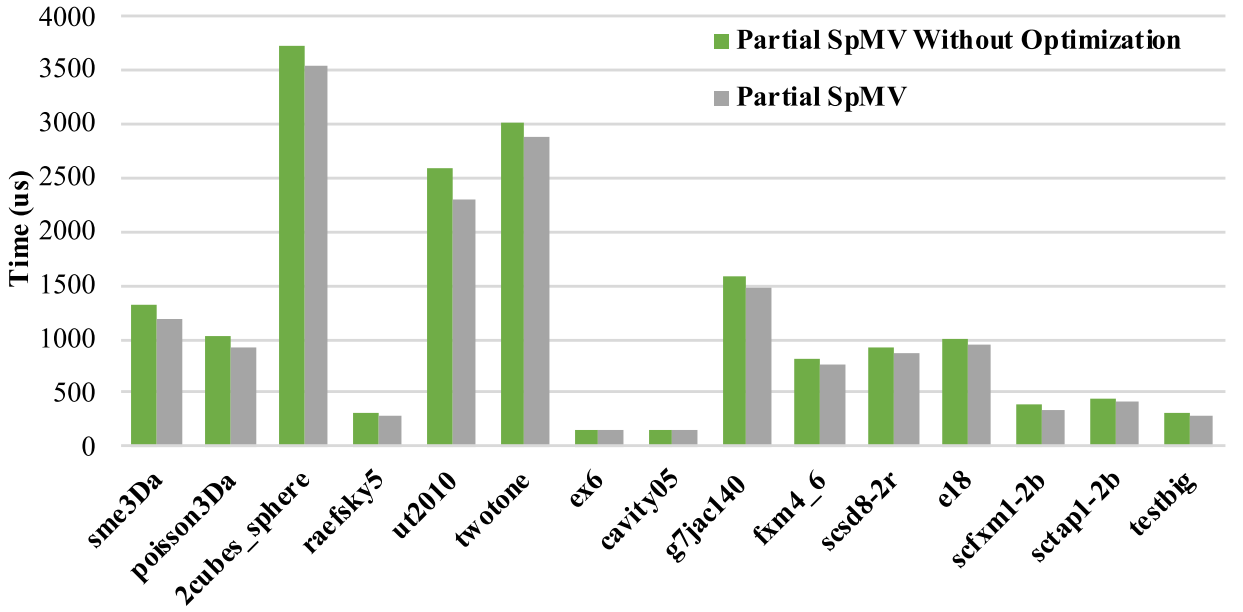

Fig. 11. Optimization effects for the partial SpMV phase.

Fig. 14 shows the proportion of execution time of the partial CSR-based SpMV phase and the accumulation phase in `tpSpMV` on 64 CPEs within a CG. Label "Partial SpMV" represents the running time of the partial CSR-based SpMV phase, and label "Accumulation" represents the running time of the accumulation phase. The parallel running time of the accumulation phase is less than that of the partial CSR-based SpMV phase. According to the previous experimental analysis, the computational data in the parallel accumulation phase is dense, which enables the parallel computing resources to be more fully utilized.

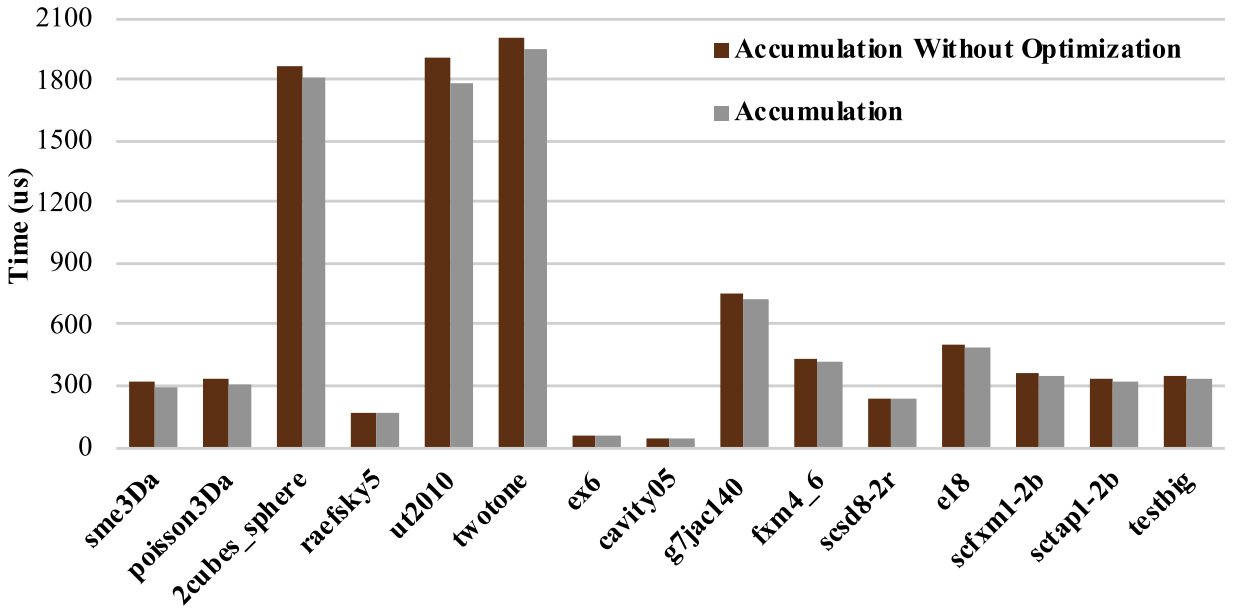

Fig. 12. Optimization effects for the accumulation phase.

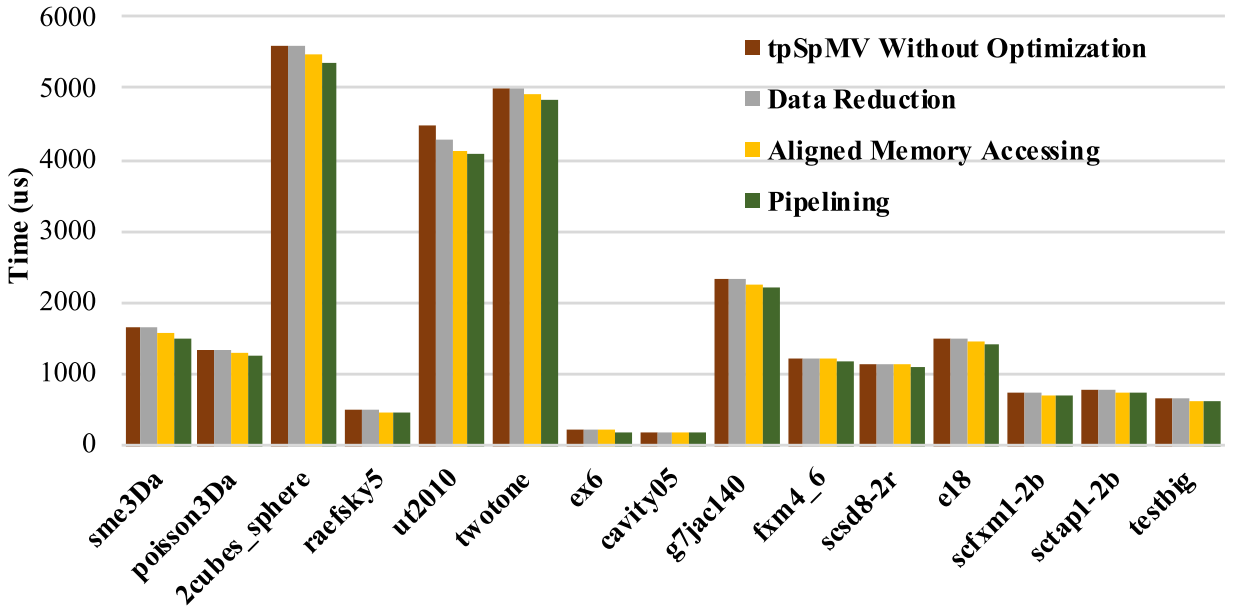

Fig. 13. Performance contributions to `tpSpMV` from each of the three optimization techniques.

Fig. 15 compares the performance of `tpSpMV` with the work reported in Ref. [34]. It is evident from the figure that our `tpSpMV` outperforms than the work in [34] on a CG. `tpSpMV` achieves the performance improvement of 13.16% on average (max: 21.63%, min: 7.58%). The reason is that the work in [34] does not consider load balancing among CPEs within a CG for the parallel partial CSR-based SpMV phase and the communication optimization techniques.

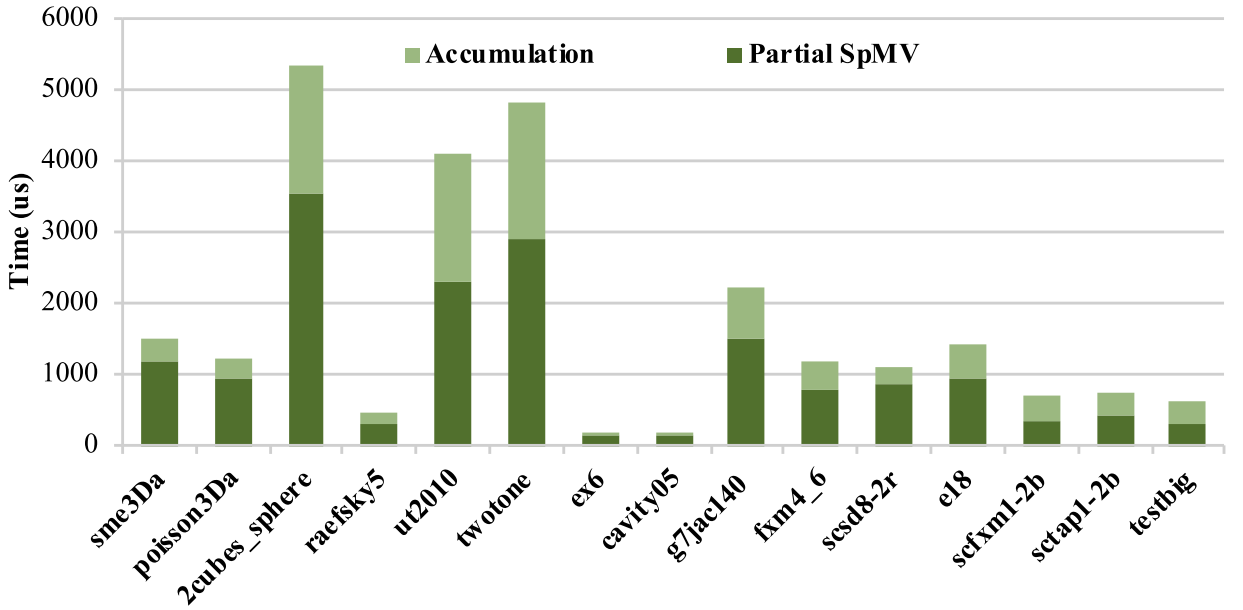

Fig. 14. Performance of the parallel partial SpMV phase and the accumulation phase of `tpSpMV` on a CG.

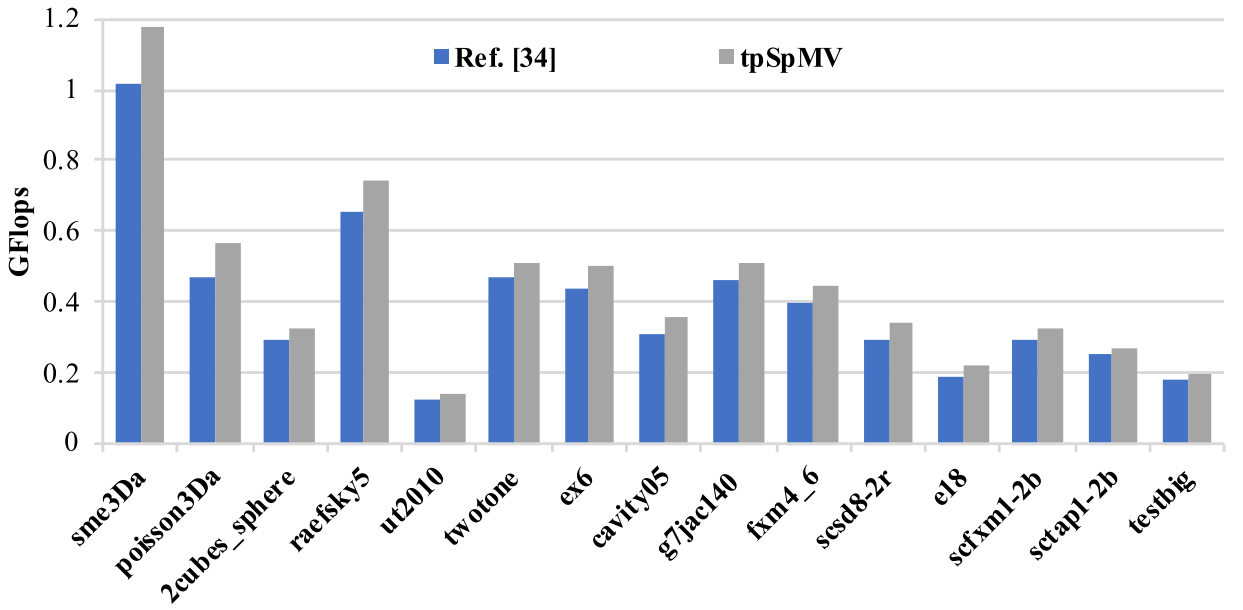

Fig. 15. GFlops comparison between `tpSpMV` and the work in Ref. [34] on a CG.

7. Conclusions

This paper proposes a high-performance and large-scale two-phase SpMV kernel, named as `tpSpMV`, on manycore architectures that alleviates three challenges of computational scale limitation, high memory access latency, and low bandwidth usage. The proposed `tpSpMV` mainly includes two parts: the parallel partial CSR-based SpMV phase and the parallel accumulation phase. We propose the adaptive partitioning methods and parallelization designs for the two parts to make full use of computational resources, respectively. We further design communication optimization techniques for `tpSpMV` to enable more efficient bandwidth usage. The performance evaluation of `tpSpMV` on SW26010 processors presents high efficiency and fine scalability.

As for the future work, we will optimize graph computations using sparse matrix algebra on high-performance computing platforms.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Yuedan Chen: Conceptualization, Methodology, Writing - original draft. **Guoqing Xiao:** Conceptualization, Methodology, Writing - review & editing. **Fan Wu:** Writing - review & editing, Validation. **Zhuo Tang:** Writing - original draft, Validation. **Keqin Li:** Writing - original draft.

Acknowledgments

The authors would like to thank the three anonymous reviewers for their valuable and helpful comments on improving the manuscript. This paper extends our preliminary work published in the 21st IEEE International Conference on High Performance Computing and Communications (HPCC 2019) and proposes a more thorough study of the problem [34]. The research was partially funded by the National Key R&D Programs of China (Grant Nos. 2018YFB0203800, 2018YFB1003401) and the Programs of National Natural Science Foundation of China (Grant Nos. 61625202, 61806077).

References

[1] Y. Umuroglu, M. Jahre, An energy efficient column-major backend for FPGA SPMV accelerators, in: 32nd IEEE International Conference on Computer Design, ICCD 2014, Seoul, South Korea, October 19–22, 2014, 2014, pp. 432–439.
[2] P. Grigoras, P. Burovskiy, E. Hung, W. Luk, Accelerating SPMV on FPGAs by compressing nonzero values, in: 23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2–6, 2015, 2015, pp. 64–67.
[3] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, L. Zhang, CVR: efficient vectorization of SPMV on x86 processors, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24–28, 2018, 2018, pp. 149–162.
[4] E. Coronado-Barrientos, G.I. Fernández, A.J. García-Loureiro, AXC: a new format to perform the SPMV oriented to intel xeon phi architecture in opencl, Concurr. Comput. 31 (1) (2019).
[5] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, X. Liu, Towards efficient SPMV on sunway manycore architectures, in: Proceedings of the 32nd International Conference on Supercomputing, ICS 2018, Beijing, China, June 12–15, 2018, 2018, pp. 363–373.
[6] J. Zhang, C. Zhou, Y. Wang, L. Ju, Q. Du, X. Chi, D. Xu, D. Chen, Y. Liu, Z. Liu, Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13–18, 2016, 2016, pp. 34–45.
[7] K. Li, W. Yang, K. Li, Performance analysis and optimization for SPMV on GPU using probabilistic modeling, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 196–205.
[8] S. Yan, C. Li, Y. Zhang, H. Zhou, yaSPMV: yet another SPMV framework on GPUs, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15–19, 2014, 2014, pp. 107–118.
[9] B. Su, K. Keutzer, clSPMV: a cross-platform opencl SPMV framework on GPUs, in: International Conference on Supercomputing, ICS'12, Venice, Italy, June 25–29, 2012, 2012, pp. 353–364.
[10] W. Yang, K. Li, K. Li, A pipeline computing method of SPTV for three-order tensors on CPU and GPU, ACM Trans. Knowl. Discov. Data 13 (6) (2019) 63:1–63:27, doi:10.1145/3363575.
[11] S. Salinas, C. Luo, X. Chen, W. Liao, P. Li, Efficient secure outsourcing of large-scale sparse linear systems of equations, IEEE Trans. Big Data 4 (1) (2018) 26–39.
[12] P. Suryanarayana, P.P. Pratapa, J.E. Pask, Alternating anderson-richardson method: an efficient alternative to preconditioned Krylov methods for large, sparse linear systems, Comput. Phys. Commun. 234 (2019) 278–285.
[13] E. Totoni, M.T. Heath, L.V. Kalé, Structure-adaptive parallel solution of sparse triangular linear systems, Parallel Comput. 40 (9) (2014) 454–470.
[14] P. Ghale, H.T. Johnson, A sparse matrix-vector multiplication based algorithm for accurate density matrix computations on systems of millions of atoms, Comput. Phys Commun. 227 (2018) 17–26.
[15] C.P. Ribeiro, J. Hutter, J. VandeVondele, Improving communication performance of sparse linear algebra for an atomistic simulation application, in: Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10–13 September 2013, Garching (near Munich), Germany, 2013, pp. 405–414.
[16] P. Herholz, T.A. Davis, M. Alexa, Localized solutions of sparse linear systems for geometry processing, ACM Trans. Graph. 36 (6) (2017) 183:1–183:8.
[17] A. Buluç, K. Madduri, Parallel breadth-first search on distributed memory systems, in: Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12–18, 2011, 2011, pp. 65:1–65:12.
[18] A. Azad, A. Buluç, A. Pothen, A parallel tree grafting algorithm for maximum cardinality matching in bipartite graphs, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25–29, 2015, 2015, pp. 1075–1084.
[19] Y. Chen, G. Xiao, W. Yang, Optimizing partitioned CSR-based SPGEMM on the sunway taihulight, Neural Comput. Appl. (2019) 1–12.
[20] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, T. Li, Performance-aware model for sparse matrix-matrix multiplication on the sunway taihulight supercomputer, IEEE Trans. Parallel Distrib. Syst. 30 (4) (2019) 923–938.
[21] F. Sadi, J. Sweeney, T.M. Low, J.C. Hoe, L.T. Pileggi, F. Franchetti, Efficient SPMV operation for large and highly sparse matrices using scalable multi-way merge parallelization, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12–16, 2019., 2019, pp. 347–358.
[22] G. Xiao, K. Li, Y. Chen, W. He, A. Zomaya, T. Li, CaSPMV: a customized and accelerative SPMV framework for the sunway taihulight, IEEE Trans. Parallel Distrib. Syst. (2019).
[23] D. Merrill, M. Garland, Merge-based sparse matrix-vector multiplication (SPMV) using the CSR storage format, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12–16, 2016, 2016, pp. 43:1–43:2.
[24] J.L. Greathouse, M. Daga, Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16–21, 2014, 2014, pp. 769–780.
[25] G. Xiao, Y. Chen, C. Liu, X. Zhou, ahSPMV: an auto-tuning hybrid computing scheme for SPMV on the sunway architecture, IEEE Internet Things J. 7 (3) (2020) 1736–1744, doi:10.1109/JIOT.2019.2947257.
[26] Y. Zhang, S. Li, S. Yan, H. Zhou, A cross-platform SPMV framework on many-core architectures, TACO 13 (4) (2016) 33:1–33:25.
[27] W. Liu, B. Vinter, CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication, in: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015, 2015, pp. 339–350.

[28] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, P. Sadayappan, Fast sparse matrix-vector multiplication on GPUs for graph applications, in: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16–21, 2014, 2014, pp. 781–792.
[29] A. Elafrou, G.I. Goumas, N. Koziris, BASMAT: bottleneck-aware sparse matrix-vector multiplication auto-tuning on GPGPUs, in: Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16–20, 2019, 2019, pp. 423–424.
[30] M.O. Karsavuran, K. Akbudak, C. Aykanat, Locality-aware parallel sparse matrix-vector and matrix-transpose-vector multiplication on many-core processors, IEEE Trans. Parallel Distrib. Syst. 27 (6) (2016) 1713–1726.
[31] A. Elafrou, G.I. Goumas, N. Koziris, Performance analysis and optimization of sparse matrix-vector multiplication on modern multi- and many-core processors, in: 46th International Conference on Parallel Processing, ICPP 2017, Bristol, United Kingdom, August 14–17, 2017, 2017, pp. 292–301.
[32] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned SPMV on GPUs and multicore CPUs, IEEE Trans. Comput. 64 (9) (2015) 2623–2636.
[33] T.A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25.
[34] Y. Chen, G. Xiao, F. Wu, Z. Tang, Towards large-scale sparse matrix-vector multiplication on the SW26010 manycore architecture, in: 21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10–12, 2019, 2019, pp. 1469–1476.