Future Generation Computer Systems 54 (2016) 490–500

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Efficient sparse matrix–vector multiplication using cache oblivious extension quadtree storage format

Jilin Zhang [a,b,c,*], Jian Wan [a,b,c], Fangfang Li [a,b,c], Jie Mao [d], Li Zhuang [a,b,c], Junfeng Yuan [a,b,c], Enyi Liu [a,b,c], Zhuoer Yu [a,b,c]

[a] *School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, 310018, China*
[b] *Key Laboratory of Complex Systems Modeling and Simulation, Ministry of Education, China*
[c] *Zhejiang Provincial Engineering Center on Media Data Cloud Processing and Analysis, Hangzhou, 310018, China*
[d] *School of Mechanical Engineering, Hangzhou Dianzi University, Hangzhou, 310018, China*

HIGHLIGHTS

- A cache oblivious extension quadtree storage structure (*COEQT*) is proposed.
- Present the converting algorithm from *CSR* storage format to the *COEQT* storage format.
- Implement the sparse matrix–vector multiplication algorithm based on the *COEQT*.
- Optimize the performance of the implemented algorithm through manual vectorization.
- Implement the parallel sparse matrix–vector multiplication in distributed system.

ARTICLE INFO

Article history:
Received 6 August 2014
Received in revised form
12 February 2015
Accepted 7 March 2015
Available online 29 June 2015

Keywords:
Sparse matrix–vector multiplication
Sparse matrix storage
Data locality
Cache oblivious
Extension quadtree
Distributed parallelism

ABSTRACT

In this paper, we elaborate on improving the sparse matrix storage format to optimize the data locality of sparse matrix–vector multiplication (*SpMVM*) algorithm, and its parallel performance. First of all, we propose a cache oblivious extension quadtree storage structure (*COEQT*), in which the sparse matrix is recursively divided into sub-regions that can well fit into cache to improve the data locality. Later on, we present a *COEQT* based *SpMVM* algorithm and optimize its performance through manual vectorization. With this storage format, the original *SpMVM* is divided into computations of relatively independent small matrices. In addition, this region-based computation framework is also suitable for high performance computing in distributed computing environment. So, we finally present a parallel *SpMVM* algorithm based on the proposed *COEQT*. Extensive and comprehensive experiments show that the sparse matrix–vector multiplication using the *COEQT* storage format achieves on average 1.1–1.5× speedup compared with *CSR* format and further higher performance through instruction level optimization techniques. The experiment in Lenovo Deepcomp 7000 demonstrates that this method achieves on average 1.63× speedup compared with the Intel Cluster Math Kernel Library implementation.

1. Introduction

Large-scale sparse matrix plays an important role in the discrete physical process of many scientific problems, such as representing the interaction between elements in the finite element analysis, describing the graph and transforming the map by sparse matrix operations, solving partial differential equations in fluid dynamics, etc. Therefore, large-scale sparse matrix–vector multiplication (*SpMVM*) is thought as one of the most important scientific and engineering computing methods in the next decade [1]. However, affected by some factors [2] (e.g., the storage format, sparse matrix pattern), *SpMVM* algorithm is generally inefficient. Thus, the premise of researching *SpMVM* algorithm is the research of sparse matrix storage format. The existing common storage formats of sparse matrix are as follows.

XY coordinate storage format (COO). In this format, the row and column coordinates of a nonzero element are stored when its value

* Corresponding author at: School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou, 310018, China. Tel.: +86 15258801227.
E-mail address: jilin.zhang@hdu.edu.cn (J. Zhang).

http://dx.doi.org/10.1016/j.future.2015.03.005
0167-739X/© 2015 Elsevier B.V. All rights reserved.

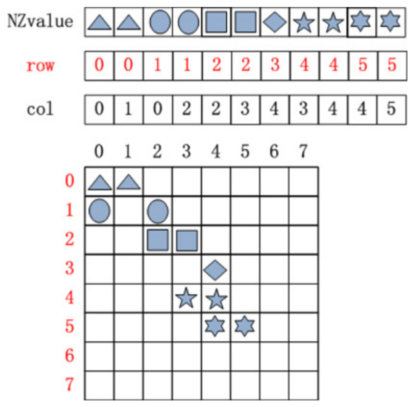

Fig. 1. The *COO* storage format for sparse matrix.

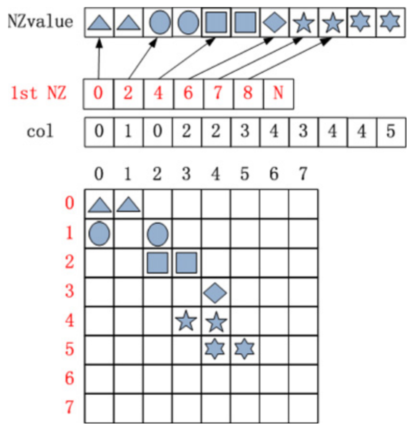

Fig. 2. The *CSR* storage format for the same sparse matrix in Fig. 1.

is stored. Therefore, it needs three arrays: NZvalue array for the values of nonzero elements, row array and col array for the row and column coordinates of nonzero elements, as shown in Fig. 1. Consider the sparse-vector multiplication $y = A_{nn}x$, in which A is a sparse matrix of order n with N nonzero elements and x is a dense vector, then the length of each array is N in the *COO* storage format. We assume that each element in A occupied S_D storage space and each element in row array and col array occupied S_I storage space, then the space overhead of *COO* is $N(S_D + 2S_I)$. In addition, the result vector y are calculated through indirectly addressing, as the following code: $(y[row[i]] += NZvalue[i] * x[col[i]](i \in [1, N]))$, during the multiplication. Hence, as for the large-scale sparse matrix–vector multiplication, indirect addressing will cause high cache miss rate and poor execution performance. Especially when the matrix is highly sparse, irregular data accessing will lead to significant reduction of the execution performance.

Compressed sparse row storage format (CSR). In order to reduce the storage redundant that results from storing the row indices of the same row in the *COO* format as shown in the row array in Fig. 1, a modified storage format—*CSR* [3,4] which is shown in Fig. 2 was proposed. *CSR* format compresses the storage space by storing the position index of the first nonzero element rather than all nonzero elements in a row, for which the space overhead is reduced to $N(S_D + S_I) + (n + 1)S_I$. Compared with *COO*, this storage format can save the storage space to some extent. Moreover, the efficiency of algorithm using this storage format is higher than the algorithm using *COO*, because the indirect addressing is reduced when the multiplication was calculated. However, *CSR* does not eliminate irregular data accesses completely. Especially, when there are many nonzero elements in a certain row, it will lead to high cache miss rate. Vuduc and Moon pointed out that the efficiency cannot reach 10% of machine peak performance when using *CSR* in the sparse matrix multiplication calculation [5].

Quadtree storage format. The high cache miss rate will be caused when the *SpMVM* uses existing sparse matrix storage format. To solve this problem, quadtree storage format was put forward. Through splitting the sparse matrix recursively (see Fig. 3), quadtree storage format builds a quadtree, as shown in Fig. 4, rather than three arrays.

As shown in Fig. 4, there are three types of node defined in the quadtree: intermediate node and two types of leaf node. Intermediate nodes are mixed-region nodes (marked as M) that contain nonzero elements and zero elements and can be decomposed continually. Leaf nodes are divided into empty-region nodes (marked as E) and dense-region nodes (marked as D). However, since the sparsity of the matrix is increasing, there are more empty-field nodes to be stored. But E nodes contain useless information, resulting in storage redundancy. To solve this problem, Simecek came up with an extended quadtree storage format, in which the empty-region nodes are deleted and the dense-region nodes are divided into full-region and sparse-region nodes according to the sparsity of current region. However, this extension quadtree is not that fit for matrix–vector multiplication due to the large control and storage overhead [6].

To solve the above mentioned problems of existing storage structures, a lot of novel ideas and methods have been introduced in recent years. These studies mainly focus on two aspects: one is reducing the complexity of the algorithm, the other is improving the efficiency by taking advantage of architectural features. As for the former aspect, Simecek et al. presented a minimal Quadtree (*MQT*) [7] format to compress the sparse matrices storage space. *MQT* stores all the nodes with an array, and each node contains four flags (i.e., 4 bits only) instead of pointers, which greatly reduces the space complexity of the quadtree storage structure. In the same year, he proposed a large-scale sparse matrix storage format for massively parallel system, which is capable of minimizing the space complexity [8]. While, as for the latter aspect, memory bandwidth limitation is one of the main bottlenecks in the existing architectures. To solve this problem, [9] designed a *CSR* based compressed storage format, which reduces the memory bandwidth demand through compressing the index and value. However, for large-scale sparse matrix, compression and decompression will introduce huge overhead. Feng et al. proposed a segment based *SpMVM* CUDA algorithm SHEC-Segmented Hybrid ELL + *CSR* [10]. Nicholas Yzelman and Roose gave strategies for parallel shared-Memory sparse matrix–vector multiplication [11]. Im et al. [12,13] and Vuduc et al. [5,14] proposed optimized Cache-Register block storage data structure and presented auto-tuning methodology using architecture features. Buluc et al. [15] presented a storage format *CSB*, which improves the *CSR* storage format and makes it easy to parallel computing. Yuan et al. analyzed the performance of these structures in [16]. Sun et al. [17] designed *CRSD* format, in which the matrix is divided into three groups and each group is computed respectively. However, only when the matrix is diagonally dominant can this format get good performance. Based on their work, Williams et al. [18] optimized the method on different multi-core architectures. Kourtis et al. [19,20] improved the efficiency of matrix–vector multiplication calculation under the circumstances of multi-threading by optimizing the compression technology of an array and summarized the characteristics of the sparse matrix–vector calculation on the popular architecture nowadays. However, these methods depend highly on the distribution of nonzero elements in the matrix. If the distribution of nonzero elements is uncertain, in most cases, these methods are not well. Although Blelloch et al. [21] presented a hierarchical diagonal blocking (HDB) approach, and optimized the division of subtree by separators in graph theory, it is necessary to traverse the