NUMA-Aware Optimization of Sparse Matrix-Vector Multiplication on ARMv8-Based Many-Core Architectures

Xiaosong Yu[1], Huihui Ma[1], Zhengyu Qu[1], Jianbin Fang[2], and Weifeng Liu[1(✉)]

[1] Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum-Beijing, Beijing, China
{2019215847,2019211254,2019215846}@student.cup.edu.cn, weifeng.liu@cup.edu.cn

[2] Institute for Computer Systems, College of Computer, National University of Defense Technology, Changsha, China
j.fang@nudt.edu.cn

Abstract. As a fundamental operation, sparse matrix-vector multiplication (SpMV) plays a key role in solving a number of scientific and engineering problems. This paper presents a NUMA-Aware optimization technique for the SpMV operation on the `Phytium 2000+` ARMv8-based 64-core processor. We first provide a performance evaluation of the NUMA architecture of the `Phytium 2000+` processor, then reorder the input sparse matrix with hypergraph partitioning for better cache locality, and redesign the SpMV algorithm with NUMA tools. The experimental results on `Phytium 2000+` show that our approach utilizes the bandwidth in a much more efficient way, and improves the performance of SpMV by an average speedup of 1.76x on `Phytium 2000+`.

Keywords: Sparse matrix-vector multiplication · NUMA architecture · Hypergraph partitioning · `Phytium 2000+`

1 Introduction

The sparse matrix-vector multiplication (SpMV) operation multiples a sparse matrix A with a dense vector x and gives a resulting dense vector y. It is one of the level 2 sparse basic linear algebra subprograms (BLAS) [13], and is one of the most frequently called kernels in the field of scientific and engineering computations. Its performance normally has a great impact on sparse iterative solvers such as conjugate gradient (CG) method and its variants [17].

To represent the sparse matrix, many storage formats and their SpMV algorithms have been proposed to save memory and execution time. Since SpMV generally implements algorithms with a very low ratio of floating-point calculations to memory accesses, and its accessing patterns can be very irregular, it is a typical memory-bound and latency-bound algorithm. Currently, many

Published by Springer Nature Switzerland AG 2021
X. He et al. (Eds.): NPC 2020, LNCS 12639, pp. 231–242, 2021.
https://doi.org/10.1007/978-3-030-79478-1_20

SpMV optimization efforts have achieved performance improvements to various degrees, but lack consideration on utilizing NUMA (non-uniform memory access) characteristics of a wide range of modern processors, such as ARM CPUs.

To obtain scale-out benefits on modern multi-core and many-core processors, NUMA architectures is often an inevitable choice. Most modern x86 processors (e.g., AMD EPYC series) and ARM processors (e.g., `Phytium 2000+`) utilize NUMA architecture for building a processor with tens of cores. To further increase the number of cores in a single node, multiple (typically two, four or eight) such processor modules are integrated onto a single motherboard and are connected through high-speed buses. But such scalable design often brings stronger NUMA effects, i.e., giving noticeable lower bandwidth and larger latency when cross-NUMA accesses occur.

To improve the SpMV performance on modern processors, we in this work develop a NUMA-Aware SpMV approach. We first reorder the input sparse matrix with hypergraph partitioning tools, then allocate a row block of A and the corresponding part of x for different NUMA nodes, and pin threads onto hardware cores of the NUMA nodes for running parallel SpMV operation. Because the reordering technique can organize the non-zeros in A on diagonal blocks and naturally brings the affinity between the blocks and the vector x, the data locality of accessing x can be significantly improved.

We benchmark 15 sparse matrices from the SuiteSparse Matrix Collection [3] on a 64-core ARMv8-based `Phytium 2000+` processor. We set the number of hypergraph partitions to 2, 4, 8, 16, 32 and 64, and set the number of threads to 8, 16, 32, and 64, then measure the performance of their combinations. The experimental results show that, compared to classical OpenMP SpMV implementation, our NUMA-Aware approach greatly improves the SpMV performance by 1.76x on average (up to 2.88x).

2 Background

2.1 Parallel Sparse Matrix-Vector Multiplication

Sparse matrices can be represented with various storage formats, and SpMV with different storage formats often has noticeable performance differences [1]. The most widely-used format is the compressed sparse row (CSR) containing three arrays for row pointers, column indices and values. The SpMV algorithm using the CSR format can be parallelized by assigning a group of rows to a thread. Algorithm 1 shows the pseudocode of an OpenMP parallel SpMV method with the CSR format.

2.2 NUMA Architecture of the `Phytium 2000+` Processor

Figure 1 gives a high-level view of the `Phytium 2000+` processor. It uses the `Mars II` architecture [16], and features 64 high-performance ARMv8 compatible `xiaomi` cores running at 2.2 GHz. The entire chip offers a peak performance

Algorithm 1. An OpenMP implementation of parallel SpMV.

```
1: #pragma omp parallel for
2: for i = 0 → A.row_nums do
3:     y[i] = 0
4:     for j = A.rowptr[i] → A.rowptr[i + 1] do
5:         y[i] = y[i] + A.val[j] * x[A.col[j]]
6:     end for
7: end for
```

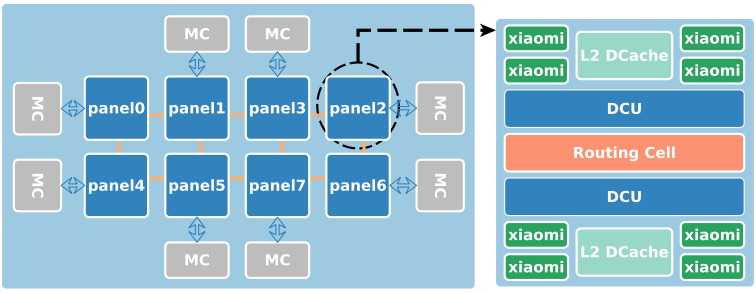


Fig. 1. A high-level view of the `Phytium 2000+` architecture. Processor cores are groups into panels (left) where each panel contains eight ARMv8 based Xiaomi cores (right).

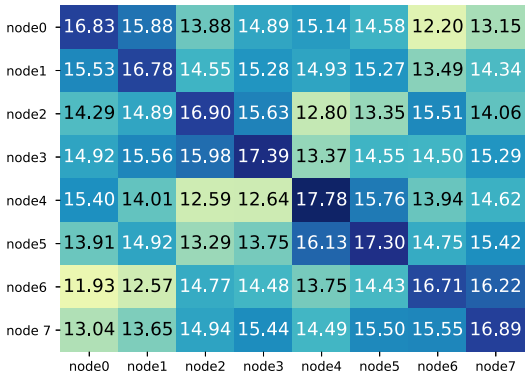


Fig. 2. STREAM triad bandwidth test on `Phytium 2000+`.

of 563.2 Gflops for double-precision operations, with a maximum power consumption of 96 W. The 64 hardware cores are organized into 8 `panels`, where each panel connects a memory control unit.

The panel architecture of `Phytium 2000+` is shown in the right part of Fig. 1. It can be seen that each panel has eight `xiaomi` cores, and each core has a private L1 cache of 32KB for data and instructions, respectively. Every four cores form

a `core group` and share a 2MB L2 cache. The L2 cache of `Phytium 2000+` uses an *inclusive* policy, i.e., the cachelines in L1 are also present in L2.

Each panel contains two Directory Control Units (`DCU`) and one `routing cell`. The `DCUs` on each panel act as dictionary nodes of the entire on-chip network. With these function modules, `Mars II` conducts a hierarchical on-chip network, with a local interconnect on each panel and a global connect for the entire chip. The former couples cores and L2 cache slices as a local cluster, achieving a good data locality. The latter is implemented by a configurable cell-network to connect panels to gain a better scalability. `Phytium 2000+` uses a home-grown `Hawk` cache coherency protocol to implement a distributed directory-based global cache coherency across panels.

We run a customized Linux OS with Linux Kernel v4.4 on the `Phytium 2000+` system. We use gcc v8.2.0 compiler and the OpenMP/POSIX threading model.

We measure NUMA-Aware bandwidth of `Phytium 2000+` by running a Pthread version of the STREAM bandwidth code [15] bound to eight NUMA nodes on the processor. Figure 2 plots the `triad` bandwidth of all pairs of NUMA nodes. It can be seen that the bandwidth results obtained inside the same NUMA node are the highest, and the bandwidth between difference nodes are noticeable much lower. On `Phytium 2000+`, the maximum bandwidth within the same nodes is 17.78 GB/s, while the bandwidth of cross-border access can be down to only 11.93 GB/s. The benchmark results further motivate us to design a NUMA-Aware SpMV approach for better utilizing the NUMA architectures.

2.3 Hypergraph Partitioning

Hypergraph can be seen as a general form of graph. A hypergraph is often denoted by $H = (V, E)$, where V is a vertex set, and E is a hyperedge set. A hyperedge can link more than two vertices, which is different to an edge in a graph [18]. A hypergraph can also correspond to a sparse matrix through the row-net model (columns and rows are vertices and hyperedges, respectively).

The hypergraph partitioning problem divides a hypergraph into a given number of partitions, and each partition includes roughly the same number of vertices. Its effect is that the connections between different partitions can be minimized. Thus when hypergraph partitioning is used for distributed SpMV, both load balancing (because the sizes of the partitions are almost the same) and less remote memory accesses (because connections between partitions are reduced) can be obtained for better performance [4,17,22]. In this work, we use PaToH v3.0 [19,23] as the hypergraph partitioning tool for preparing the sparse matrices for our NUMA-Aware SpMV implementation.

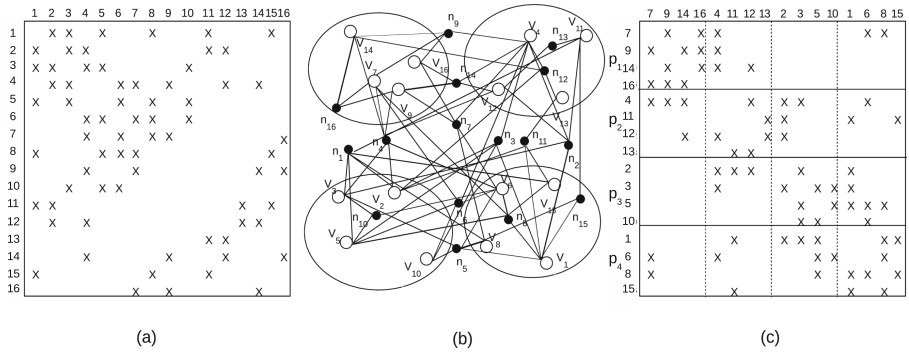


Fig. 3. (a) An original sparse matrix A of size 16×16, (b) A row-net representation of hypergraph H of matrix A, and a four-way partitioning of H, (c) The matrix A reordered according to the hypergraph partitioning. As can be seen, the non-zero entries in the reordered matrix are gathered onto the diagonal blocks, meaning that the number of connections between the four partitions are reduced.

3 NUMA-Aware SpMV Alogrithm

The conventional way of parallelizing SpMV is by assigning distinct rows to different threads. But the irregularity of accessing x through indirect indices of the non-zeros of A may degrade the overall performance a lot. To address the issue, considering the memory accessing characteristics of the NUMA architecture, storing a group of rows of A and part of the vector x most needed by the rows onto the same NUMA node should in general bring a performance improvement. In this way, the cross-node memory accesses (i.e., accessing the elements of x not stored on the local node) can be largely avoided.

To this end, we first need to partition the hypergraph form of a sparse matrix. Figure 3 plots an example showing the difference between a matrix before and after reordering according to hypergraph partitioning. It can be seen that some of the non-zero elements of the matrix move to the diagonal blocks. The number of non-zeros in the off-diagonal blocks in Fig. 3(a) is 48, but in Fig. 3(c), the number is reduced to 38.

After the reordering, the matrix is divided into sub-matrices i.e. row blocks, and the vector is also divided into sub-vectors. For example, the matrix in Fig. 3 now includes four sub-matrices of four rows. Then we use the memory allocation API in libnuma-2.0 [2] to allocate memory for the sub-matrices and sub-vectors on each NUMA node. Figure 4 demonstrates this procedure, and Algorithm 2 lists the pseudocode. As can be seen, the local memory of each NUMA node contains one sub-matrices of four rows and one sub-vector of four elements. In Algorithm 2, the *start* and *end* (line 3) represent the beginning and the end of the row positions for each thread, and the $Xpos$ and *remainder* (line 6) are used to locate the vector x needed for local calculation.

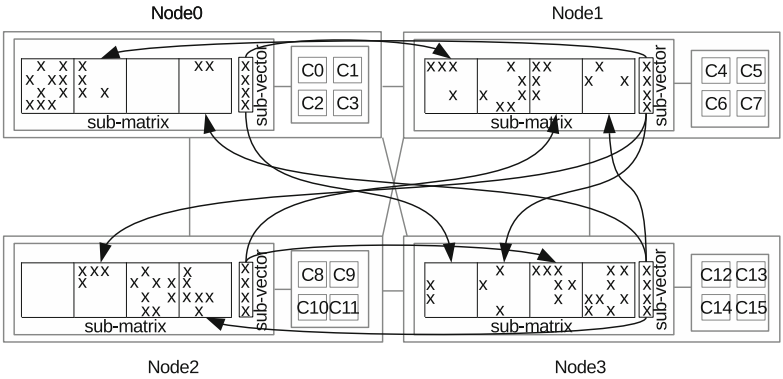


Fig. 4. The computational process of the NUMA-Aware SpMV algorithm on a 16-core four-node NUMA system. The four row blocks of the sparse matrix in Fig. 3(c) are allocated onto the four nodes, and the four sub-vectors are also evenly distributed. When computing SpMV, the cross-node memory accesses for remote x are visualized by using lines with arrows.

When the matrix and vector are allocated, our algorithm creates Pthread threads and bind them to the corresponding NUMA nodes (lines 10–12 in Algorithm 2). For the example in Fig. 4, we issue and bind four threads to each NUMA node and let each one computes a row of the sub-matrix allocated on the same NUMA node (lines 1–9). Since the memory on the node only stores a part of the full vector, the threads will access both the local sub-vector and the remote ones. In the example in Fig. 4, the threads in NUMA node 0 will also access the sub-vectors stored in nodes 1 and 4.

Algorithm 2. A NUMA-Aware Pthread implementation of parallel SpMV.

```
1: function SpMV
2:     numa_run_on_node(numanode)
3:     for i = start → end do
4:         suby[i] = 0
5:         for j = A.subrowptr[i] → A.subrowptr[i + 1] do
6:             suby[i] = suby[i] + A.subval[j] ∗ subx[Xpos][remainder]
7:         end for
8:     end for
9: end function
10: for i = 0 → thread_nums in parallel do
11:     pthread_create(&thread[i], NULL, SpMV, (void∗)parameter)
12: end for
13: for i = 0 → thread_nums in parallel do
14:     pthread_join(thread[i], NULL)
15: end for
```

4 Performance and Analysis

4.1 Experimental Setup and Dataset

In this work, we benchmark 15 sparse matrices from the SuiteSparse Matrix Collection [3] (formerly known as the University of Florida Sparse Matrix Collection). The 15 sparse matrices include seven regular and eight irregular ones. The classification is mainly based on the distribution of non-zero elements. The non-zero elements of regular matrices are mostly located on diagonals, while those of irregular matrices are distributed in a pretty random way. We in Table 1 list the 15 original sparse matrices and their reordered forms generated by the PaToH library. Each matrix is divided into 2, 4, 8, 16, 32 and 64 partitions, and reordered according to the partitions. In terms of the sparsity structures of these matrices, as the number of partitions increases, the non-zero elements will move towards the diagonal blocks.

We measure the performance of OpenMP SpMV and NUMA-Aware SpMV on `Phytium 2000+`. The total number of threads is set to 8, 16, 32 and 64, and the threads are created and pinned to NUMA nodes in an interleaved way. We run each SpMV 100 times, and report the average execution time. For both algorithms, we run original matrices and their reordered forms according to hypergraph partitioning.

4.2 NUMA-Aware SpMV Performance

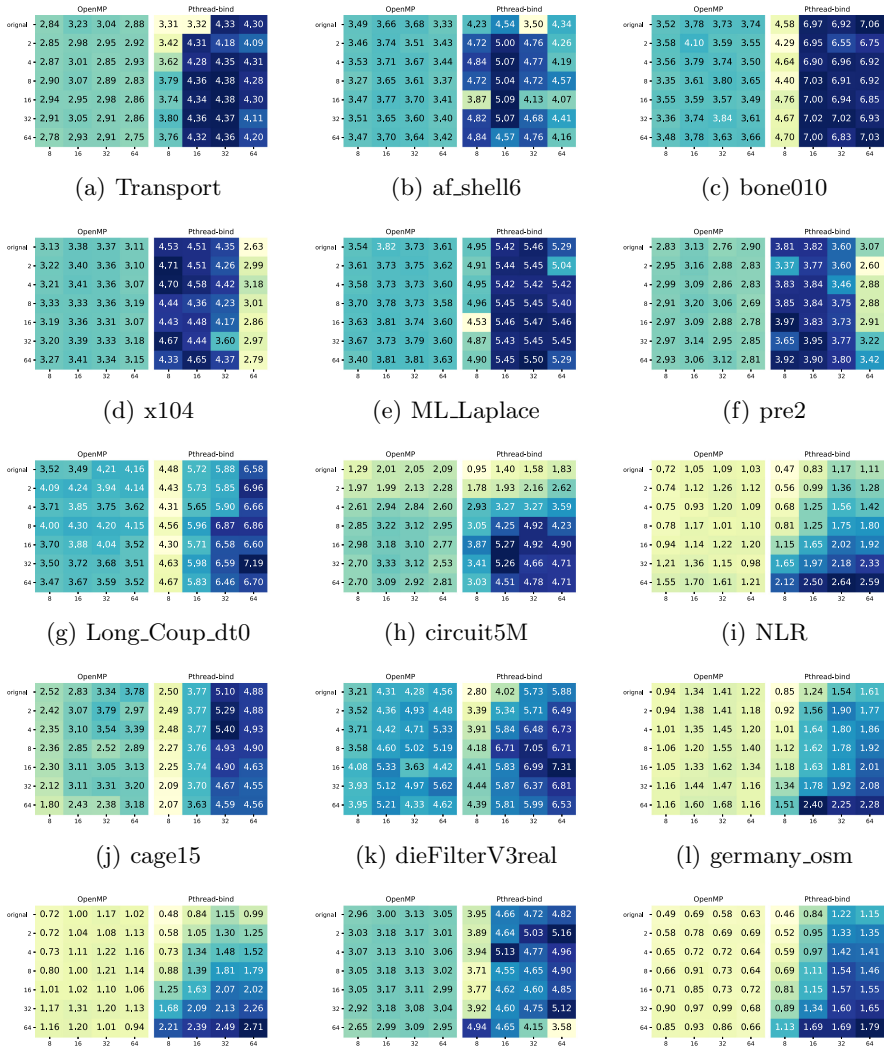
Figure 5 shows the performance comparison of OpenMP SpMV and NUMA-Aware SpMV running the 15 test matrices. Compared to OpenMP SpMV, our NUMA-Aware SpMV obtains an average speedup of 1.76x, and the best speedup is 2.88x (occurs in matrix *M6*). We see that both regular and irregular matrices obtain a significant speedup. The average speedup of irregular matrices is 1.91x, and that of regular matrices is 1.59x.

According to the experimental data, it can be seen that hypergraph partitioning has greatly improved our NUMA-Aware SpMV, but has little impact on OpenMP SpMV. Moreover, for the same matrix, the number of partitions brings noticeable different performance. In can also be seen that, after partitioning, the more non-zero elements the diagonal blocks have, the better the performance.

Specifically, in Fig. 6, with the increase of the number of blocks, the number of remote memory accesses has been continuously decreased. Before partitioning matrix *M6*, SpMV needs a total of 16,242,632 cross-node accesses to the vector x. But when the matrix is divided into 64 partitions, that number drops to 1,100,551, meaning that the number of cross-node accesses decreased by 93.22%. As can be seen from Table 1, the non-zero elements of the split matrix move towards the diagonal. As for the matrix *circuit5M*, the best performance is achieved when having 16 partitions, and the number of cross-node accesses has been dropped by 65%, compared to the original form.

Table 1. Sparsity structures of the original and reordered sparse matrices. The top seven are regular matrices and the bottom eight are irregular ones.

Matrix Name #rows& #non-zeros	Original	#par.=2	#par.=4	#par.=8	#par.=16	#par.=32	#par.=64
Transport 1.6M&23.4M							
*af_shell*6 0.5M&17.5M							
*bone*010 0.9M&47.8M							
*x*104 0.1M&8.7M							
ML_Laplace 0.3M&27.5M							
*pre*2 0.6M&5.8M							
Long_COUP 1.4M&84.4M							
*circuit*5*M* 5.5M&59.5M							
NLR 4.1M&24.9M							
*cage*15 5.1M&99.1M							
*dielFilterV*3 1.1M&89.3M							
germany_osm 11.5M&24.7M							
*M*6 3.5M&21M							
$packing-500$ 2.1M&34.9M							
road_central 14M&33.8M							



(a) Transport (b) af_shell6 (c) bone010

(d) x104 (e) ML_Laplace (f) pre2

(g) Long_Coup_dt0 (h) circuit5M (i) NLR

(j) cage15 (k) dieFilterV3real (l) germany_osm

(m) M6 (n) packing - 500 (o) road_central

Fig. 5. Performance of OpenMP SpMV (left) and NUMA-Aware SpMV (right) on `Phytium 2000+`. In each subfigure, x-axis and y-axis refer to the number of threads and partitions, respectively. The heatmap values are in single precision GFlop/s.

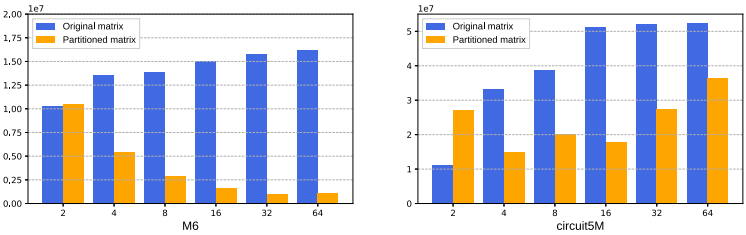


Fig. 6. Comparison of communication volume before and after partitioning under different number of blocks (the x-axis is the number of partitioned blocks, the y-axis is the communication volume).

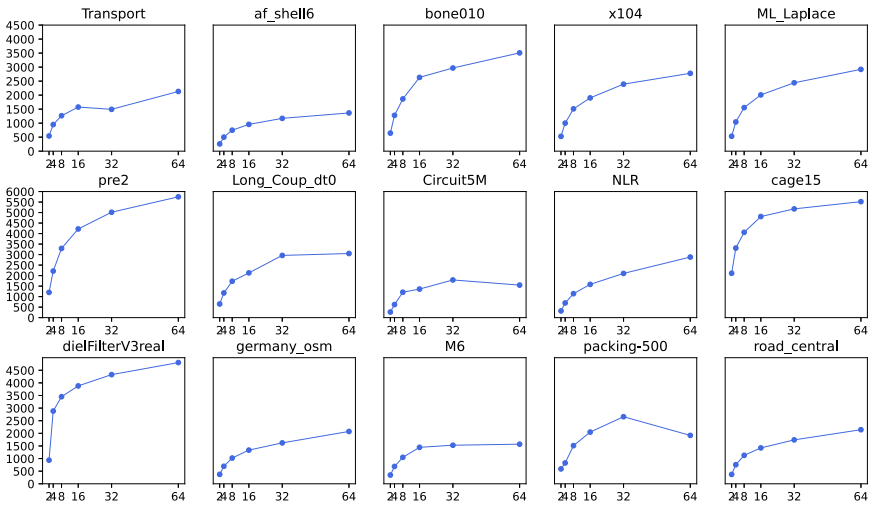


Fig. 7. The ratio of hypergraph partition time to a single NUMA-Aware SpMV time. The x-axis represents the number of blocks from the hypergraph partitioning.

4.3 Preprocessing Overhead (Hypergraph Partitioning Runtime)

The preprocessing overhead (i.e., the execution time of the hypergraph partition) is another important metric for parallel SpMV. Figure 7 reports the ratio of the running time of the hypergraph partition to a single NUMA-Aware SpMV. From the overall perspective of the 15 matrices, due to the different distribution of non-zero elements of the matrix, the best observed performance of NUMA-Aware SpMV is mainly concentrated when the numbers of partitions are 16 and 32. It can be seen that as the number of partitions increases, the ratio increases accordingly. As the number of partition blocks increases, the partition time in general increases as well. Specifically, the matrix *pre2* has a maximum ratio of 5749 times when the number of hypergraph partitions is 64. In contrast, the minimum ratio is 257, which is from the matrix *Af_shell6*.

5 Related Work

SpMV has been widely studied in the area of parallel algorithm. A number of data structures and algorithms, such as BCSR [7], CSX [12] and CSR5 [14], have been proposed for accelerating SpMV on a variety of parallel platforms. Williams et al. [20], Goumas et al. [6], Filippone et al. [5], and Zhang et al. [21] evaluated performance of parallel SpMV on shared memory processors.

Hypergraph partitioning received much attention when accelerating SpMV on distributed platforms. Over the past few decades, researchers have proposed a few partitioning approaches and models for various graph structures and algorithm scenarios [4,17,22,24,25]. A few software packages containing these algorithms have been developed and widely used. For example, Karypis et al. developed MeTiS [9,11], ParMeTiS [10] and hMeTiS [8], and Çatalyürek et al. developed PaToH [19,23], which is used in this work.

6 Conclusions

We have presented a NUMA-Aware SpMV approach and benchmarked 15 representative sparse matrices on a `Phytium 2000+` processor. The experimental results showed that our approach can significantly outperform the classical OpenMP SpMV approach, and the number of generated hypergraph partitions demonstrates a dramatic impact on the SpMV performance.

Acknowledgments. We would like to thank the invaluable comments from all the reviewers. This research was supported by the Science Challenge Project under Grant No. TZZT2016002, the National Natural Science Foundation of China under Grant No. 61972415 and 61972408, and the Science Foundation of China University of Petroleum, Beijing under Grant No. 2462019YJRC004, 2462020XKJS03.

References

1. Asanovic, K., et al.: The landscape of parallel computing research: a view from berkeley. Technical report Uc Berkeley (2006)
2. Bligh, M.J., Dobson, M.: Linux on NUMA systems. In: Ottawa Linux Symposium (2004)
3. Davis, T.A., Hu, Y.: The university of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1–25 (2011)
4. Devine, K.D., Boman, E.G., Heaphy, R.T., Bisseling, R.H., Çatalyürek, Ü.V.: Parallel hypergraph partitioning for scientific computing. In: International Parallel & Distributed Processing Symposium (2006)
5. Filippone, S., Cardellini, V., Barbieri, D., Fanfarillo, A.: Sparse matrix-vector multiplication on GPGPUs. ACM Trans. Math. Softw. **43**(4), 1–49 (2017)
6. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. J. Supercomput. **50**, 36–77 (2009)
7. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: optimization framework for sparse matrix kernels. Int. J. High Perform. Comput. Appl. **18**(1), 135–158 (2004)

8. Karypis, G., Aggarwal, R., Kumar, V., Shekhar, S.: Multilevel hypergraph partitioning: applications in VLSI domain. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **7**(1), 69–79 (1999)
9. Karypis, G., Kumar, V.: Analysis of multilevel graph partitioning. In: Supercomputing 1995: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, pp. 29–29 (1995)
10. Karypis, G., Kumar, V.: Parallel multilevel k-way partitioning scheme for irregular graphs. In: Proceedings of the 1996 ACM/IEEE Conference on Supercomputing, Supercomputing 1996, p. 35-es (1996)
11. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
12. Kourtis, K., Karakasis, V., Goumas, G., Koziris, N.: CSX: an extended compression format for SPMV on shared memory systems. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, pp. 247–256 (2011)
13. Liu, W.: Parallel and scalable sparse basic linear algebra subprograms. Ph.D. thesis, University of Copenhagen (2015)
14. Liu, W., Vinter, B.: CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 339–350 (2015)
15. McCalpin, J.D.: Stream: sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia (1991–2007). A continually updated technical report
16. Phytium: Mars ii - microarchitectures. https://en.wikichip.org/wiki/phytium/microarchitectures/mars_ii
17. Uçar, B., Aykanat, C.: Partitioning sparse matrices for parallel preconditioned iterative methods. SIAM J. Sci. Comput. **29**, 1683–1709 (2007)
18. Uçar, B., Aykanat, C.: Revisiting hypergraph models for sparse matrix partitioning. Siam Rev. **49**(4), 595–603 (2007)
19. Uçar, B., Çatalyürek, V., Aykanat, C.: A matrix partitioning interface to PaToH in MATLAB. Parallel Comput. **36**(5), 254–272 (2010)
20. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Comput. **35**(3), 178–194 (2009)
21. Zhang, F., Liu, W., Feng, N., Zhai, J., Du, X.: Performance evaluation and analysis of sparse matrix and graph kernels on heterogeneous processors. CCF Trans. High Perform. Comput. **1**, 131–143 (2019)
22. Çatalyürek, V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. Parallel Distrib. Syst. **10**(7), 673–693 (1999)
23. Çatalyürek, V., Aykanat, C.: Patoh (partitioning tool for hypergraphs). In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1479–1487. Springer, Heidelberg (2011). https://doi.org/10.1007/978-0-387-09766-4_93
24. Çatalyürek, V., Aykanat, C., Uçar, B.: On two-dimensional sparse matrix partitioning: models, methods, and a recipe. SIAM J. Sci. Comput. **32**(2), 656–683 (2010)
25. Çatalyürek, V., Boman, E.G., Devine, K.D., Bozda, D., Heaphy, R.T., Riesen, L.A.: A repartitioning hypergraph model for dynamic load balancing. J. Parallel Distrib. Comput. **69**, 711–724 (2009)