OPEN ACCESS
IJoICT
ISSN 2356-5462
socj.telkomuniversity.ac.id/ijoict

Intl. Journal on ICT
Vol. 2, Issue. 1, June 2016. pp. 13-22
doi:10.21108/ijoict.2016.2.1.71

Sliced Coordinate List Implementation Analysis on Sparse Matrix-Vector Multiplication Using Compute Unified Device Architecture

Andi Ariffin [#1], Fitriyani [#2] and Izzatul Ummah [#3]

# School of Computing, Telkom University
Jl. Telekomunikasi Terusan Buah Batu Bandung 40257 Indonesia

[1] andi@ariff.in
[2] fitriyani.y@gmail.com
[3] izza@hpc.telkomuniversity.ac.id

Abstract

Many problems in the scientific area and in the real world application are modeled as sparse matrix. Thus sparse matrices are very important and studied by many researchers. As these problems grow in scale, parallel computing resources are required to meet their computational needs. Coordinate List (COO) is one of sparse matrix's formats. And SCOO format was developed by modified COO format and combined within an implementation using Compute Unified Device Architecture (CUDA)-architecture Graphics Processing Unit (GPU) to get better performance. This research addressed to evaluate the comparison of performance Sparse Matrix-Vector Multiplication (SpMV) using COO and SCOO format based on its memory usage and execution time. Results showed that although SCOO implementation for sparse matrix use memory 1.000529 larger than COO format, its performance is around 123.8 faster than parallel COO or 77 times faster than parallel COO using one of the available library for CUDA, named CUSP.

Keywords: COO, CUSP, SCOO, Sparse matrix, SpMV

I. Introduction

In daily life, it is often to find problems which can be formulated into mathematics form for solving them. Its form is useful to solve real world problems using available mathematics formula. Matrix can be used to represent information from real world related to the problem that has been formulated into mathematics form. This representation method is commonly used in several cases which use graph as its data structure [7]. Matrix obtained from a graph can be dense or sparse. Generally, a lot of matrix which represents real world problems are sparse matrices. It is very large in size but information stored inside is relatively small, which cause bigger computational resource needs to do some calculation into those matrices [8]. Flight scheduling, social media, internet, image, etc has produced large-scale sparce matrices data which is very interesting to analyze. Sparse Matrix-Vector Multiplication (SpMV) is one of the most frequent operation performed to the sparse matrix and needs long enough time because of involving a lot of multiplication and addition operation to do [1].

Received on Jan 2016. Accepted on June 2016

We consider linear system $Ax = b$. Many linear systems have a matrix A in which almost all the elements are zero, called sparse matrix. There are two general categories of numerical methods for solving $Ax = b$, Direct Methods and Iteration Methods. Sparse matrix-vector multiply (SpMV) is one of the most heavily used kernels in scientific computing. For a matrix A of order $n \times n$, it will need $4n^2$ bytes to store it in single precision. Thus a matrix of order 10.000 will need around 375 MB of storage. It would be too large to be stored in the computer's memory, and take long computational time. Large-scale matrices mean large memory, large bandwidth, and long computational time.

Hence sparse matrix usually stored in special format to store information that is sparse, such as Diagonal (DIA), Ellpack (ELL), Compressed Sparse Row (CSR), Coordinate List (COO), Coordinate List (COO), which expects reduced storage needed and improved operation performance to those matrices. Hoang-Vu Dang introduces a new sparse matrix format which is a modification of COO format, named Sliced Coordinate List (SCOO) [1]. SCOO accelerate the performance of SpMV implementation [1].

CUDA (Compute Unified Device Architecture) is a programming model and a computational platform developed by NVIDIA. CUDA allows software developers to utilize NVIDIA GPU (Graphics Processing Unit) to accelerate the process of computing. GPU has a great resource with thousand of cores and increasing every year. GPU CUDA allows us to solve the problem with huge data in parallel. CUDA programming models architecture as if SIMD (Single Instruction Multiple Data) parallel arcitecture. So CUDA is suitable for large computational problem with large data and less intruction such as matrix, graph, array ect. CUSP is a library for C/C ++ programming language. CUSP is an open-source like a generic parallel algorithms for sparse linear algebra and graphs computation in GPU with CUDA architecture

In this research, we evaluated the performance and memory usage of SpMV using COO and SCOO format. We campared serial COO, parallel COO using CUDA, parallel COO using CUSP, serial SCOO, and parallel SCOO.

II. LITERATURE REVIEW

A. Sparse Matrix

Matrix is an arranged entries, rectangular shape which placed in horizontal row and vertical column. Entries from a matrix is called element. Matrix symbol is in bold uppercase letter and element symbol is in lowercase letter. Two subscript put near the element to denote its position in the matrix, first subscript denote the row position and second subscript denote the column position [2].

For example, suppose a matrix as follows:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix}$$

Matrix **A** has 4 rows and 4 columns with 16 elements. From **A**, it can be known that $a_{11} = 1, a_{12} = 7$, etc.

Matrix which contain a lot of zero elements is called sparse matrix. If a matrix is sparse then it is possible to exploit its characteristics to reduce size or to shorten computational time needed to do basic linear algebra operation by avoiding storing zero value element and computational to the zero value element [3].

There are some commonly used sparse matrix format, such as:

1. *Diagonal (DIA)*

   DIA format is suitable to represent sparse matrix which its non-zero entries are limited to the diagonal of the matrix [3]. DIA formed by two array: data array – to store non zero values, and offsets array – to store each diagonal distance to the main diagonal. Diagonal which is over the main diagonal is positive, while diagonal below the main diagonal is negative.

   As an example, DIA format for matrix **A** is:

$$\mathbf{data} = \begin{bmatrix} * & 1 & 7 \\ * & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathbf{offsets} = [-2 \quad 0 \quad 1]$$

2. *Ellpack (ELL)*

   ELL is a more general form of DIA format and it is suitable for matrix which its architecture is like vector. An **M** × **N**-sized matrix with a maximum of **K** non-zero values in each row can be stored as a dense matrix with a size of **M** × **K** non-zero data array and indices array as an index of column.

   As an example, ELL format for matrix **A** is:

$$\mathbf{data} = \begin{bmatrix} 1 & 7 & * \\ 2 & 8 & * \\ 5 & 3 & 9 \\ 6 & 4 & * \end{bmatrix} \qquad \mathbf{indices} = \begin{bmatrix} 0 & 1 & * \\ 1 & 2 & * \\ 0 & 2 & 3 \\ 1 & 3 & * \end{bmatrix}$$

3. *Compressed Sparse Row (CSR)*

   CSR is the most popular multifunction sparse matrix representation. Like ELL format, CSR store column indices to indices array and non-zero values to data array. Third array, ptr, is a row marker which make CSR format possible to represent varying length of row.

   As an example, CSR format for matrix **A** is:

$$\begin{aligned} \mathbf{ptr} &= [0 \quad 2 \quad 4 \quad 7 \quad 9] \\ \mathbf{indices} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 3] \\ \mathbf{data} &= [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4] \end{aligned}$$

4. *Coordinate List (COO)*

   COO format is the simplest storing scheme. COO use row, indices, and data array, each store row indices, column indices, and non-zero values.

   As an example, COO format for matrix **A** is:

$$\begin{aligned} \mathbf{row} &= [0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3] \\ \mathbf{indices} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 2 \quad 3 \quad 1 \quad 3] \\ \mathbf{data} &= [1 \quad 7 \quad 2 \quad 8 \quad 5 \quad 3 \quad 9 \quad 6 \quad 4] \end{aligned}$$

5. *Sliced Coordinate List (SCOO)*

   SCOO format is a modified version to the current COO sparse matrix format. SCOO format also store row and column indices for each non zero entries like COO format, but SCOO format is sorted by column indices so entries placed in the same column are stored adjacently [3]. By moving entries which has identical column index together in an increasing order, SCOO expected to maximize regular access to the input vector which can increase performance [1].

This below SCOO format for matrix **A** with slice size = 2. In SCOO format, there is slice size terms which mean how many a matrix is divided by the amount of its row. Besides that, there is an index array which is not exists in COO format. Index array store the next entries order values when changing slice.

$$\begin{array}{|c|c|c|c|}
\hline
1 & 7 & 0 & 0 \\
\hline
0 & 2 & 8 & 0 \\
\hline\hline
5 & 0 & 3 & 9 \\
\hline
0 & 6 & 0 & 4 \\
\hline
\end{array}$$

c_index = [0 1 1 2 | 0 1 2 3 3]
r_index = [0 0 1 1 | 2 3 2 2 3]
value = [1 7 2 8 | 5 6 3 9 4]
index = [0 4]

So, for specific matrix, there is a preferred format which more suitable to its sparsity pattern rather than the other format. Bell identify sparsity pattern of a matrix into three types, which is: diagonal matrix, matrix with more or less same amount of row length, and those who are not. Each sparse pattern has its best format which is different between each other [3]..

B. Vector

Vector is a mathematics form to represent a magnitude which has values and direction [6]. Vector denoted by a lowercase letter with an arrow sign on the top of the letter $\left(\text{For example}: \vec{a}, \vec{b}\right)$. Vector can be represented by a column matrix which has $m \times 1$ size [4]. As an example, suppose $\vec{a} = (2i - 3j)$, then its vector can be written in a matrix form, for example matrix $\boldsymbol{B} = \begin{bmatrix} 2 \\ -3 \end{bmatrix}$.

C. Sparse Matrix-Vector Multiplication

If **A** is an m x n-sized input matrix, $\vec{x}$ is an $n \times 1$-sized input vector, y is an $m \times 1$-sized output vector, and γ is an amount of non-zero entries from **A**, then operation of multiplication between matrix and vector can be written as $y = \mathbf{A}.x$ [3]. The only requirement to do the operation is the amount of column in matrix **A** must be equal to the amount of element in vector **x**.

The algorithm to do the matrix-vector multiplication is as follows [1] :

```
Input: A: $m\ x\ n$-sized matrix ; x: input vector
Output: $y \leftarrow \mathbf{A}.x$
for $i \leftarrow 0$ to $m - 1$ do
        $y[i] \leftarrow 0$ for each non-zero entries at row $r$, column $c$ do
                $y[i] \leftarrow y[i] + A[r][c]$ x $x[c]$
        end for
end for
```

III. SYSTEM DESIGN

In this research, a system which is able to utilize NVIDIA GPU with CUDA will be created to do SpMV operation. This system refers to the efficient SpMV computational algorithm which use the power of GPU to do the calculation.

System designed as an overview consists of two main processes. The first step is to convert the matrix data which are used as an input to the COO and SCOO sparse matrix format. The second step is the process of multiplication between sparse matrix and vector. Fig. 1 below show the design of our system.

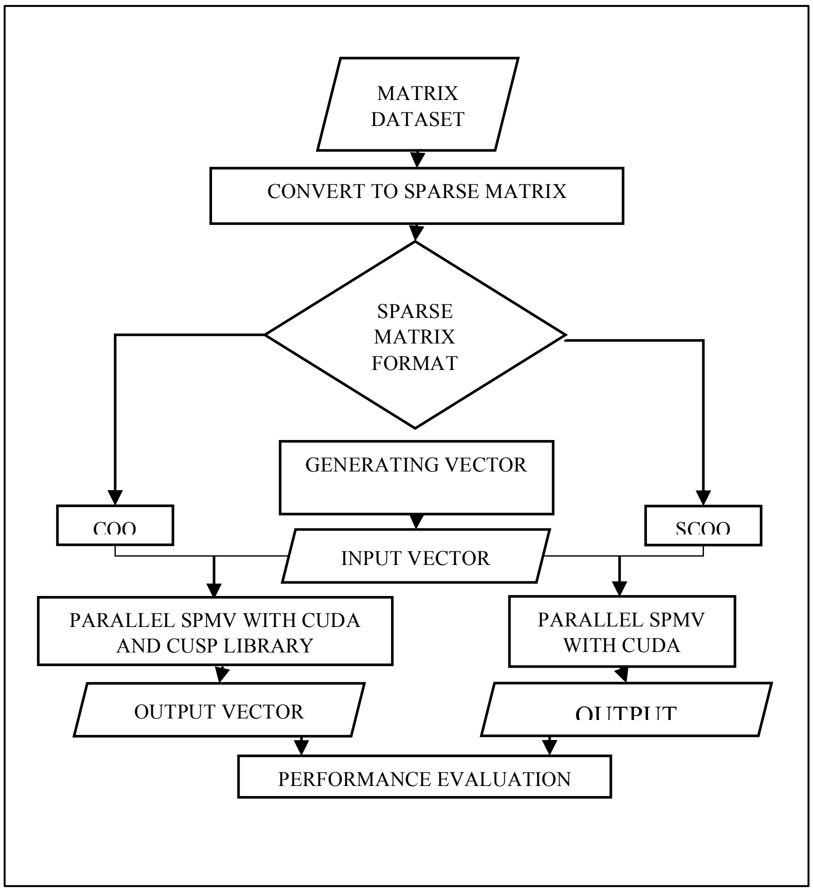

Fig. 1. SpMV Implementation System Design

A. *SpMV Dataset*

Dataset which will be used as an input matrix at the SpMV operation were obtained from the University of Florida Sparse Matrix Collection website [2]. This website provide sparse matrices in three different formats which is free to download, those formats are MATLAB, Rutherford/Boeing, and Matrix Market. However, the system designed is only able to handle sparse matrix format presented in Matrix Market (.mtx) format. The other format, MATLAB is a binary file and its usage is specific for the MATLAB application and Rutherford/Boeing format is nearly identical to the dense matrix because it is presented in the full-formed matrix which is elongated and widened.

We used 10 sparse matrices, which are considered to represent a lot of real world problems. Table 1 shows the description of 10 sparse matrices.

TABLE I
DATASET USED IN SYSTEM DESIGNS

Name	Amount of Row	Amount of Column	Amount of Non-Zero Entry	Sparsity Level	Problem/Description
olafu	16.146	16.146	1.015.156	0,389406%	Structural problem
gyro	17.361	17.361	1.021.159	0,3388%	Model reduction problem
gyro_k	17.361	17.361	1.021.159	0,3388%	Duplicate model reduction problem
Rim	22.560	22.560	1.014.951	0,199419%	Computational fluids dynamics problem
IG5-17	30.162	27.944	1.035.008	0,122799%	Combinatorial problem
case39	40.216	40.216	1.042.160	0,064437%	Electrical network problem order
dawson5	51.537	51.537	1.010.777	0,038055%	Structural problem
Dubcova2	65.025	65.025	1.030.225	0,024365%	2D/3D problem
torso2	115.967	115.967	1.033.473	0,007685%	2D/3D problem
pds-90	142.823	475.448	1.014.136	0,001493%	Linear programming problem

For the vector that will be used in SpMV operation, it will use one as default entries. Beside that, the vector dimension must be equal to the amount of column of the sparse matrix to do the SpMV operation.

B. *Convert to Sparse Matrix*

There are two sparse matrices format that used in the designed system, COO and SCOO. Input matrix file which is in *.mtx* type will be converted to those sparse matrix format before doing the SpMV operation. We used below algorithm to modified COO become SCOO.

```
Input : A {Matriks COO}, ss {Slice Size}
Output : B {Matriks SCOO}
ptr ← 0
B.idx[0] ← 0
for  i ← 0 to (A.jml_baris / ss) do
     for j ← 0 to A.jml_entri do
          if ((A.idx_baris[j] >= (i*ss)) and (A.idx_baris[j] < (i*ss + ss))) then
               B.idx_kolom[ptr] ← A.idx_kolom[j]
               B.idx_baris[ptr] ← A.idx_baris[j]
               B.entri[ptr] ← A.entri[j]
               ptr ← ptr + 1
          endif
          B.idx[i + 1] ← ptr
     endfor
endfor
```

C. *SpMV Operation*

Multiplication operation between sparse matrix and vector is different from the general matrix multiplication, because sparse matrix is represented in different format, in this case COO and SCOO. SpMV process will be conducted in parallel using CUDA at NVIDIA GPU and for the COO sparse matrix format, CUSP library will be used to help SpMV process.

D. *Performance Evaluation*

This process evaluate the performance result obtained from the SCOO sparse matrix format to its predecessor, COO, from the side of execution time needed to do the SpMV operation by both sparse matrix format and its memory usage to store matrix data in COO and SCOO format. We used personal computer with specifies below:

TABLE II
COMPUTER SPECIFICATION

Hardware	
Processor	Intel Core i7-4770K @3.5 Ghz
RAM	8 GBs
HDD	1 TB
GPU	NVIDIA GeForce GTX 750 Ti
Software	
Operating System	Microsoft Windows 10 64-bit
CUDA Tools Kit	CUDA v7.5.18 Visual Studio Community 2013 64-bit
CUDA Library	CUSP 0.5.1

IV. TESTING RESULTS AND ANALYSIS

We use CPU and GPU with specifies as Table II in our experiments. The following Table III shows the results of experiments about sparse matrices memory usage that used dataset on Table I. Memory usage is memory space that used to store matrix information.

TABLE III
SPMV IMPLEMENTATION RESULTS BASED ON ITS MEMORY USAGE (BYTE)

Matrix Name	Dense	COO	SCOO
Olafu	1.042.773.248	12.181.872	12.183.896
Gyro	1.205.617.280	12.253.908	12.256.084
gyro_k	1.205.617.280	12.253.908	12.256.084
Rim	2.035.814.400	12.179.412	12.182.236
IG5-17	3.371.387.648	12.420.096	12.423.872
case39	6.469.306.368	12.505.920	12.510.952
dawson5	10.624.249.856	12.129.324	12.135.772
Dubcova2	16.913.002.496	12.362.700	12.370.836
torso2	53.793.382.400	12.401.676	12.416.176
pds-90	271.619.637.248	12.169.632	12.187.492

Based on the results above, it shows that the matrices representation in dense format has the highest memory consumption. The best matrices representation for the least memory usage is COO format but just a little different with SCOO format. The memory consumption of COO format can be reduced until 99.54242% and 99.54232% when using SCOO format, compared to the dense format. So it is important to store sparse matrices in the specially designed form, such as COO and SCOO.

In this research, we compare SpMV using COO and SCOO format on CPU and GPU. We execute serial SpMV implementation for both sparse matrices format, parallel SpMV implementation for COO format, parallel SpMV implementation using CUSP library for COO format, and parallel SpMV implementation for SCOO format. For the implementation using dense matrix format can not be conducted because of the limitation of the devices used to handle very large memory needed.

Computational time needed for each data as of the dataset in the Table I to execute SpMV within the implementation as described above can be viewed on the following Table IV.

TABLE IV
SPMV IMPLEMENTATION RESULT BASED ON ITS EXECUTION TIME (IN MILLISECONDS)

Matrix Name	Serial COO	Parallel COO	Parallel COO with CUSP	Serial SCOO	Parallel SCOO
olafu	3.59	1.22	0.77	1.05	0.01
gyro	3.67	1.26	0.8	1.09	0
gyro_k	3.7	1.25	0.74	1.12	0.01
Rim	3.55	1.21	0.77	1.1	0.01
IG5-17	3.73	1.26	0.79	1.15	0
case39	3.69	1.3	0.78	1.08	0.01
dawson5	3.54	1.09	0.73	1.1	0
Dubcova2	3.74	1.21	0.79	1.15	0.01
torso2	3.68	1.25	0.76	1.13	0
pds-90	3.63	1.19	0.98	1.69	0

Based on the table above, it can be seen that serial SCOO implementation gives an enhanced performance around 3.18 times faster than serial COO implementation. Besides that, parallel SCOO implementation results is 123.8 times faster than parallel COO implementation or 77 times faster than parallel COO implementation using CUSP library.

V. Conclusion

The implementation SpMV in COO and SCOO format using CUDA-architecture GPU show that storing matrix in COO format can reduce memory usage around 99.5424216% less than dense matrix format and around 99.54232% if using SCOO format. It also gives speed up as shown the table below.

TABLE V
SPEED UP

		Speed Up
Serial COO	Serial SCOO	3.18
Parallel COO	Parallel SCOO	123.8
COO + CUSP	Parallel SCOO	77

Although implementing in SCOO need more memory space needed than COO memory usage, the speed up is more significant to consider. The increasing of performance is obtained from the index array which exists in SCOO format that make access to the specified matrix element located in the same slice faster. So finding matrix element in the certain position can be done without iterating from the first part of the matrix.

To develop this system in the future, some suggestions are listed below which can be used as the parameter to create the new system which has a better performance, which is:

A. Make SCOO as one of the most commonly used sparse matrix format to store sparse matrix, so it does not need to convert the matrix from another format to SCOO anymore.
B. Test the existing system design using larger dataset with better devices, so measuring the time needed to perform SpMV operation using SCOO format in parallel way can be more measurable and reliable compared to the other SpMV implementation

ACKNOWLEDGMENT

The authors would like to thank...

REFERENCES

Dang, H.V and Schmidt, B. (2013). CUDA-enabled Sparse Matrix–Vector Multiplication on GPUs using atomic operations. Parallel Computing, 39 (11), 737-750.

Richard.B, Gabriel, C, & John , S.(2014), Linear Algebra: Algorithms. Applications. and Techniques: Third Edition. USA, Academic Press.

Nicholas, Wilt. (2013). The CUDA Handbook: A Comprehensive Guide to GPU Programming. USA, Addison-Wesley Professional.

Halliday & Resnick (2011). Fundamental of Physics 9th Edition Extended. USA, John Wiley & Sons.

N. Bell & M. Garland (2009). Implementing Sparse Matrix–Vector Multiplication on Throughput-Oriented Processors, SC, ACM, 11.

N. Bell & M. Garland (2012). Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. <http://cusp-library.googlecode.com>, version 0.3.0.

Davis, T. A. & Y. Hu. "Tim Davis: University of Florida Sparse Matrix Collectio : sparse matrices from a wide range of applications". http://www.cise.ufl.edu/research/sparse/matrices/

Davis. Tim. "Research Sparse Matrix Algorithms and Software". http://faculty.cse.tamu.edu/davis/research.html.

Rauber,T. & Runger, G. (2010) Parallel Programming For Multicore and Cluster Systems, Springer.

NVIDIA Corporation, CUDA C Programming Guide, (2011) .<http://docs.nvidia.com/cuda/cuda-c-programming-guide/>.

NVIDIA. "What is GPU Computing? | High Performance Computing | NVIDIA | NVIDIA".
http://www.nvidia.com/object/what-is-gpu-computing.html

GilbJ.R., Reinhardt, S. & Shah V. B. (2006). Highperformance Graph Algorithms from Parallel Sparse Matrices,. Proc. of the

Int’l Workshop on Applied Parallel Computing.

Farzaneh,A., Kheiri, H. & Shahmersi, M.A. (2009). An Efficient Storage Format For Large Sparse Matrices, Commun. Fac.Sci. Univ.Ank. Series A1 Volume 58 , Number 2 , 1-10

Hong,S. & Kim, S. (2009) . Memory-level and Thread-level Parallelism Aware GPU Architecture Performance Analytical Model, ISCA’09 Proceeding, ACM.

Zlatev & Zahari (1991). Computational Methods for General Sparse Matrices. Netherlands: Springer.