Control Formats for Unsymmetric and Symmetric Sparse Matrix–Vector Multiplications on OpenMP Implementations

Takahiro Katagiri[1], Takao Sakurai[2], Mitsuyoshi Igai[3], Satoshi Ohshima[1], Hisayasu Kuroda[4], Ken Naono[2], and Kengo Nakajima[1]

[1] Information Technology Center, The University of Tokyo,
2-11-16 Yayoi, Bunkyo-ku, Tokyo 113-8658, Japan
[2] Central Research Laboratory, Hitachi, Ltd.
292 Yoshida-cho, Totsuka-ku, Yokohama, Kanagawa 244-0817, Japan
[3] Hitachi ULSI Systems Co., Ltd.
292 Yoshida-cho, Totsuka-ku, Yokohama, Kanagawa 244-0817, Japan
[4] Graduate School of Science and Engineering, Ehime University
3 Bunkyo-cho, Matsuyama, Ehime 790-8577, Japan
katagiri@cc.u-tokyo.ac.jp

Abstract. In this paper, we propose "control formats" to obtain better thread performance of sparse matrix–vector multiplication (SpMV) for unsymmetric and symmetric matrices. By using the control formats, we established the following maximum speedups of SpMV in 16-thread execution on one node of the T2K Open Supercomputer: (1) 7.14× for an unsymmetric matrix by using the proposed Branchless Segmented Scan compared to the original Segmented Scan method; (2) 12.7× for a symmetric matrix by using the proposed Zero-element Computation-free method compared to a simple SpMV implementation.

Keywords: Sparse Matrix–Vector Multiplication (SpMV), Control Formats, Zero-element Computation-free, Branchless Segmented Scan.

1 Introduction

Current computer architectures are very complex due to their hierarchical caches and unsymmetric memory accesses. With the increasingly pervasive use of multicore CPUs, highly threaded parallelism is required. To solve this hardware complexity, many numerical libraries with an auto-tuning (AT) facility have been studied and developed [1][2][3][4].

In this paper, we focus on a sparse iterative solver for linear equations. The main part of the solver is sparse matrix–vector multiplication (SpMV). The performance of SpMV depends on the computer architecture, the number of parallel threads, and the locations of non-zero elements in the input sparse matrix. Hence, it is desired to adapt AT technologies.

M. Daydé, O. Marques, and K. Nakajima (Eds.): VECPAR 2012, LNCS 7851, pp. 236–248, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Before actually adapting these AT technologies, it is important to know the variants of SpMV implementation, since the total performance of sparse iterative solvers affects the AT performance with respect to SpMV. In particular, controlling the thread parallelism for CPUs is crucial in current multicore architectures. This is because the number of parallel threads can reach more than 100.

The objective of this paper is to obtain better performance in thread execution. We accomplish this by introducing "control formats" for SpMV. The control format, which is a different concept from the sparse matrix format, controls sequential and parallel (thread) optimizations for symmetric and unsymmetric SpMVs.

The contributions of this paper are summarized as follows. First, we propose a new control format for a symmetric SpMV. Second, we evaluate three kinds of control formats for symmetric and unsymmetric cases on one node of a parallel machine.

This paper is organized as follows. Section 2 explains the control formats and their SpMV implementations. Section 3 is a performance evaluation of control formats on one node of the T2K supercomputer (U. Tokyo). Section 4 shows related work. Finally, we present our conclusions about this research.

2 Control Formats of SpMV

2.1 Definition of Computation

SpMV is defined as

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

where y and x are dense vectors in $\Re^n$, and A is a sparse matrix in $\Re^{n\times n}$. Since A is a sparse matrix, we need to reduce the amount of memory. We can use several formants to represent sparse matrices. For example, Compressed Row Storage (CRS), Coordinate (COO), Ellpack (ELL), and DIA (Diagonal) are widely used. We focus on the CRS format because it is an easy and widely used format in several numerical libraries.

The CRS format uses the following three arrays to represent a space matrix: IRP(1:N+1) for row index pointers of the matrix, ICOL(1:NNZ) for indexes of the columns of the matrix, and VAL(1:NNZ) for the values of non-zero elements, where N is the total number of rows, and NNZ is the total number of non-zero elements. If the matrix is symmetric, lower elements of A are *not* stored in order to reduce the amount of memory. This is a restriction of the design policy of Xabclib.

2.2 Control Formats for SpMV

First we define the control format for SpMV. To represent sparse matrices, we should select a sparse matrix format, such as CRS.

SpMV from the viewpoint of computational components is organized according to the following formula (2).

$$SpMV := Sparse\ Matrix\ Format + Control\ Format. \quad (2)$$

As seen in formula (2), we need an additional format to control computations in SpMV to implement effective computation of SpMV with a sparse matrix format. We call the additional format for computations the control format.

For example, the control format keeps its own row indexes for each thread for the sparse matrix to increase thread parallelism. The control format to manage the computational complexity of SpMV has other potential abilities. Increasing sequential efficiency is also a crucial factor to be considered.

In the next paragraphs, we will show examples of control formats.

Control Format to Establish Load Balancing

With respect to thread parallelization, load imbalance occurs when the total number of rows is not equally divisible by the number of threads. Fig. 1 shows an example of this load imbalance.

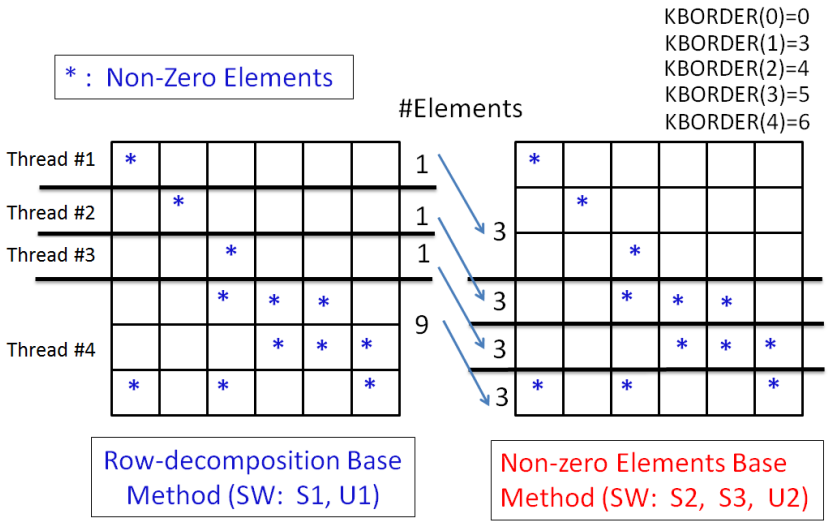


Fig. 1. An example of a load imbalance. The number of threads is 4, the number of sparse rows is 6, and the number of non-zero elements (NNZ) is 12. The U1 and U2 are SpMV implementations for non-symmetric matrices. The S1, S2, and S3 are SpMV implementations for symmetric matrices.

Fig. 1 shows that a heavy load imbalance is caused by the division (Number of rows)/(Number of threads) = floor(6/4) = 1. Thread #4 owns all of the remaining rows, which are three rows in this case. Consequently, a heavy load imbalance for execution time is also caused by OpenMP parallelization. We call this conventional parallelization method the *Row-decomposition Base* method.

To solve the load imbalance problem in the Row-decomposition Base method, we reallocate its load according to the number of non-zero elements per thread. In Fig. 1,

NNZ is 12; hence, perfect load balancing is established with NNZ/(Number of threads) = 12/4 = 3 elements per thread. That is, thread #1 owns three rows while the others own one row in this example. We call this the *Non-zero Element Base* method.

To establish load balancing, we introduce a new control format. Let KBORDER(0:4) be the control format that keeps its own rows for each thread. For example, the number of rows to be owned for thread #1 is represented by KBORDER(1). Determining the number of rows to maintain good load balancing is one of the crucial issues. We implemented a method to find better splitting points by utilizing the divide-and-conquer approach [5].

Branchless Control Format

The Segmented Scan (SS) method [6] is known as an efficient implementation for vector computers. In the control format of SS, a flag array, FLAG(1:NNZ), is used to know the row ends. However, one disadvantage of SS is inefficient execution of the innermost loop, which has an IF-line to prevent branch prediction. The IF-line prevents several optimizations of current cache machines (both compiler and hardware).

To solve this problem, we propose a pair of new control formats, which are JFSTART(1:JL) and MFLAG(1:*). We call computation of SpMV with these new formats the *Branchless Segmented Scan* (BSS) method.[5] Fig. 2 shows the control formats.

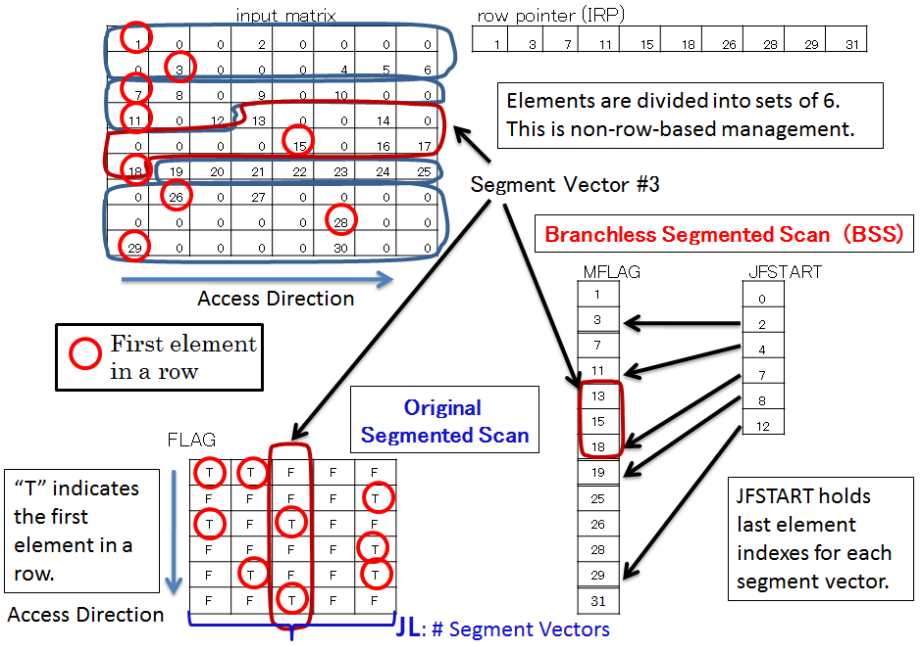


Fig. 2. Control formats of Segmented Scan (SS) and Branchless Segmented Scan (BSS). FLAG(1:NNZ) is the flag to know row ends. JFSTART(1:JL) and MFLAG(1:*) are stride information to know the loop start and loop end.

JFSTART holds the last element indexes for each row of MFLAG. MFLAG holds row-continuous information. This information shows continuous access in the innermost loop, which means the access information of the computation for each segment vector with respect to the CRS format. By using these two control formats, we can establish an "IF-line-free" version of SS. This is why we refer to this method, BSS, as Branchless SS.

In Fig. 2, segment vector #3 starts with the third element in the fourth row. To know the row end, the original SS holds information about the row end in the third row of the array of FLAG. The BSS holds this information in the array of MFLAG. To know the MFLAG information, we refer to the array of JFSTART.

Zero-Element Computation-Free Control Format

If the matrix is symmetric, a workspace is required to maintain parallelism on an OpenMP implementation. One easy way to remove the workspace is to copy elements from the upper part to the lower part. Then, an unsymmetric SpMV is performed. This is against our library design policy.

This procedure raises another issue. The memory space for the working space depends on the number of threads if we use symmetricity for the sparse matrix format. Using symmetricity makes additional computations, such as additions to all workspace areas. The computational costs increase according to the number of threads; hence, this can degrade the parallelization performance. Once again, please note that this additional computation for the working space is not needed when taking an unsymmetric sparse matrix format for symmetric SpMV.

If the matrix is a band matrix, most elements of the workspace must be zero-elements, since there are no off-diagonal non-zero elements. In this situation, we can omit the additions for the off-diagonal part of the workspace. The difficulty is that the input matrix determines whether the omission is performed.

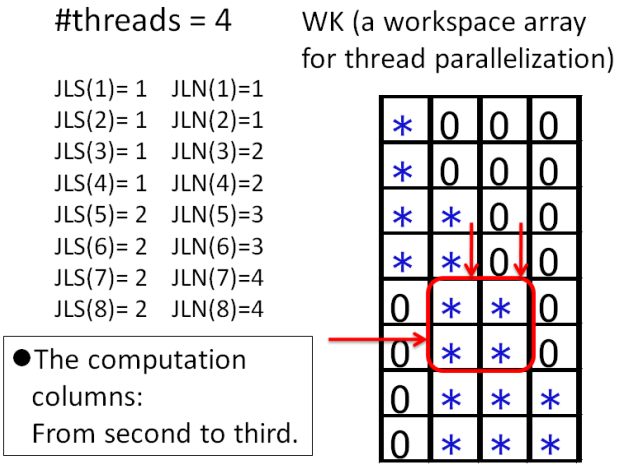


Fig. 3. An example of removing zero-element computations in symmetric SpMV

To address this issue, we introduce a new control format, the *Zero-element Computation-free* method. With respect to the locations of non-zero elements of *A*, we determine the locations of the additions before calling the symmetric SpMV. Fig. 3 gives an example.

As shown in Fig. 3, JLS(1:n) represents the start rows of the index for non-zero elements. JLN(1:n) represents the end rows of the index for non-zero elements.

By using these two control formats, we can compute non-zero elements only for the working space. This is very crucial if the input matrix is a stencil or a band matrix.

2.3 Implementation Details of SpMV on OpenMP

Fig. 4 shows the SpMV implementation of unsymmetric matrices using the control format KBORDER, explained in Section 2.2.

```
!$OMP PARALLEL DO PRIVATE(S, J_PTR, I)
<1>DO K=1,NUM_SMP
<2>   DO I=KBORDER(K-1)+1,KBORDER(K)
<3>     S=0.0D0
<4>     DO J_PTR=IRP(I),IRP(I+1)-1
<5>       S=S+VAL(J_PTR)*X(ICOL(J_PTR))
<6>     END DO
<7>     Y(I)=S
<8>   END DO
<9>END DO
!$OMP END DO PARALLEL
```

Fig. 4. SpMV for unsymmetric matrices with KBORDER

With KBORDER, the computation load from lines <4> to line <6> is almost balanced if it works well. Fig. 5 shows a BSS implementation with JFSTART and MFLAG.

In Fig. 5, by using MFLAG, no IF-line is needed in the innermost loop of the BSS. (See lines <4> to <6>.)

Fig. 6 also shows a simple SpMV implementation for symmetric matrices. Lines <9> to <14> in Fig. 6 cannot be parallelized since a data dependency exists.

Fig. 7 shows the implementation of the SpMV for symmetric matrices with the Non-zero Element Base and Zero-element Computation-free methods.

```
!$OMP PARALLEL DO
            PRIVATE(K,K1,S,I)
<1>DO J=1,JL
<2>   DO K=JFSTART(J),
             JFSTART(J-1)+1,-1
<3>     K1=K+1; S=0.0D0;
<4>     DO I=MFLAG(K1)-1,
           MFLAG(K),-1
<5>       S=VAL(I)*X(ICOL(I))+S
<6>     END DO
<7>     VALSS(K)=S
<8>   END DO
<9>END DO
!$OMP END DO PARALLEL
* Make spanning array
!$OMP PARALLEL DO
<10>DO J=2,JL
<11>   IF(JSFLAG(J).EQV.
            .FALSE.) THEN
<12>     SUM(J-1)=
         VALSS(JFSTART(J-1)+1)
<13>   END IF
<14>END DO
!$OMP END DO PARALLEL
```

```
* Sum up SUM array
<15>DO J=JL-1,1,-1
<16>   IF(PRESENT(J+1) .EQV.
            .FALSE.) THEN
<18>   SUM(J)=SUM(J)+SUM(J+1)
<19>   END IF
<20>END DO
*   Spanning sum
!$OMP PARALLEL DO
<21>DO J=1,JL
<22>   VALSS(JFSTART(J))=
     VALSS(JFSTART(J))+SUM(J)
<23>END DO
!$OMP END PARALLEL
*   Make output
!$OMP PARALLEL DO
       PRIVATE(IN,JN,I)
<24>DO J=1,JL
<25>   IN=JFSTART(J);
<26>   JN=JSY(J);
<27>   DO I=JYN(J),1,-1
<28>     Y(JN)=VALSS(IN)
<29>     IN=IN-1; JN=JN-1;
<30>   END DO; END DO;
!$OMP END DO PARALLEL
```

Fig. 5. SpMV for unsymmetric matrices of BSS with JFSTART and MFLAG

```
!$OMP PARALLEL DO PRIVATE
(S,JJ,JC,I)
<1>DO I=1,N
<2> S=0.0D0
<3> DO JC=IRP(I),IRP(I+1)-1
<4>    JJ=ICOL(JC)
<5>    S=S+VAL(JC)*X(JJ)
<6> ENDDO
<7> Y(I)=S
<8>ENDDO
!$OMP END DO PARALLEL
```

```
<9>DO I=1,N
<10> DO JC=IRP(I)+1,
            IRP(I+1)-1
<11>   JJ=ICOL(JC)
<12>   Y(JJ)=Y(JJ)
            +VAL(JC)*X(I)
<13> ENDDO
<14>ENDDO
```

Fig. 6. Simple SpMV for symmetric matrices

```
!$OMP PARALLEL DO PRIVATE
(S, XDIAG, AA, JJ, I, JC)
<1>DO K=1,NUM_SMP
<2>   DO I=KMBORDER(K-1)+1, N
<3>     WK(I,K)=0.0D0
<4>   ENDDO
<5>   DO I=KMBORDER(K-1)+1,
         KMBORDER(K)
<6>     XDIAG=X(I)
<7>     S=VAL(IRP(I))*XDIAG
<8>     DO JC=IRP(I)+1,
             IRP(I+1)-1
<9>       JJ=ICOL(JC)
<10>       AA=VAL(JC)
<11>       S=S+AA*X(JJ)
<12>       WK(JJ,K)=WK(JJ,K)
             +AA*XDIAG
<13>    ENDDO
<14>    Y(I)=S
<15>  ENDDO
<16>ENDDO
!$OMP END DO PARALLEL

!$OMP PARALLEL DO PRIVATE(S)
<17>DO K=1, NUM_SMP
<18>  DO I=KWBORDER(K-1)+1,
        KWBORDER(K)
<19>    S=0.0D0
<20>    DO J=JLS(I), JLN(I)
<21>      S=S+WK(I,J)
<22>    ENDDO
<23>    Y(I)=Y(I)+S
<24>  ENDDO
<25>END DO
!$OMP END DO PARALLEL
```

Fig. 7. SpMV for symmetric matrices with control formats KMBORDER, JLS, and JLN

In Fig. 7, lines <20> to <22> indicate the kernel of the Zero-element Computation-free method with JLS and JLN.

3 Performance Evaluation

We used the T2K Open Supercomputer (HITACHI HA8000) installed at the Information Technology Center at the University of Tokyo. Each node contains four sockets with AMD Opteron 8356 processors (Quad core, 2.3 GHz). The L1 cache is 64 KB/core, the L2 cache is 512 KB/core, and the L3 cache is 2 MB/4 cores. The memory on each node is 32 GB with 667 MHz DDR2. The theoretical peak is 147.2 GFLOPS/node.

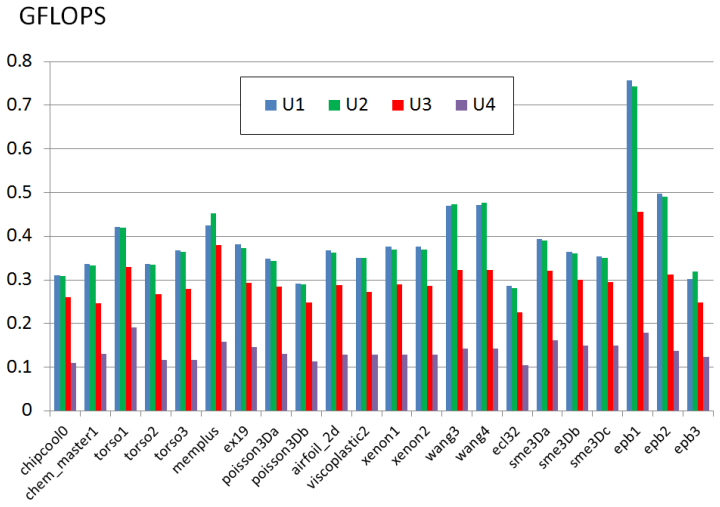
We used Intel Fortran Compiler Professional Version 11.0 with the options "-O3 -m64 -openmp -mcmodel=medium." We used 20 types of symmetric matrices and 22 types of asymmetric matrices from the University of Florida sparse matrix collection (referred to hereinafter as UF collection) [7].

We implemented the SpMV with the proposed control formats in OpenATLib [8]. In OpenATLib, the AT of the SpMV is implemented as the first call of the SpMV routine to survey the performances of the implemented SpMVs. In the current version, three kinds of SpMVs were implemented with an AT switch for symmetric and unsymmetric matrices.

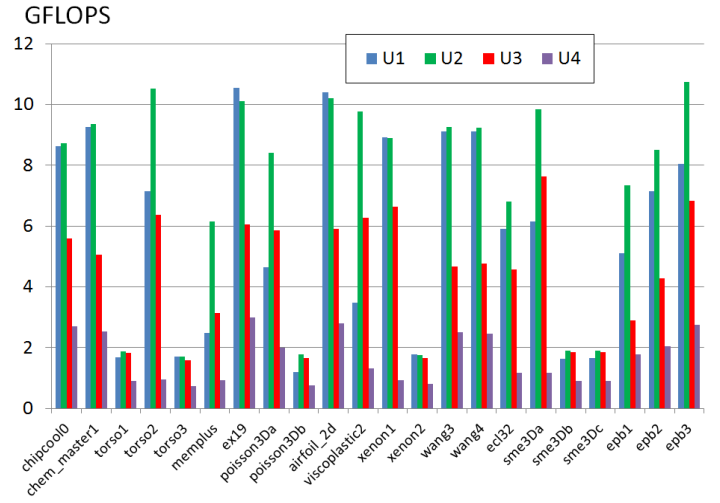
The switch of implementations of the SpMVs is summarized as follows: (1) simple SpMV (U1 for unsymmetric, S1 for symmetric); (2) Non-zero Element Base (U2 for unsymmetric, S2 for symmetric with a simple workspace for each thread); (3) BSS (U3, unsymmetric only); (4) original SS (U4, unsymmetric only); and (5) Zero-element Computation-free with Non-zero Element Base (S3, symmetric only). In this section, we evaluate these implementations of SpMV to know the AT effect of OpenATLib.

3.1 Performance of Unsymmetric SpMV

Fig. 8 shows the performance of the unsymmetric SpMV.



(a) #threads=1



(b) #threads=16

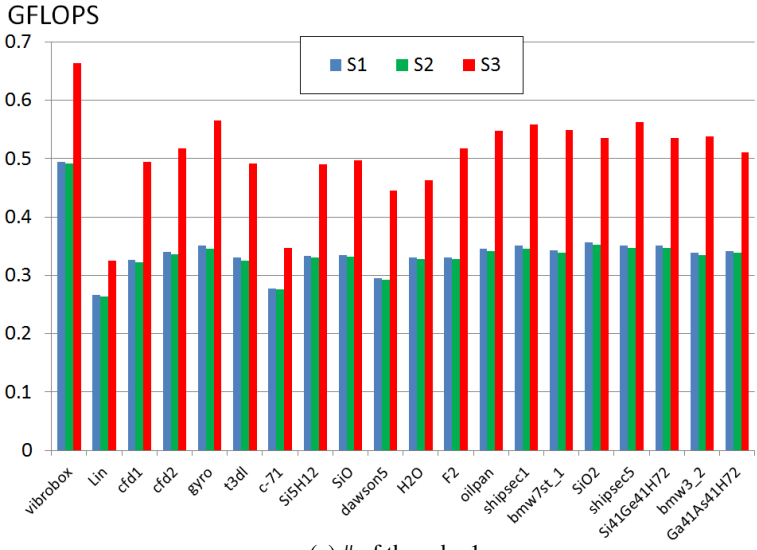
Fig. 8. Performance of unsymmetric SpMV on one node of T2K

According to Fig. 8, we can establish the following speedups compared to U4 (SS): (1) #thread=1: from 1.72× (torso1) to 2.57× (epb1); (2) #threads=16: from 1.63× (epb1) to 7.14× (xenon1) with the BSS control format. The average number of non-zero elements per row (NZEPR) for the UF collection of xenon1 is 24.3. Its NNZ is 1,181,120. One of reasons why we can obtain such a high performance for xenon1 is the decreased cache miss–hit ratio by adapting the branchless control format, since xenon1 is almost a stencil matrix, which is suitable for access optimization for the right-hand-side vector *b*.

Moreover, we can establish the following speedups compared to U1 with the Non-zero Element Base control format: #threads=16: from 0.95× (ex19) to 2.81× (viscoplastic2). The average number of NZEPR for viscoplastic2 is 11.6. The derivation ratio of NZEPR is 13.9. Since the derivation ratio is very close to the average number of NZEPR for this matrix, a load imbalance easily occurs with highly threaded execution; hence, U2 is very crucial in this situation.

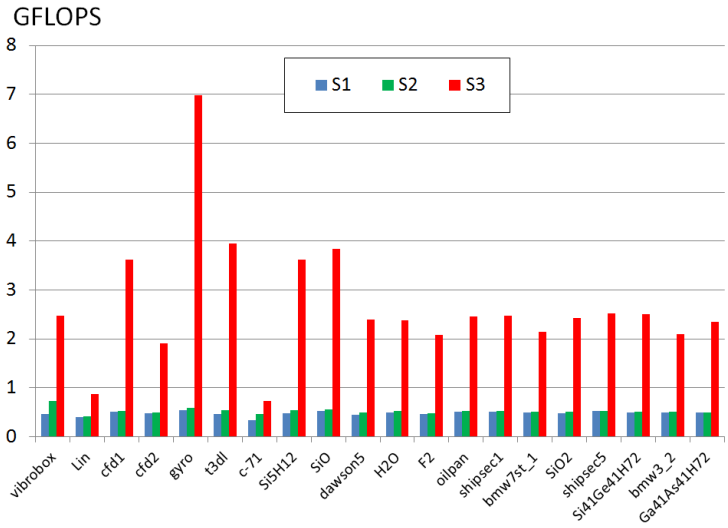
3.2 Performance of Symmetric SpMV

Fig. 9 shows the performance of the symmetric SpMV.



(a) # of threads=1

Fig. 9. Performance of symmetric SpMV on one node of T2K



(b) # of threads=16

Fig. 9. (*continued*)

According to Fig. 9, we can establish the following speedups compared to S1: (1) #thread=1: from 1.22× (Lin) to 1.61× (gyro); (2) #threads=16: from 2.14× (c-71) to 12.7× (gyro) with the control format of the Zero-element Computation-free method.

The UF collection gyro forms almost a block diagonal matrix, but two big blocks appear in its central part. We think that the reduction of the zero-element computation for these two blocks works well. As a result, great performance improvement was established in gyro.

4 Related Work

Although several new sparse matrix formats have been proposed to establish high performance for current CPU architectures, almost no method has been proposed for the control format of SpMV with the CRS format. For example, BELLPACK [9] is a new blocking format for the ELL format. SCSR [10] is a new stream format for the CRS format. These new matrix formats are suitable for current CPU architectures, including graphics processing units (GPUs).

In particular, controlling thread level parallelism (TLP) of SpMV is one of crucial tasks for multicore processors because their parallelism can reach more than 100 threads in the current trend of CPU architectures. Our proposed Non-zero Element Base method to increase TLP is a new control format that supports TLP.

Moreover, the conventional Segmented Scan (SS) method has a drawback for optimization of instruction level parallelism (ILP) in CPUs. To increase ILP, our proposed new control format, BSS, enables us to remove IF-lines from the

computational kernel of the original SS. We think that BSS is the first control format for SpMV with CRS that is aimed at the increase of ILP in CPUs.

Several approaches focusing primarily on TLP to increase parallelism in GPUs are also in demand for SpMVs.

5 Conclusion

In this paper, we proposed new "control formats" to obtain better thread performance in computations of SpMV for unsymmetric and symmetric matrices. Especially, the control format for symmetric matrices to reduce computations of zero-elements is a contribution of this paper.

By using these control formats, we established maximum speedups in 16-thread execution on one node of the T2K Open Supercomputer: (1) 7.14× for an unsymmetric matrix using the BSS method compared to the original SS method; (2) 12.7× for a symmetric matrix using the Zero-element Computation-free method compared to a simple symmetric SpMV implementation. These speedups are crucial for applications with SpMVs.

We implemented these control formats for a sparse iterative solver with an AT facility, named Xabclib [8]. The AT in Xabclib was implemented with a run-time selection function for all implementations of the SpMV in the first call of the library. The effects of AT, hence, depend on the performance of the SpMV.

Evaluating Xabclib with the SpMV by utilizing the proposed control formats in real applications is important future work.

References

1. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated Empirical Optimizations of Software and The ATLAS Project. Parallel Computing 27(1-2), 3–35 (2001)
2. Frigo, M., Johnson, S.G.: FFTW: An Adaptive Software Architecture for the FFT. In: Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing, vol. 3, pp. 1381–1384. IEEE Press, Los Alamitos (1998)
3. Katagiri, T., Kise, K., Honda, H., Yuba, T.: ABCLib_DRSSED: A Parallel Eigensolver with An Auto-tuning Facility. Parallel Computing 32(3), 231–250 (2006)
4. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In: Proceedings of SciDAC, Journal of Physics: Conference Series, vol. 16, pp. 521–530 (2005)
5. Sakurai, T., Naono, K., Katagiri, T., Nakajima, K., Kuroda, H., Igai, M.: Sparse Matrix-Vector Multiplication Algorithm for Auto-Tuning Interface "OpenATLib". IPSJ SIG Notes, vol. 2010-HPC-125(2), pp. 1–8 (2010) (in Japanese)
6. Blelloch, G.E., Heroux, M.A., Zagha, M.: Segmented Operations for Sparse Matrix Computation on Vector Multiprocessors. Carnegie Mellon University, Pittsburgh (1993)
7. The university of Florida sparse matrix collection, `http://www.cise.ufl.edu/research/sparse/matrices/`

8. Xabclib and OpenATLib, http://www.abc-lib.org/Xabclib/index.html
9. Chop, J.W., Singh, A., Vuduc, R.: Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In: Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP) (2010)
10. Guo, D., Gropp, W.: Optimizing Sparse Data Structures for Matrix-vector Multiply. International Journal of High Performance Computing Applications 25(1), 115–131 (2011)