Multi-Role SpTRSV on Sunway Many-core Architecture

Mingzhen Li, Yi Liu, Hailong Yang, Zhongzhi Luan, Depei Qian
School of Computer Science and Engineering
Beihang University, Beijing, China
Email: {lmzhhh, yi.liu, hailong.yang, 07680, depeiq}@buaa.edu.cn

***Abstract*—Sparse triangular solver (SpTRSV) is an important and indispensable building block for many scientific applications. The parallelism of SpTRSV is exploited using Level-Set method in literature, however this method still suffers from high synchronization cost and irregular global memory access especially on many-core architecture such as Sunway. In this paper, we propose an efficient implementation of SpTRSV using the massive computing resources on Sunway architecture. Specifically, we divide the 64 CPEs in a core group into three different roles, *worker*, *router* and *storer*. We also build a logical shared memory by carefully manipulating the scratchpad memory located in each *storer* and allow synchronization using the unique register communication on Sunway architecture. We partition the sparse matrix into multiple *bands* and replace the irregular global memory accesses with shared memory accesses, which significantly improves the data locality during the calculation of a *band*. Our experiments with 12 representative datasets demonstrate that our approach achieves up to 5.14× (2.65× on average) speedup.**

I. Introduction

Sparse triangular solver (SpTRSV) is an important and indispensable building block of various applications from many domains, including traditional domains such as computational fluid dynamics (CFD) and molecular dynamics (MD), as well as emerging domains such as big data and machine learning. SpTRSV solves a system of linear equations $Lx = b$, where L is a sparse lower triangular matrix, b is a dense vector. The SpTRSV returns the solution vector x. SpTRSV is also the most important part of sparse linear solvers. Algorithm for sparse linear solver can be divided into two categories, direct method and iterative method. The direct method contains two steps, matrix decomposition and SpTRSV. The iterative method usually uses SpTRSV as pre-conditioner to reduce iteration steps, such as pre-conditioned conjugate gradient (PCG).

SpTRSV exhibits limited parallelism in the literature due to data dependency between each element of *x* and part of its preceding elements, where *x[i]* can only be calculated after the completion of *x[i-1]*. The level-set method [1] exploits the parallelism of SpTRSV from the sparsity of matrix L. It divides the elements of *x* into several levels. Elements within a level can be calculated in parallel, whereas calculations across different levels need to be synchronized due to the inter-level dependency. The sparse matrix calculation of SpTRSV exhibits irregular memory access patterns, which leads to poor data locality and prohibits a high-performance implementation.

Sunway TaihuLight supercomputer ranks the first place in the top500 list, whose peak performance is 125PFlops. It is equipped with 40960 pieces of SW26010 many-core processors. Although SW26010 is a brand new many-core architecture, it has already been demonstrated with promising computation capability in many application domains such as Dynamic Model [2] and Earthquake Simulation [3]. Besides, many important kernels have been optimized on Sunway and exhibit significant speedup such as Stencil [4], DNN [5], Graph Traversal [6] and MD [7]. However, to implement SpTRSV efficiently on Sunway, we need to leverage the unique architecture features such as Local Data Memory and register communication (details in Section II-A) to adapt the computation of SpTRSV to Sunway processor.

In this paper, we propose Multi-Role SpTRSV (MR-SpTRSV) scheme on Sunway architecture, which leverages the unique design of Sunway to address the challenges for high-performance SpTRSV implementation, such as synchronization and irregular memory access. To address the synchronization, we divide the 64 CPEs into three roles, *worker*, *router* and *storer*. Also, we build a logical shared memory based on the scratchpad memory of each *storer*, which is shared by all *workers*. The *routers* take charge of routing data transfers between arbitrary cores, which solves the limitation of register communication. Through our multi-role re-design of the CPEs, *workers* can synchronize data at very low cost using register communication. Also, the design of logical shared memory also increases data reuse during calculation.

To address the irregular memory access, we partition the sparse lower/upper triangular matrix into *rect_blocks* and *tri_blocks*, based on whether there are non-zero values residing above/below the diagonal. We refer all blocks in the same row as a *band*. *rect_blocks* can be calculated by *workers* in parallel without synchronization, elements of *x* from the left-hand side are sent to logical shared memory using register communication. Whereas for *tri_blocks*, elements of *x* from both left-hand side and right-hand side are sent to logical shared memory using register communication. With such matrix partition, all the irregular memory accesses are replaced by shared memory accesses, which enables data reuse during the calculation of a *band*.

We implement our scheme on Sunway SW26010 processor using CSC format and compare it with several the-state-of-the-art parallel SpTRSV algorithms. Our experiment results

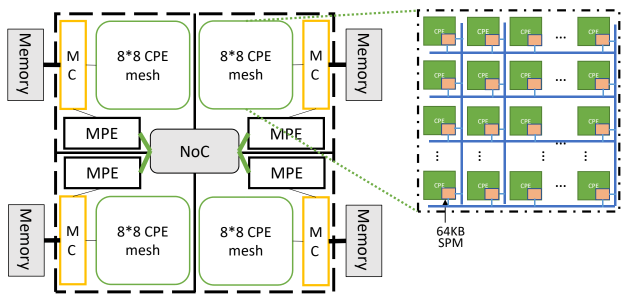


Fig. 1. The architecture of the Sunway SW26010.

with representative datasets demonstrate that our scheme can better utilize the architecture features of Sunway and thus achieves higher performance improvement. Specifically, this paper makes the following contributions:

- We propose a partitioning mechanism that divides the sparse matrices into blocks and uses the LDM to cache the required elements of *x*, which enables data reuse during the calculation.
- We propose a multi-role SpTRSV (MR-SpTRSV) scheme that divides the CPEs into three different roles, which work collaboratively to support efficient synchronization between arbitrary cores using register communication.
- We implement MR-SpTRSV scheme on Sunway SW26010 processor and demonstrate its effectiveness by comparing with the-state-of-the-art parallel SpTRSV algorithms using representative datasets.

This paper is organized as follows. Section II describes the background and our motivation to implement SpTRSV on Sunway. Section III presents our methodology for an efficient SpTRSV implementation on Sunway. Section IV presents the experiment results with our approach. Section V describes the related work and section VI concludes our work.

II. Background and Motivation

In this section, we introduce the Sunway SW26010 architecture and describe the serial and parallel algorithm of SpTRSV. Also, we highlight the challenges for implementing SpTRSV efficiently on Sunway architecture.

A. Sunway SW26010 Many-Core Processor

Sunway TaihuLight supercomputer ranks first place in top500 list, which is equipped with 40960 pieces of SW26010 many-core processors. SW26010 many-core processor works at the frequency of 1.45GHz with the peak performance of 3.06GFLOPS in double precision. The architecture is shown in Figure 1. Each SW26010 processor contains 4 Core Groups (CGs), where each CG consists of a Memory Controller (MC), a Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs). MPE and CPEs play different roles. MPE is similar to modern CPU processor, which supports the complete interrupt functions, memory management, superscalar and out-of-order issue/execution. Whereas, CPEs are simplified to achieve high computing throughput.

Regarding memory hierarchy on SW26010, each CG has 8GB main memory, which is shared by MPE and all CPEs. Each MPE has a 32KB L1 data cache and a 256KB L2 cache for both instruction and data. Each CPE has a 16KB L1 instruction cache and a 64KB programmable Scratchpad Memory (SPM), which is also referred as Local Data Memory (LDM). The LDM can be configured as a limited fast memory, a software-emulated cache, or hybrid, which is fully controlled by the programmer. CPEs can use both Direct Memory Access (DMA) and normal global load/store (Gload/Gstore) instructions to access data in main memory. However, DMA outperforms in both bandwidth and latency than Gload/Gstore.

Moreover, CPEs are organized in 8×8 mesh, which supports register communication between CPEs in the same row or the same column at low latency. Each register communication operation can get/put 256bits data, and each CPE has 6×288bits (256bits for data and 32bits for ECC checksum) send buffer shared by both row and column, and 4×256bits receive buffer for row and column separately. If a CPE tries to put data when the send buffer is full, its pipeline stops until the send buffer is available again. If a CPE tries to put data when the receive buffer on the target CPE is full, its pipeline also stops until the receive buffer on target CPE is available. Therefore it is essential for programmers to manage the register communication carefully to achieve better performance and throughput without introducing deadlock.

B. Serial and Parallel SpTRSV

Algorithm 1 Serial SpTRSV with CSC format.

```
1: $x \leftarrow b$
2: for $i = 0$ to $numCols - 1$ do
3:     for $j = ColPtr[i] + 1$ to $ColPtr[i+1] - 1$ do
4:         $x[RowIdx[j]] \leftarrow x[RowIdx[j]] + Val[j] \times x[i]$
5:     end for
6: end for
```

In this paper, we use matrix L, vector x and vector b to describe the algorithm of SpTRSV. We assume that L is a lower triangular matrix with unit diagonal. When processing general lower triangular matrix $\hat{L}$ without a unit diagonal, we first factorize matrix $\hat{L} = DL$, where D is a diagonal matrix whose diagonal elements is the same as $\hat{L}$, and L is the unit lower triangular matrix. The SpTRSV algorithm solves the linear system $Lx = b$, and returns the solution x. L is stored in CSC format. There are three auxiliary vectors with CSC format: vector *Val* stores the values of the non-zero element, vector *RowIdx* stores the row indices of the non-zero elements and vector *ColPtr* stores the indices of the first non-zero element of each column in vector *Val* and vector *RowIdx*.

As shown in Algorithm 1, the serial SpTRSV has no concurrency due to data dependency between each element of x and part of its preceding elements. Before $x[i]$ updates the value of $x[j] (j > i)$, $x[i]$ itself should be updated by $x[k] (k < i)$ first. Therefore, the strong data dependency prevents an easy parallel implementation of SpTRSV that calculates different elements of x concurrently.

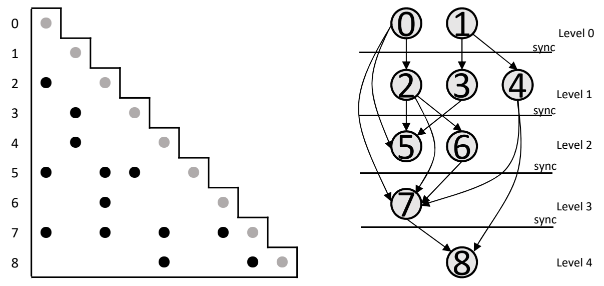


Fig. 2. The lower triangular matrix and its level-set implementation.

In literature, level-set method [1] is proposed to exploit the hidden parallelism of SpTRSV from the sparsity of matrix *L*. By carefully analyzing the sparse matrix *L*, a set of elements in *x* can be identified so that all elements in this set are independent of each other and can be calculated in parallel.

Take the matrix in Figure 2 (left) for example, the level-set implementation is shown in Figure 2 (right), we divide this matrix into five levels. It is clear that we can parallelize the calculation within level 0, 1 and 2. However, synchronization should be enforced after the calculation at each level to make sure that the latest data is available for next level.

C. Challenges for Parallel SpTRSV on Sunway

We identify several challenges to implement parallel SpTRSV on Sunway many-core architecture efficiently, which motivates this paper.

Synchronization: Although we can design a parallel SpTRSV with the level-set method, the synchronization is still required due to data dependency across levels. Unfortunately, there is no shared memory among CPEs on Sunway. Therefore, we need to leverage the unique register communication for synchronization in order to avoid the long latency for accessing main memory. Since the CPEs are organized as 8×8 mesh, it only supports register communication between the cores within the same row or column. Therefore, to enable synchronization between arbitrary cores, new communication method needs to be designed to mitigate the limitation of register communication and provide efficient synchronization across cores.

Irregular memory access: The sparse matrix is usually stored in the compressed format such as Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC). However, accessing the compressed matrix causes irregular memory accesses in SpTRSV implementation. For example, a lower triangular $n \times n$ matrix *L* stored in CSC format, the nonzeros of *L* in the same column is stored in neighboring memory blocks, which means that the value of *RowIdx* varies from 0 to n during the calculation. Since the size of LDM on CPEs is only 64KB, it is impossible to store all elements in LDM when *x* is large. In that case, without careful management of the data accesses, the CPEs have to access main memory for *x* discretely through slow Gload/Gstore instructions rather than through DMA in batch.

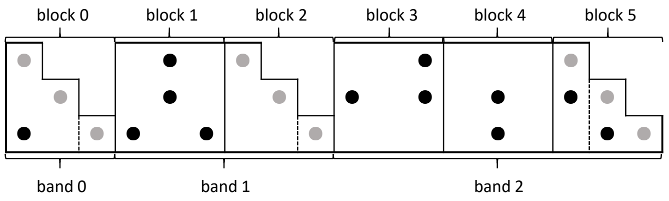


Fig. 3. The block partition of sparse matrix.

Correspondingly, we propose a multi-role SpTRSV approach to address the challenges on Sunway architecture.

- We divide the compressed CSC matrices into blocks so that we can control the range of *RowIdx* to access and cache the required elements of *x* into LDM. In this way, we can reuse elements of *x* to achieve better performance.
- We adapt the communication pattern of SpTRSV to Sunway architecture by dividing the 64 CPEs into three roles, *worker*, *router* and *storer*. The LDM of each *Storer* is shared by all *workers*. During the SpTRSV computation, *workers* can exchange data with *storers* through register communication, which can better utilize the limited LDM through fast register communication.

III. Implementation

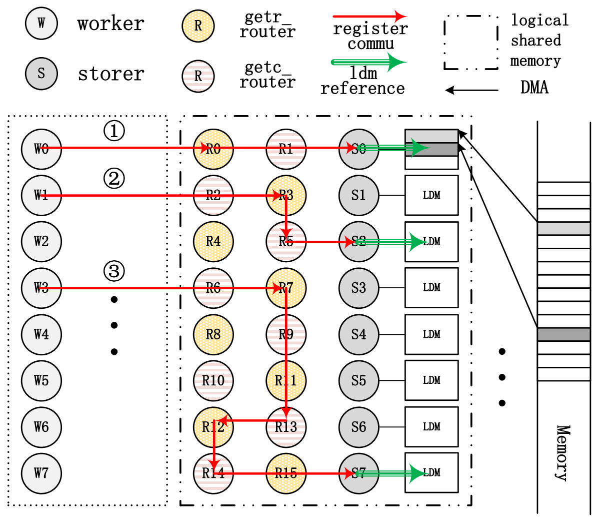
A. Partitioning the Matrix

As shown in Figure 3, we partition the lower triangular matrix into square blocks. The size of each block is fixed, which is referred as *SL* in our implementation. The setting of *SL* is further discussed in Section III-C. Depending on whether there are non-zero values above the diagonal, we classify blocks into 2 types, *rect_block* and *tri_block*.

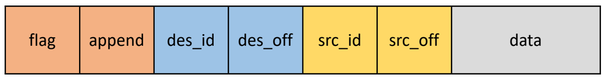
For example, *block 0*, *block 2* and *block 5* belong to *tri_block*, whereas *block 1*, *block 3* and *block 4* belong to *rect_block*. A level may stretch to several blocks due to block partition. For example, level 1 may stretch to both *block 0* and *block 1*. We apply a customized Level-Set for each *tri_block*. A *band* contains blocks in the same row. The ID of each *band* is set to the row number in the original matrix. Then we perform SpTRSV *band* by *band*.

B. Dividing CPEs into Multiple Roles

As shown in Figure 1, each CPE has a 64KB LDM. However, CPEs cannot access the LDMs of each other. In other words, the LDM is private for each CPE. When implementing SpTRSV on Sunway architecture, it is preferred to have a fast logical memory shared by all CPEs. So that CPEs could exchange data with each other through this shared on-chip memory instead of through main memory with long latency. The key idea in our implementation to build such logical shared memory is to leverage the unique register communication between CPEs on Sunway. However, there is a hardware limitation on Sunway that a CPE can only send/receive data to/from CPEs within the same row or same column. A software mechanism is required to enable the data communication between any pair of CPEs.



(a) The role assignment of CPEs, and three typical communication situations.



(b) The format of data packet in register communication.

Fig. 4. The multi-role implementation of SpTRSV on Sunway architecture.

To enable the pair-wise communication between CPEs, we divide all 64 CPEs into three roles such as *worker*, *router* and *storer* as shown in Figure 4a. Besides, we propose a new format for storing data in the register to support this role assignment. The size of the packet for register communication is *32 bytes*, based on which we divide the register data into two parts: one for header information and the other for the value, as shown in Figure 4b. The first 24 bytes are occupied by the header information, which contains six variables. *flag* denotes packet type (request or update), *append* denotes whether *workers* have finished their work, *des_id* and *src_id* denote the IDs of receiver and sender respectively, *des_off* and *src_off* denote the data offsets in LDMs of receiver and sender respectively. The last 8 bytes are occupied by value.

Workers are responsible for performing the calculation. They load the required data from main memory or *routers*, perform the calculation, and send results to *routers*. *Workers* always send/receive data packets to/from the *routers* in the same row.

Routers are responsible for forwarding the data packet. *Routers* are further divided into 2 types, *getr_router* and *getc_router* to avoid deadlock. We assign two columns of CPEs as *routers*. In the first column, *storers* with even *row numbers* are assigned as *getr_routers*. In the second column, *storers* with odd *row numbers* are assigned as *getr_routers*. Other *storers* in both columns are assigned as *getc_routers*. Each *getr_router* always receives data packet along the same row, and sends data packet along the same column. Each *getc_router* always receives data packet along the same column, and sends data packet along the same row. Each *router* reads the ID of the receiver from the data packet, if the receiver is in the same row with the router, it directly forwards the data packet to the CPE of the receiver. Otherwise, it forwards the data packet to next *router*.

Storers are responsible for providing their LDMs as logical shared memory to *Workers*. Each *storer* receives and sends the data packets along the same row. If $flag = 0$, it updates the value $x[des\_off]$ in its own LDM. If $flag = 1$, it reads the value $x[des\_off]$ in its own LDM, switches the *des_info* and *src_info*, inserts $x[des\_off]$ in data packet, and then sends it to *routers*. After the calculation of each *band*, *storers* update the data in main memory through DMA.

As for *workers* and *storers*, they only need to communicate with *routers* along the same row. The *storers* decide locally whether to store the data in their own LDMs or continue to forward the packet to next *routers*. The processing logic of *router* is listed in Algorithm 2. To better illustrate, we explain three different situations handled by the *router* as shown in Figure 4a ① to ③.

Situation 1: *worker* and *storer* are in the same row. *Worker 0* needs to send a data packet to *storer 0*, first it sends the packet to *router 0*, because *router 0* is a *getr_router* in the same row with *worker 0*. When *router 0* receives the packet, it reads *des_id*, finds out that CPE with *des_id* is in the same row, it then just forwards this packet to CPE with *des_id* through register communication. *Storer 0* receives and stores the packet in the *row receive buffer*.

Situation 2: If the difference between the row number of the *worker* and the *storer* is odd, for example, *worker 1* needs to send a data packet to *storer 2*, it first sends the packet to *router 3*, because *router 3* is a *getr_router* in the same row with *worker 1*. When *router 3* receives the packet, it reads *des_id*, finds out that CPE with *des_id* is in *row 2*, while itself is in *row 1*. The difference is odd, *router 3* forwards packet to *router 5*, because *router 5* is a *getc_router* in the same column with *router 5* and in the same row with *des_id* CPE. Then *router 5* receives and forwards this packet to *des_id* CPE through register communication.

Situation 3: If difference between the row number of the *worker* and the *storer* is even, for example, *worker 3* needs to send a data packet to *storer 7*, it first sends the packet to *router 7*, because *router 7* is a *getr_router* in the same row with *worker 3*. When *router 3* receives the packet, it reads *des_id*, finds out that *des_id* CPE is in *row 7*, while itself is in *row 1*. The difference is odd, *router 7* forwards packet to *router 13*, because *router 13* is the nearest *getc_router* to *des_id* CPE and within the range that *router 7* can reach. After *router 13* receives the packet and the difference of row number becomes odd. The processing logic is repeated similarly to *Situation 2*.

Through role assignment, the LDMs of *storers* are shared by all *workers*. We assign one column of CPEs as *storers*, therefore the *workers* have $8 \times 64KB = 512KB$ logical shared memory at most. With this method, most main memory accesses with Gload/Gstore are converted to LDM assess using register communication.

Algorithm 2 The processing logic of *Router*.

```
1:  while (1) do
2:      if $role == getr\_router$ then
3:          $RegRecvRow(packet)$
4:      else
5:          $RegRecvCol(packet)$
6:      end if
7:      if $packet.flag == -64$ then
8:          $Break$
9:      end if
10:     $\Delta \leftarrow ROW(des\_id) - ROW(my\_id)$
11:     if $\Delta == 0$ then
12:         $RegSendRow(packet, ROW(dest\_id))$
13:         $Continue$
14:     end if
15:     if $role == getr\_router$ then
16:         if $\Delta == odd$ then
17:             $RegSendCol(packet, ROW(dest\_id))$
18:             $Continue$
19:         end if
20:         if $\Delta == even$ then
21:             if $\Delta > 0$ then
22:                 $RegSendCol(packet, ROW(dest\_id) - 1)$
23:             else
24:                 $RegSendCol(packet, ROW(dest\_id) + 1)$
25:             end if
26:             $Continue$
27:         end if
28:     end if
29:     if $role == getc\_router$ then
30:         if $\Delta == odd$ then
31:             if $COL(my\_id) == first\_router\_column$ then
32:                 $RegSendRow(packet, COL(my\_id) + 1)$
33:                 $Continue$
34:             end if
35:             if $COL(my\_id) == second\_router\_column$ then
36:                 $RegSendRow(packet, COL(my\_id) - 1)$
37:                 $Continue$
38:             end if
39:         end if
40:     end if
41: end while
```

Algorithm 3 The processing logic of *Storer*.

```
1:  $MallocMEM[SL/StorerNum]$
2:  for $band\_id = 0 \rightarrow bdnum - 1$ do
3:      /*Iterate through all bands*/
4:      for $cycle = 0 \rightarrow cnum - 1$ do
5:          $bgn \leftarrow (role\_id + cycle * StorerNum) * \Theta$
6:          $end \leftarrow (role\_id + cycle * StorerNum + 1) * \Theta$
7:          $DMA\_GET(x[bgn : end], MEM)$
8:      end for
9:      while (1) do
10:         $RegRecvRow(packet)$
11:         if $packet.flag == -64$ then
12:             $Break$
13:         end if
14:         if $packet.flag == 0$ then
15:             /*Update local data*/
16:             $MEM[dest\_off] -= packet.data$
17:         end if
18:         if $packet.flag == 1$ then
19:             /*Send corresponding data*/
20:             $RegSendRow(ROW(packet.src\_id, MEM[dest\_off])$
21:         end if
22:     end while
23:     for $cycle = 0 \rightarrow cnum - 1$ do
24:         $bgn \leftarrow (role\_id + cycle * StorerNum) * \Theta$
25:         $end \leftarrow (role\_id + cycle * StorerNum + 1) * \Theta$
26:         $DMA\_PUT(MEM[cycle * \Theta, (cycle + 1) * \Theta], x[bgn, end])$
27:     end for
28: end for
```

C. Implementation on Sunway

In this section, we elaborate on the implementation of our Multi-Role SpTRSV on Sunway. Although our approach is designed for Sunway, it is generally applicable to other manycore architectures, especially with cache-less design.

As described in Section III-A and III-B, the whole matrix is divided into *blocks* and *bands*. As for *block*, its size is referred as *SL*. The *blocks* in the same row form a *band*. So the height of a band is also *SL*. *SL* is set to the maximum number of double-precision floating point elements our logical shared memory can store. In this way, our logical shared memory can store all elements in *x* that needs to be updated. As for the CPEs, We assign the five columns (i.e., 40 CPEs on the leftmost) to be *workers*, the one column (i.e., 8 CPEs on the rightmost) to be *storers*, and the last two columns left (i.e., CPEs in *column 5* and *column 6*) to be *routers*. One example of such role assignment is shown in Figure 4a.

Algorithm 3 elaborates the processing logic of *storers*. At the beginning of calculating each *band*, *storers* preload elements into LDM. The data in LDM is stored using one-dimensional cyclic block distribution (line 4-8). The parameter Θ represents block size. We adopt this data distribution to ensure the load balance across *storers*. The *storer* receives data packet along the row (line 10). If $packet.flag = -64$, the packet is a quit packet (line 11-13). If $packet.flag = 0$, *storer* updates the corresponding element in LDM (line 14-17). If $packet.flag = 1$, *storer* sends corresponding data to Row($packet.dest\_id$) (line 18-21). Before finishing, *storer* updates corresponding data in main memory by DMA.

Algorithm 4 The processing logic of *Worker*.

```
1:  for $band\_id = 0 \rightarrow bdnum - 1$ do
2:      /*Iterate through all bands*/
3:      $buffer\_x\_get[L\_GET\_SIZE]$, $buffer\_x\_put[L\_GET\_SIZE]$
4:      $blknum \leftarrow band(i).blknum$
5:      for $block\_id = 0 \rightarrow bnum - 1$ do
6:          /*Iterate through all blocks*/
7:          $flag \leftarrow block.flag$, $blksize \leftarrow block.blksize$
8:          if $flag == 0$ then
9:              /*solve rect_block*/
10:             $pnum \leftarrow blksize/L\_GET\_SIZE$
11:             for $pid = 0 \rightarrow pnum - 1$ do
12:                 /*Parallel execution by workers*/
13:                 $DMA\_GET\_SIZE \leftarrow L\_GET\_SIZE$
14:                 $DMA\_GET(RowIdx, ColIdx, Val)$
15:                 for $k = 0 \rightarrow L\_GET\_SIZE - 1$ do
16:                     if $x\_bgn < ColIdx[k]$ then
17:                         $DMA\_GET\_SIZE \leftarrow \Omega$
18:                         $DMA\_GET(buffer\_x\_get)$
19:                     end if
20:                     $buffer\_x\_get[k] \leftarrow buffer\_x\_get[k] - Val[k] *$
                        $buffer\_x\_get[ColIdx[k] - x\_bgn]$
21:                 end for
22:                 /*Update result to Storers*/
23:                 $RegSendRow(buffer\_x\_get, RowIdx)$
24:             end for
25:         end if
26:         if $flag == 1$ then
27:             /*solve tri_block*/
28:             for $l = 0 \rightarrow levels - 1$ do
29:                 /*Iterate through all levels*/
30:                 /*Calculation similar with rect_block*/
31:                 /*Except loading required $x$ from logical shared memory*/
32:                 $SyncWorkers()$
33:             end for
34:         end if
35:     end for
36:     $CALL\_QUIT()$
37: end for
```

Algorithm 4 lists the implementation details of the *worker*. *Band* and *block* denote the set of *bands* and *blocks* of the

TABLE I
THE SPARSE MATRIX DATASETS.

Matrix	rows/columns	lower_nnz	levels	parallelism
ASIC_680k	682682	2277355	60	11378
eu-2005	862664	19235140	2726	316
wiki-Talk	2394385	3072221	515	4649
webbase-1M	1000005	2348442	512	1953
FEM_Accelerator	121192	1383436	239	507
kron_g500-logn17	131071	5245057	1361	96
nlpkkt80	1062400	14883536	2	531200
in-2004	1382908	9834648	1320	1048
Dense	2000	2001000	2000	1
dc2	116835	441781	14	8345
road_usa	23947347	52801659	77	311005
road_central	14081816	31015229	59	238675

sparse matrix respectively, *flag* denotes whether a block is *tri_block*, $buffer\_x\_get$ denotes the buffer for loading *x* and $buffer\_x\_put$ denotes the buffer for storing intermediate results. The *workers* process the matrix by *band*, and within a *band*, it processes the *band* by *block*, as shown in Algorithm 4. When processing a *rect_block*, the non-zeros in the block are divided into several parts; each part has L_GET_SIZE number of non-zeros. Each *worker* can process one part (line 11) and load the non-zeros of *Val*, *RowIdx* and *ColIdx* through DMA. When loading the required *x*, *workers* also prefetch a certain number of elements from *x* into LDM (line 16-19). The *workers* perform the calculation and add the result to the corresponding element in $buffer\_x\_put$ (line 20). Then they send $buffer\_x\_put$ to our logical shared memory (line 23). When processing a *tri_block*, the non-zeros in every level are divided into several parts, each part has $L\_GET\_LEVEL\_SIZE$ non-zeros. When loading the required *x*, *workers* just need to load from our logical shared memory(line 31). Then the *workers* perform the calculation, update elements in $buffer\_x\_put$ and send $buffer\_x\_put$ using register communication. After the calculation of each level, there is a synchronization between all *workers* to ensure correctness (line 32). When all blocks have been processed, the *workers* finish their work and send quit packet to *routers* and *storers* (line 36), which then exit the execution.

IV. EVALUATION

A. Experimental Setup

Our experiments are conducted on a CG of a Sunway SW26010 processor. The performance of SpTRSV on a CG is important, because most scientific applications decompose their computation into CGs and perform SpTRSV on one CG intensively. To evaluate our MR-SpTRSV, we select 12 representative sparse matrices from the University of Florida Sparse Matrix Collection [8] as listed in Table I. All experiments use double precision. For comparison, we implement two other state-of-the-art parallel SpTRSV algorithms on SW26010 architecture and use Serial-SpTRSV as our baseline.

Serial-SpTRSV: We implement the serial SpTRSV on MPE of Sunway, the algorithm is shown in Algorithm 1. We use this implementation as the baseline.

P-Levelset: Based on the Level-Set method [1], we implement it on CPEs of Sunway. The input matrix, as well as the auxiliary data (row index and column index), are loaded into the LDM of CPEs using DMA.

Hybrid-ADC: Based on Mayer's [9] hybrid parallel triangular solve with hybrid anti-diagonal-column format, we adopt this parallel method on Sunway by using MPE and CPEs cooperatively. The MPE processes the anti-diagonal block, while CPEs process the column blocks at the same time. The input matrix and the required subvector on the right-hand side are loaded into LDM using DMA.

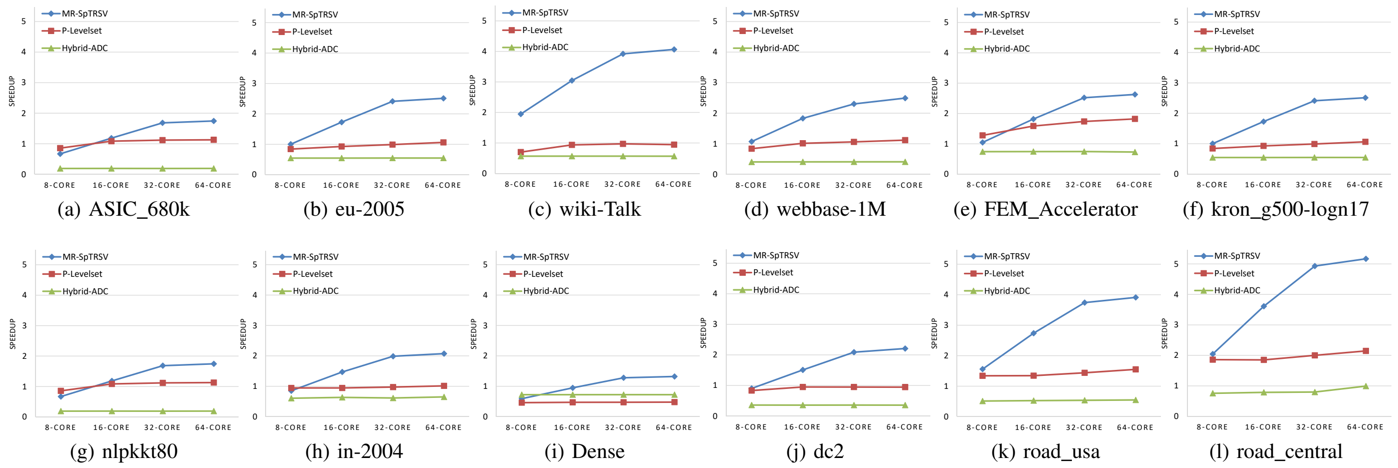
B. Performance Analysis

Figure 5 presents the performance comparison of our approach with two other parallel implementations. The performance results are normalized to the baseline using serial implementation. It's clear that our MR-SpTRSV has better performance across all the datasets than other implementations. Our approach MR-SpTRSV achieves $2.65\times$ speedup on average compared to the baseline. When assigning 40 CPEs as *workers*, 16 CPEs as *routers* and 8 CPEs as *storers*, MR-SpTRSV achieves the best speedup of $5.14\times$. In a few cases, the two other implementations suffer from performance degradation compared to the baseline. This is because without careful design, a large number of irregular memory references are using Gload/Gstore instructions instead of efficient DMA, which leads to performance degradation.

It's encouraging to find that our approach achieves $1.32\times$speedup on dataset `Dense` while other implementations behave worse than the baseline. This is because our approach builds a logical shared memory, the elements stored in it can be reused during the calculation. When the matrix is dense, elements can be reused by *workers* more frequently. At the same time, our approach develops more parallelism by partitioning dense matrix into blocks.

When the parallelism of the matrix is large enough, P-LevelSet can obtain about $2\times$ speedup with its best speedup of $2.14\times$ on matrix `road_central`. Nevertheless, even when the parallelism of matrix `nlpkkt80` is 531200, P-Levelset only obtains $1.29\times$ speedup. This is because parallel level-set prevents any reuse of vector *x* during the calculation. Even when there is enough parallelism, P-Levelset still performs many irregular memory accesses which becomes a performance bottleneck as the number of CPEs increases. Whereas our approach MR-SpTRSV achieves $5.17\times$ speedup with `road_central` and $1.75\times$ speedup with `nlpkkt80`. Compared to P-Levelset, our approach achieves $2.32\times$ speedup on average.

The performance of Hybrid-ADC is the same as the baseline on `roed_central`, and even worse on other datasets. Although this implementation develops parallelism by partitioning the matrix, the anti-diagonal block must be processed sequentially, and the column blocks require frequent thread



(a) ASIC_680k (b) eu-2005 (c) wiki-Talk (d) webbase-1M (e) FEM_Accelerator (f) kron_g500-logn17

(g) nlpkkt80 (h) in-2004 (i) Dense (j) dc2 (k) road_usa (l) road_central

Fig. 5. Performance and scalability comparison between MR-SpTRSV, P-Levelset and Hybrid-ADC on Sunway SW26010 processor with the number of cores equals to 8, 16, 32 and 64, note that for MR-SpTRSV the x-axis is 8, 16, 32, 40 *workers*.

synchronization and many irregular memory accesses. All the above drawbacks limit the performance of Hybrid-ADC running on Sunway. Compared to Hybrid-ADC, our approach utilizes logical shared memory and multi-role register communication to achieve efficient thread synchronization and data reuse.

C. Scalability Analysis

Figure 5 also shows the scalability of each approach. Recall that in our MR-SpTRSV, only 40 CPEs are assigned as *workers* performing the calculation. As the number of cores increases, more *workers* can leverage the logical on-chip memory to improve data reuse. It is clear in Figure 5 that the performance of MR-SpTRSV increases as the number of cores increases, which demonstrates our approach has good scalability across all datasets. For example, with matrix `road_usa`, MR-SpTRSV achieves 1.75× speedup (16 *workers*), 2.3× speedup (32 *workers*) and 2.5× speedup (40 *workers*) compared to the baseline. Both P-LevelSet and Hybrid-ADC exhibit poor scalability due to irregular memory accesses. The scalability of P-Levelset is a little better than Hybrid-ADC on matrices with large parallelism such as `FEM_Acceleraor`, `road_usa` and `road_central`.

D. Parameter Sensitivity Analysis

There are two parameters important for the performance of MR-SpTRSV, Θ (line 4-8 in Algorithm 3) and Ω (line 16-19 in Algorithm 4). Θ determines the block size of one-dimensional cyclic block distribution in *storers*. If it is too small, there will be more DMA transactions for *storers*. If it is too large, there will be fewer blocks in *storers*, which may lead to load imbalance among *storers*. Ω determines the number of elements in vector x prefetched during the computation of a *rect_block*. If it is too small, the prefetch mechanism will be less efficient, and if it is too large, more useless data will be loaded. To evaluate the parameter sensitivity of MR-SpTRSV to the settings of Θ and Ω, we test all 12 datasets by varying

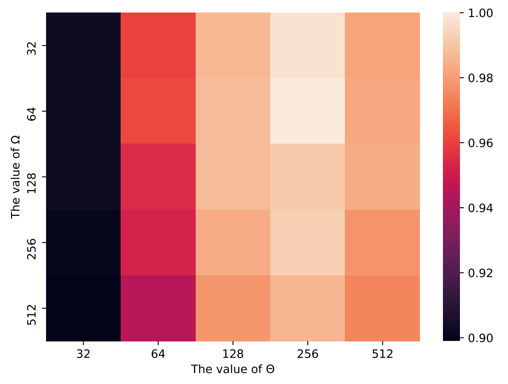


Fig. 6. The performance sensitivity of MR-SpTRSV to parameter Θ and Ω. Each cell in the heatmap is the harmonic mean of performance deviations to the optimal settings across all datasets using 40 *workers*.

the value of Θ (32, 64, 128, 256, 512) and Ω (32, 64, 128, 256, 512). Figure 6 shows the performance heatmap under different parameter settings. It is clear that the performance difference is small (less than 10%) across all parameter settings, which indicates our MR-SpTRSV is less sensitive to Θ and Ω. In our implementation, we set Θ to 256 and Ω to 64 for optimal performance.

V. Related Work

Level-set method [1], [10] divides all the tasks into several levels. Tasks in the same level can be parallelized, but the calculation of levels must be sequential. Wolf analyzed the performance of SpTRSV on CPU and found its bottleneck in [11], the barrier after each level's calculation is most time-consuming. Other papers [12] [13] also focus on barrier synchronization. Park replaced the barrier with P2P synchronization [14] to prune unneeded dependencies; this work was implemented on CPU. And Liu proposed a synchronization-free [15] using atomic operations on GPU, at the same time, this algorithm does not analyze the sparsity structure of the input matrix and thus completely removes costs for generating sets and executing barrier synchronization. Kabir [16] designed a multilevel SpTRSV to improve the data locality on NUMA multicores, found out that how packs of independent tasks may be scheduled for enhanced levels of reuse from the temporal locality of accesses.

Some work pays attention to aspects other than synchronization. Anzt [17] and Chow [18] designed SpTRSV for iterative methods because iterative methods have no demand to figure out the accurate results during one step. However, in some conditions, these methods may change the character of convergence. Venkat's work [19] [20] is interesting. Venkat developed a compiler and runtime framework for parallelizing sparse matrix computations that have loop-carried dependencies. Code optimized by this framework outperforms code tuned by MKL. Some previous papers also focus on data layout [21] [22] [23] [24] of sparse matrix to get better locality.

Our MR-SpTRSV scheme is different from existing approaches, which leverages the unique features of Sunway architecture to build up a logical on-chip shared memory to improve the data locality and enable communication among arbitrary cores to reduce synchronization cost.

VI. Conclusions

In this paper, we present a Multi-Role SpTRSV scheme on Sunway architecture that partitions the sparse matrices into multiple blocks and divides the CPEs into three different roles. The partitioning mechanism enables us to build up a logical shared memory using the scratchpad memory on Sunway, which increases the data reuse during the calculation. The organization of CPEs into multiple roles enables synchronization among arbitrary cores using register communication. The experiments with 12 representative datasets demonstrate our scheme can better utilize the architecture features on Sunway and thus achieve higher performance improvement compared to the-state-of-the-art approaches.

Acknowledgement

This work is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000503) and National Natural Science Foundation of China (Grant No. 61502019). Hailong Yang is the corresponding author.

References

[1] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[2] J. Zhang, C. Zhou, Y. Wang, L. Ju, Q. Du, X. Chi, D. Xu, D. Chen, Y. Liu, and 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*. IEEE Press, 2016, p. 4.

[3] H. Fu, C. He, B. Chen, Z. Yin, Z. Zhang, W. Zhang, T. Zhang, W. Xue, W. Liu, W. Yin *et al.*, "18.9-pflops nonlinear earthquake simulation on sunway taihulight: enabling depiction of 18-hz and 8-meter scenarios," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 2.

[4] Y. Ao, C. Yang, X. Wang, W. Xue, H. Fu, F. Liu, L. Gan, P. Xu, and W. Ma, "26 pflops stencil computations for atmospheric modeling on sunway taihulight," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 535–544.

[5] J. Fang, H. Fu, W. Zhao, B. Chen, W. Zheng, and G. Yang, "swdnn: A library for accelerating deep learning applications on sunway taihulight," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 615–624.

[6] H. Lin, X. Tang, B. Yu, Y. Zhuo, W. Chen, J. Zhai, W. Yin, and W. Zheng, "Scalable graph traversal on sunway taihulight with ten million cores," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 635–645.

[7] W. Dong, L. Kang, Z. Quan, K. Li, K. Li, Z. Hao, and X.-H. Xie, "Implementing molecular dynamics simulation on sunway taihulight system," in *IEEE 18th International Conference on High Performance Computing and Communications*. IEEE, 2016, pp. 443–450.

[8] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[9] J. Mayer, "Parallel algorithms for solving linear systems with sparse triangular matrices," *Computing*, vol. 86, no. 4, p. 291, 2009.

[10] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM journal on scientific and statistical computing*, vol. 11, no. 1, pp. 123–144, 1990.

[11] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors impacting performance of multithreaded sparse triangular solve," in *International Conference on High Performance Computing for Computational Science*. Springer, 2010, pp. 32–44.

[12] M. Bättig and T. R. Gross, "Synchronized-by-default concurrency for shared-memory systems," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2017, pp. 299–312.

[13] E. Gidron, I. Keidar, D. Perelman, and Y. Perez, "Salsa: scalable and low synchronization numa-aware algorithm for producer-consumer pools," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 151–160.

[14] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *International Supercomputing Conference*. Springer, 2014, pp. 124–140.

[15] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," in *European Conference on Parallel Processing*. Springer, 2016, pp. 617–630.

[16] H. Kabir, J. D. Booth, G. Aupy, A. Benoit, Y. Robert, and P. Raghavan, "Sts-k: a multilevel sparse triangular solution scheme for numa multicores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 55.

[17] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *European Conference on Parallel Processing*. Springer, 2015, pp. 650–661.

[18] E. Chow and A. Patel, "Fine-grained parallel incomplete lu factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[19] A. Venkat, M. Hall, and M. Strout, "Loop and data transformations for sparse matrix code," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 521–532.

[20] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall, "Automating wavefront parallelization for sparse matrix computations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 41.

[21] D. Padua, D. Barthou, and A. X. Duchateau, "Hydra: Automatic algorithm exploration from linear algebra equations," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–10.

[22] Z. Liu, I. Calciu, M. Herlihy, and O. Mutlu, "Concurrent data structures for near-memory computing," in *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2017, pp. 235–245.

[23] H. Han and C.-W. Tseng, "Exploiting locality for irregular scientific codes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 606–618, 2006.

[24] Y. Zhang, W. Ding, J. Liu, and M. Kandemir, "Optimizing data layouts for parallel computation on multicores," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 143–154.