LETTER

Greedy Approach Based Heuristics for Partitioning Sparse Matrices

Jiasen HUANG[†], **Junyan REN**[†a)], ***and*** **Wei LI**[†], ***Members***

SUMMARY Sparse Matrix-Vector Multiplication (SpMxV) is widely used in many high-performance computing applications, including information retrieval, medical imaging, and economic modeling. To eliminate the overhead of zero padding in SpMxV, prior works have focused on partitioning a sparse matrix into row vectors sets (RVS's) or sub-matrices. However, performance was still degraded due to the sparsity pattern of a sparse matrix. In this letter, we propose a heuristics, called *recursive merging*, which uses a greedy approach to recursively merge those row vectors of nonzeros in a matrix into the RVS's, such that each set included is ensured a local optimal solution. For ten uneven benchmark matrices from the University of Florida Sparse Matrix Collection, our proposed partitioning algorithm is always identified as the method with the highest mean density (over 96%), but with the lowest average relative difference (below 0.07%) over computing powers.

key words: *partitioning, greedy approach, recursive merging, highest mean density, lowest average relative difference*

1. Introduction

Zero padding is the most frequently used strategy for sparse matrix-vector multiplication (SpMxV) kernel to mitigate the memory bandwidth pressure caused by irregular memory access. However, zero padding itself also contributes to the degradation of the performance of the kernel. On one hand, the huge number of trivial operations with zero elements will surely bring about latency [1]–[5]. On the other hand, the sparsity structure of a sparse matrix causes unbalanced load distribution [13]. A lot of partitioning strategies were then proposed to improve load balance for SpMxV [1]–[9], [13].

In previous work [1], each row of the matrix data is zero-padded to force the number of nonzeros (*nnz*) per row to be a multiple of sixteen. Gregg et al. [3] used a variant of CSC, dubbed the sparse matrix architecture and representation (SPAR) format [2], where the *row_ind* and *col_ptr* are combined into a single vector with zero padding introduced at the start of each column of *data* vector. David DuBois and Andrew DuBois [4] made no assumption about the structure of the sparse matrix with the exception that the implementation was designed to process up to 7 elements per row, and if any row contains fewer than 7 elements it must be padded with zeros to the full 7 elements. The Blocked Compressed Sparse Row (BCSR) format, which was proposed by Im and Yelick [5], split a sparse matrix into K RVS's while failing to consider the *nnz* of the rows, thus unbalanced load distribution remained.

Quite recently, Yang [13] improved the algorithms based on *nnz* of row vectors in a matrix [8], [9], and proposed a partitioning algorithm based on *probabilistic model function* (PMF), which essentially was to group those row vectors with the same or similar *nnz* together. However, we discovered that the sparse matrices selected by Yang et al. were either symmetric or symmetric-like, indicating that the algorithm based on PMF depended on the sparsity structure of the matrix as well. Simulation results revealed that the performance of this algorithm greatly degraded when there were few rows with the same or similar *nnz* existing in a matrix.

In this letter, we propose a heuristics to efficiently partition a sparse matrix regardless of the sparsity pattern. In this algorithm, greedy approach is exploited to recursively merge those row vectors of nonzeros in a matrix into the RVS's, such that each vector included is ensured a local optimal solution. For ten benchmark matrices from the University of Florida Sparse Matrix Collection, our proposed algorithm further increases the mean density of partition to 98.55%, 96.27%, and 96.24% with 32, 16, and 8 processors, respectively. Moreover, when 32 processors are used, our proposed algorithm also achieves the lowest average relative difference of 0.068% over computing powers compared with the other strategies.

The rest of this letter is organized as follows: Section 2 reviews related work on sparse matrix partitioning strategies. Section 3 describes the proposed partitioning algorithm. Simulation results are given in Sect. 4.

2. Related Work

Two basic types of strategies for partitioning a sparse matrix are based on one-dimension and two-dimensions respectively.

The two-dimensional strategy that aims to partition a sparse matrix into submatrices can only adapt to the sparse matrices with strong diagonal feature, and the performance will deteriorate even further if *nnz* outside the diagonal counts much. An attempt to improve such strategy by combining different sparse matrix storage formats, such as the CSR format [10] and the DIA format [11], [12] has also been proved ineffective [13].

Depending on whether the row vectors are kept in their

Manuscript received April 14, 2015.
Manuscript revised June 23, 2015.
Manuscript publicized July 2, 2015.

[†]The authors are with the State-Key Laboratory of ASIC and Systems, Fudan University, Shanghai, 200433 China.
a) E-mail: jyren@fudan.edu.cn

DOI: 10.1587/transinf.2015EDL8088

original order in the matrix, the one-dimensional strategies can be further divided into two types. The first type, including the strategies respectively based on the number of rows [5], [6] and on the *nnz* of rows [8], [9], do not split a row into different blocks. Whilst the second type, represented by the algorithm based on PMF [13], aims to further reduce the overhead introduced in the design by grouping different rows together.

2.1 Algorithm 1: Based on Number of Rows

This algorithm splits a sparse matrix $\mathbf{A}$ into K RVS's $A := \{A_i | i = 1, 2, \ldots, K\}$ while ensuring the number of rows in A_i being linearly proportional to the computing power CP_i of a processor. Without considering the *nnz* of the rows, this algorithm suffers from unbalanced load distribution due to irregularity of the distribution of nonzeros in the matrix [5], [6].

2.2 Algorithm 2: Based on the *nnz*

To mitigate the performance degradation caused by the sparsity structure of a sparse matrix $\mathbf{A}$, the matrix is then partitioned into several blocks according to the *nnz* of the rows. Compared with the CSR or the COO format, the ELL format [7] as well as its variations, such as the BELLPACK [8] and the SELLPACK [9] are much better suited for implementing this algorithm on GPU or muti-core CPUs that based on vector processors. However, zeros introduced by this algorithm usually have large deviation due to the fixed order of the row vectors in $\mathbf{A}$, and in some case, such as Schenk_ISEI/ohne2 [14], this algorithm performs even worse than Algorithm 1.

2.3 Algorithm 3: Based on PMF

Quite recently, Yang et al. [13] proposed an algorithm based on *probability mass function* (PMF). In this algorithm, those row vectors with the same or similar *nnz* in a sparse matrix $\mathbf{A}$ are grouped together, suggesting the row vectors are no longer considered in their original order. Hence large deviation involved in Algorithm 2 is relieved, and some increase in the mean density of the partition is thus expected.

Nevertheless, the request for row vectors with the same or similar *nnz* is not always met. In fact, among those ten so-called uneven matrices for comparison [14], four matrices, including nlpkkt120, nlpkkt160, nlpkkt200 and pwtk, are symmetric, whilst the left six matrices, TSOPF_RS_b300_c3, PR02R, pwtk, rajat31, ohne2 and cage15, are all symmetric-like. For those matrices where few rows with the same or similar *nnz* can be found, the request itself turns to the constraint instead.

3. Proposed Partitioning Algorithm

Let NNZ be the total *nnz* in a sparse matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$, k be the number of the processors pre-allocated, and $\overline{NNZ} = \lceil NNZ/k \rceil$ be the mean *nnz* thereby decided. Define $A := \{A_i | i = 1, 2, \ldots, N\}$ as the partition composed of the nonzeros of all the row vectors in $\mathbf{A}$.

The proposed heuristics considers the items $A_1, \ldots, A_N$ sorted by the non-increasing order of $nnz(A_i)$ $(1 \le i \le N)$. The idea is to continuously merge the items into the row vectors following a *best-fit* policy until all those vectors are full or maximally filled. Precisely, this policy falls into two cases: 1) The *Gap* between $nnz(V_1)$ and $\overline{NNZ}$ can be exactly filled in with V_α $(2 \le \alpha \le k, 1 \le k \le< N)$. Such a row α is considered a best-fit row (14^{th} line); 2) The *Gap* can not be filled in within one step, but there exist some rows can be exploited to coalesce with V_1 to shorten the *Gap*. Among these rows, a row β with bigger $nnz(V_\beta)$ is preferred as a best-fit row as well (19^{th} line). In the first case, $V_1 \bigcup V_\alpha$ is returned as an item in A_{final}; both V_1 and V_α are removed from V thereafter. Whilst in the second case, only V_β is to be removed right after coalescing while V_1 will always wait for updating until there are no more sets in V can be further exploited for merging. Hence, such a greedy approach always ensures a set achieved at the end of each iteration a local optimal solution. The numerical description of the proposed heuristics is shown as follow:

Greedy Approach Based Heuristics

1. Sort A:=$\{A_1, ..., A_N\}$ by the non-increasing order of $nnz(A_i)$ $(1 \le i \le N)$, and then A:=$\{A_{I(1)}, ..., A_{I(N)}\}$ *s.t.* $nnz(A_{I(1)}) \ge ... \ge nnz(A_{I(N)})$.
2. $i \leftarrow 0$; ▶ Initialize the iterator
3. $A_{final} \leftarrow \Theta$; ▶ Initialize the final RVS's
4. $V_{in} \leftarrow A$; ▶ Initialize the input RVS's as the sorted A:=$\{A_1, ..., A_N\}$

5. **while** $i < N$
6. $[V_{out}, V_{in}, count] = merging(V_{in}, \overline{NNZ})$;
7. $A_{final} \leftarrow A_{final} \cup V_{out}$;
8. $i \leftarrow i + count$;
9. **end while**

10. **function** $[V_{out}, V_{new}, count] = merging(V, \overline{NNZ})$
11. Set $nnz(V)$:=$\{nnz(V_1), ..., nnz(V_k)\}$ $(1 \le k \le N)$;
12. $Gap \leftarrow \overline{NNZ} - nnz(V_1)$.
13. Traverse $V_2, ..., V_k$.
14. **if** best-fit row α is found ($\bullet$ $nnz(V_\alpha) = Gap$)
15. $V_{out} \leftarrow V_1 \cup V_\alpha$;
16. $V_{new} \leftarrow V - V_\alpha - V_1$;
17. $count \leftarrow 2$;
18. **else**
19. **if** best-fit row β found
($\bullet$ $nnz(V_\beta) < Gap$ $\bullet$ bigger $nnz(V_\beta)$ is preferred)
20. $V_1 \leftarrow V_1 \cup V_\beta$;
21. $V_{new} \leftarrow V - V_\beta$;
22. $V_{out} \leftarrow \Theta$;
23. $count \leftarrow 1$;
24. **else**
25. $V_{out} \leftarrow \{V_{in}(1)\}$;
26. $V_{new} \leftarrow V_{in} - \{V_{in}(1)\}$;
27. $count \leftarrow 1$;
28. **end if**
29. **end if**
30. **end function**

Table 1 Performance comparison for a small scale case.

Algorithm	RVS's	$nnz(B_1)$	$nnz(B_2)$	$nnz(B_3)$	$E(B_1)$/%	$E(B_2)$/%	$E(B_3)$/%	$E(\mathbf{A})$/%	$D(\mathbf{A})$/%
1	$\{A_1, A_2, A_3, A_4, A_5, A_6, A_7\}$, $\{A_8, A_9, A_{10}, A_{11}, A_{12}, A_{13}, A_{14}\}$, $\{A_{15}, A_{16}, A_{17}, A_{18}, A_{19}, A_{20}\}$	25	53	39	35.90	35.90	0	23.93	53.67
2	$\{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9\}$, $\{A_{10}, A_{11}, A_{12}, A_{13}, A_{14}, A_{15}\}$, $\{A_{16}, A_{17}, A_{18}, A_{19}, A_{20}\}$	44	41	32	12.82	5.13	17.95	11.97	48.35
3	$\{A_5, A_{16}, A_1, A_7, A_{19}, A_3, A_{13}, A_2, A_6, A_8, A_{10}, A_{17}\}$, $\{A_4, A_{11}, A_{15}, A_{12}\}$, $\{A_{14}, A_{20}, A_9, A_{18}\}$	41	30	46	5.13	23.08	17.95	15.38	73.12
Proposed	$\{A_{18}, A_{14}, A_9, A_{13}\}$, $\{A_{12}, A_{15}, A_4, A_{20}, A_8\}$, $\{A_{17}, A_6, A_{10}, A_2, A_3, A_7, A_{19}, A_1, A_5, A_{11}, A_{16}\}$	39	39	39	0	0	0	0	100

4. Performance Evaluation

4.1 Definition

Assume k processors are allocated in the design, and then the RVS's got after partitioning is defined as $B := \{B_1, B_2, \ldots, B_k\}$. Let NNZ be the total nnz of the sparse matrix $\mathbf{A}$, $W(B_i)$ be the maximum nnz in B_i, and $R(B_i)$ be the number of the row vectors classified into B_i. The mean density of this partition of $\mathbf{A}$ is then calculated as

$$D(\mathbf{A}) = NNZ/\sum_{i=1}^{k}(W(B_i) \times R(B_i)). \quad (1)$$

Let $nnz\,(B_i)$ denote the nnz of B_i, and then the relative difference of this partition over the computing power CP_i is given by

$$E(B_i) = |nnz(B_i) - CP_i \times NNZ|\,/(CP_i \times NNZ), \quad (2)$$

Without loss of generality, assume all the processors are with the same computing power, i.e., $CP_1 = CP_2 = \ldots = CP_k$, and then (2) is revised as

$$E(B_i) = |nnz(B_i) - NNZ/k|\,/\,(NNZ/k)\,, \quad (3)$$

Accordingly, the average relative difference of this partition over each computing power CP_i is evaluated by

$$E(\mathbf{A}) = \left(\sum_{i=1}^{k}|nnz(B_i) - NNZ/k|\right)/NNZ. \quad (4)$$

4.2 Preliminary Exploration

For ease of illustration, take the 20×20 sparse matrix $\mathbf{A}$ in [13] for example at first. The nnz of the row vectors in $\mathbf{A}$ can be referred to in Fig. 1. As mentioned above, three processors *s.t.* $CP_1 = CP_2 = CP_3$ are allocated in this case. As shown in Table 1, each set of the RVS's got with our proposed algorithm is with a length being exactly equal to the $\overline{NNZ} = \lceil 117/3 \rceil = 39$. Hence, no zero is to be filled, thus achieving a one hundred percent $D(\mathbf{A})$, which increases by 46.33%, 51.65% and 26.88% than that of Algorithm 1, 2 and 3 respectively. Similarly, $E(\mathbf{A})$ decreases to zero, indicating a perfect match between the processors and the assigned tasks.

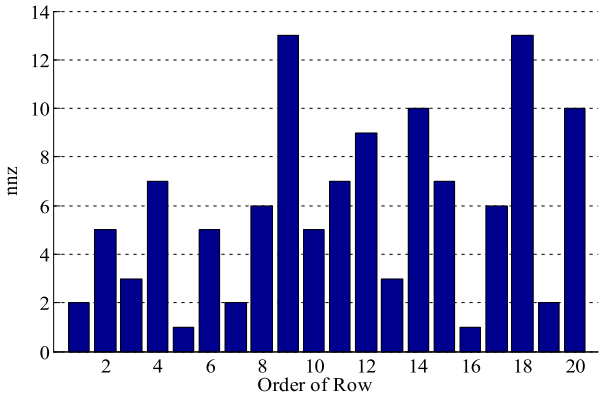


Fig. 1 The nnz of the row vectors in $\mathbf{A}$.

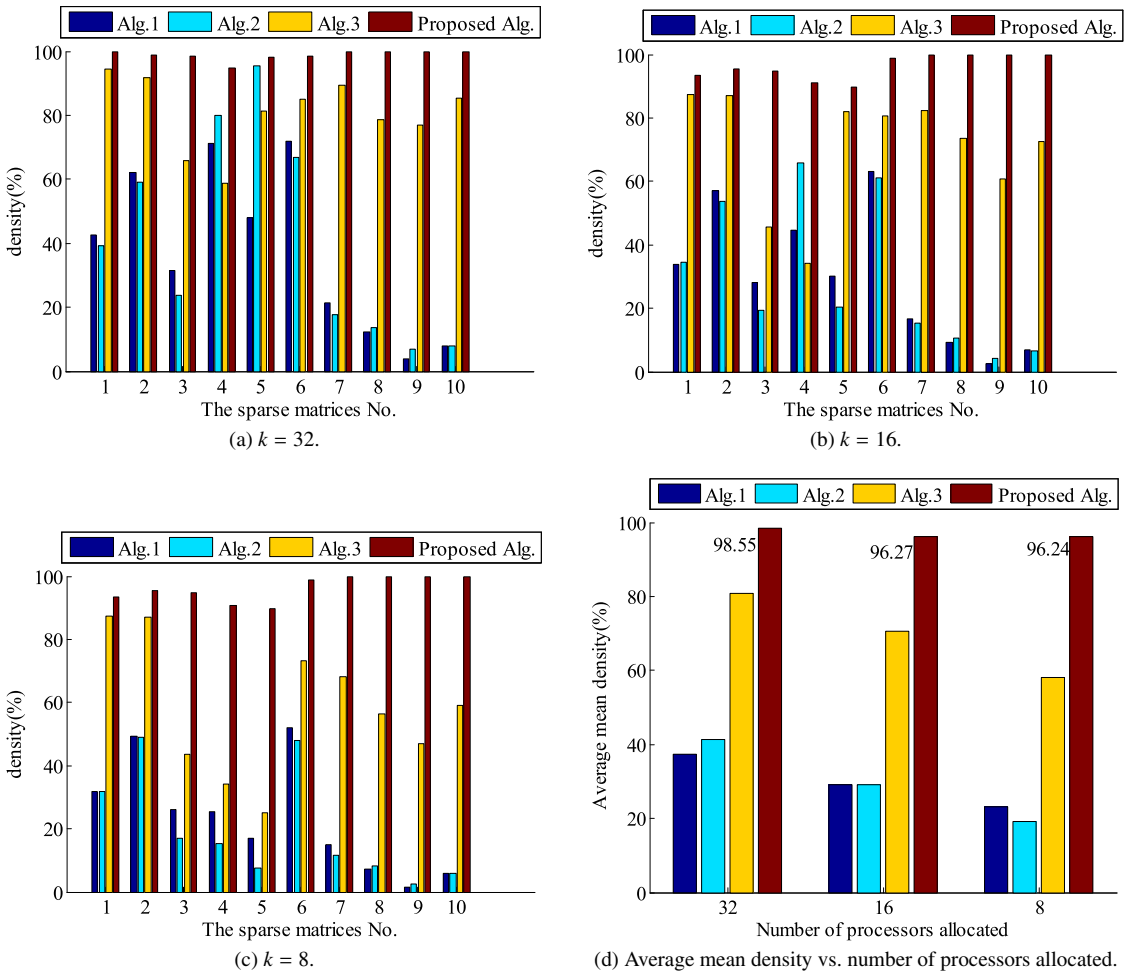
Table 2 Benchmark matrices used in experiments.

No.	Sparse Matrix	Dimension	Density(%)	Feature
1	HB/str_600	363	2.48845	relatively dense
2	HB/jpwh_991	991	0.61370	symmetric-like
3	HB/bcsstm13	2,003	0.52794	symmetric-like
4	Bai/tols1090	1,090	0.29846	irregular
5	Bai/tols4000	4,000	0.05490	irregular
6	Pajek/GD00_a	352	0.36964	irregular
7	Pajek/Erdos972	5,488	0.04705	symmetric-like
8	Pajek/SmaGri	1,059	0.43862	triangular-like
9	Pajek/Kohonen	4,470	0.06372	triangular-like
10	Pajek/Zewail	6,752	0.11896	triangular-like

4.3 Benchmark Studies

Table 2 characterizes ten sparse matrices from the University of Florida Sparse Matrix Collection [14]. Performance degradation of Algorithm 3 is expected for those irregular or triangular-like featured matrices, where few rows with the same or similar nnz exist. In addition, in order to see if the performance of different algorithms will be affected by number of the processors allocated, 32, 16, and 8 processors are respectively assumed in the experiments. As shown in Fig. 2, the performance of our proposed algorithm suffers little from the number of the processors allocated, while all the other three algorithms have exactly the reverse effect.

Precisely, see Fig. 2 (d), with our proposed algorithm, the average $D(\mathbf{A})$ of 98.55%, 96.27%, and 96.24% are respectively achieved in the cases of $k = 32$, $k = 16$, and $k = 8$, while for each of the other three algorithms, the average $D(\mathbf{A})$ drops rapidly with the decrease of k.



(a) $k = 32$. (b) $k = 16$. (c) $k = 8$. (d) Average mean density vs. number of processors allocated.

Fig. 2 Performance comparison of different partitioning algorithms. The parameter k is the number of the processors allocated. Figure 2 (a∼c) illustrates the mean density respectively achieved by the proposed algorithm and by the other three algorithms discussed above. Figure 2 (d) calculates the average mean density of partitioning for all those ten benchmark matrices when 32, 16, and 8 processors are allocated.

5. Conclusions

In this letter, we propose a greedy approach based heuristics, called *recursive merging*, which is to partition a sparse matrix into RVS's by recursively merging the row vectors of nonzeros in the matrix. Ensured by the *best-fit* policy, each set included in the final RVS's is a local optimal solution. For ten irregular benchmark matrices from the University of Florida Sparse Matrix Collection, our proposed algorithm achieves so far the highest mean density (> 96%), but the lowest average relative difference (< 0.07%) over computing powers.

References

[1] Y. Zhang, Y.H. Shalabi, R. Jain, K.K. Nagar, and J.D. Bakos, "FPGA vs. GPU for Sparse Matrix Vector Multiply," Int. Conf. Field-Programmable Tech. (FPT 2009), pp.255–262, Dec. 2009.

[2] V.E. Taylor, A. Ranade, and D.G. Messerschmitt, "SPAR: a new architecture for large finite element computations," IEEE Trans. Computers, vol.44, no.4, pp.531–545, April 1995.

[3] D. Gregg, C. McSweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory," Proc. Int. Conf. Field Programmable Logic Applicat. (FPL 2007), pp.786–791, Aug. 2007.

[4] D. DuBois, A. DuBois, C. Connor, and S. Poole, "Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer," Proc. Int. Symp. Field-programmable gate array (FPGA '08), pp.239–247, 2008.

[5] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," Int. Conf. Comput. Sci. (ISSC '01), vol.2073, pp.127–136, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[6] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: a gpu implementation of a general sparse linear solver," Int. J. Parallel Emerg. Distrib. Syst., vol.24, no.3, pp.205–223, 2009.

[7] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," Conference on High Performance Computing Networking, Storage and Analysis, 2009.

[8] M. Belgin, G. Back, and C.J. Ribbens, "Pattern based sparse matrix representation for memory-efficient SMVM kernels," Proc. ACM Int. Conf. Supercomputing, pp.100–109, 2009.

[9] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix vector multiplication for GPU architectures," High Performance Embedded Architectures and Compilers, vol.5952, pp.111–125, Berlin, Heidelberg, 2010.

[10] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, et al., Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Society for Industrial and Applied Mathematics, 2000.

[11] E.-J. Im, "Optimizing the performance of sparse matrix-vector multiplication," PhD thesis, UC Berkeley, 2000.

[12] R.W. Vuduc, "Automatic Performance Tuning of Sparse Matrix Kernels," Ph.D. thesis, UC Berkeley, 2003.

[13] W. Yang, K. Li, Z. Mo, and K. Li, "Performance Optimization Using Partitioned SpMV on GPUs and Multicore CPUs," IEEE Trans. Comput., p.1, 2014.

[14] T.A. Davis and Y. Hu, "The university of Florida sparse matrix collection," ACM Trans. Math. Softw., vol.38, no.1, pp.1–25, 2011. Available: www.cise.ufl.edu/research/sparse/matrices