

SMALL SCALE FOR PARALLEL PROGRAMMING

Sparse Matrix-Vector Product Kernel

April 8, 2018

António Pedro Araújo Fraga

Student ID: 279654

Cranfield University

M.Sc. in Software Engineering for Technical Computing

Contents

Introduction	4
Sparse Matrices Formats	4
Problem definition	6
Performance analysis	7
CPU vs GPU	7
Procedures	7
Parsing	8
Average non-zero deviation	9
OpenMP	9
Solution Design	9
Results & Discussion	9
Conclusions	10
Appendices	12
Source Code	13
Doxygen Documentation	13

Abstract

The product of a sparse matrix and a vector was calculated both in parallel and sequentially. The parallel procedure was executed with two different technologies, Open Multi-Processing and Compute Unified Device Architecture. Sparse matrices were stored in two different formats, Compressed Sparse Row and Ellpack. It was seen that the two formats were adequate for different types of matrices. Different procedures produced different effects, those effects were discussed and studied.

Table 1: Nomenclature

Matrix number of rows	m
Matrix number of columns	n
Sparse matrix	A
Vector to be multiplied	x
Resulting vector	b
Floating point operations per second	$FLOPS$
Number of non-zero values	nz
Maximum number of non-zero values per row	$maxnz$
Average computation time of a given kernel	T

Introduction

Bidimensional matrices are often represented in a bidimensional array of values of $m \times n$ elements. When matrices with this representation are multiplied by a vector, one has to iterate through every element of a matrix. This approach can be rather time expensive when dealing with large matrices.

Sparse Matrices have a particular characteristic. They are formed by a larger number of **zero** values, compared to the amount of **non-zero** elements. Therefore, these matrices can be represented in different formats in order to avoid multiplications by zero.[2] The distribution of **non-zero** values within these matrices is unpredictable. Hence, one has to come up with workable formats, allowing to have knowledge of a particular row, column and value.

Sparse Matrices Formats

Sparse Matrices can be stored in several formats like **DIA** (diagonal), more adequate for matrices which elements are spread across the diagonal, and **COO** (Coordinate Format).[2] In this project, the **CSR** (Compressed Sparse Row) and **Ellpack** formats were the only ones to be compared. Using **0-based indexing**, and having a matrix defined by,

$$A = \begin{pmatrix} \mathbf{3} & \mathbf{4} & 0 & 0 & 0 \\ 0 & \mathbf{5} & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{11} & \mathbf{12} & 0 & 0 \\ 0 & 0 & \mathbf{2} & \mathbf{13} & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{-1} \end{pmatrix} \quad \begin{array}{l} m = 5 \\ n = 5 \end{array}$$

the CSR format can be represented by declaring the number of *non-zero* values, *nz*. An array of values, *as*, of length *nz*. One array of pointers, *irp*, indicating which value in the previous array belongs to the next row. And finally, an array of columns of length *nz*, *ja*, indicating the column index of each value. Therefore, for the given matrix, one can write the described arrays and value as,

$$nz = 10$$

$$irp = \begin{pmatrix} 0 & 2 & 4 & 6 & 8 \end{pmatrix}$$

$$as = \begin{pmatrix} 3 & 4 & 5 & 1 & 11 & 12 & 2 & 13 & 1 & -1 \end{pmatrix}$$

$$ja = \begin{pmatrix} 0 & 1 & 1 & 2 & 1 & 2 & 2 & 3 & 3 & 4 \end{pmatrix}$$

Following a C-like syntax, the $A.x = b$ equation, with A following the previously described format, can be solved by,

```
for (int i = 0; i < m; ++i) {
    double temp = 0.0;
    for (int j = irp[i]; j < irp[i + 1]; ++j) {
        temp += as[j] * x[ja[j]];
    }
    y[i] = temp;
}
```

One can represent a matrix in the **Ellpack** format by defining two **bidimensional** arrays and one value. Declaring *maxnz* as the maximum number of non-zero values in a given row, one can declare two arrays of *m* rows by *maxnz* elements. Similarly to the CSR format, the first array, *ja*, contains the column index of each value per row. The second array, *as*, contains the given values per row,

$$\text{maxnz} = 2$$

$$as = \begin{pmatrix} 3 & 4 \\ 5 & 1 \\ 11 & 12 \\ 2 & 13 \\ 1 & -1 \end{pmatrix} \quad ja = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{pmatrix}$$

Following a C-like syntax, the $A.x = b$ equation, with A following the previously described format, can be solved by,

```
for (int i = 0; i < m; ++i) {
    double temp = 0.0;
    for (int j = 0; j < maxnz; ++j) {
        temp += as[i][j] * x[ja[i][j]];
    }
    y[i] = temp;
}
```

Problem definition

The problem consisted in analysing the performance produced by a developed kernel. This kernel was able to solve a **Sparse Matrix-Vector Multiplication**, $A.x = b$. Besides of being able to solve the previous equation, the code should be able to do it in parallel, using **OpenMP** (Open Multi-Processing) and **CUDA** (Compute Unified Device Architecture).

A set of matrices obtained from the University of Florida Sparse Matrix Collection[3], should be used to conduct the performance analysis. The files which contained the matrices were represented in the **Matrix Market** format, and they were read with software based on an existent reader. [4]

Performance analysis

The performance of a given method could be measured in *FLOPS*, the number of floating point operations per second. For each method, it was possible to calculate this value by the following formula,

$$FLOPS = \frac{2 \times nz}{T}$$

In order to calculate the resulting vector in the $A.x = b$ equation, one has to execute **two** floating point operations per *non-zero* value. Thus, the **numerator** expression of the previous division is obtained. The **denominator** is defined by the time, in seconds, to solve the equation.

CPU vs GPU

OpenMP, is a technology that makes use of threads running in **Central Processing Units**. These processors, are often few compared to the number of cores present in a **Graphics Processing Units** device, and the level of parallelism found in these devices is comparatively low to the level of parallelism found on GPUs. One can expect to find hundreds of cores working in parallel in a GPU.

On other hand, the CUDA technology, developed by NVIDIA, offers a coding interface with a C-like syntax. Not only it allows a fast shared memory between cores within a block, but the possibility of gather data from different blocks as well. These systems allow developers to build applications capable of processing larger sets of data more quickly.[5]

Procedures

A parsing mechanism was developed in order to convert information in the File System into the previously referred formats. The several files were in the Matrix Market format, which describes the size of the matrix, **M** rows and **N** columns, and the number of *non-zero* values.

Three distinct methods were developed. The first one computed the results sequentially, the second one computed the results with OpenMP, and the third used CUDA. All the three methods were capable of computing matrices in the CSR and Ellpack formats.

The results were exported into **csv** files.

Parsing

Parsing was done with by using one proper library to read sparse matrices in the **MatrixMarket** format. This format contains information about the type of matrix the program is about to read. These files begin with a banner, which describes the matrix author, date of creation, id and other fields.

```
%%MatrixMarket matrix coordinate real general
%-----
% UF Sparse Matrix Collection, Tim Davis
% http://www.cise.ufl.edu/research/sparse/matrices/vanHeukelum/cage4
% name: vanHeukelum/cage4
% [DNA electrophoresis, 4 monomers in polymer. A. van Heukelum, Utrecht U.]
% id: 905
% date: 2003
% author: A. van Heukelum
% ed: T. Davis
% fields: title A name id date author ed kind
% kind: directed weighted graph
%-----
```

The first line of the banner describes the type of the matrix being read. The first line of the body contains information about the number of rows, columns and non-zero values.

Most of the matrices in the proposed set were of type **coordinate real general**. The values are identified by three elements, row, column and value,

```
1 1 .75
2 1 .075027667114587
4 1 .0916389995520797
5 1 .0375138335572935
(...)
```


but the rows and columns are identified by using an **one-indexing** format. Therefore they had to be converted into the proper format in order to be used with C-like arrays.

Two matrices were of type **coordinate real symmetric**, which follow the format described above, but they only contained the upper or lower triangle of a given matrix. The matrix could trivially be represented in memory by adding one extra value with **row** and **col** swapped. Notice, that this process is not needed if a value is part of the main diagonal of a matrix, since the set contained square matrices only. The *non-zero* values were incremented in these type of matrices.

The last type of matrices is declared as **coordinate pattern general**. In this type of matrices the values were assumed to be **one**. Therefore the body contained information about the **row** and **column** of a given value only. These matrices are often used to represent graphs.

The values in the MatrixMarket format were sorted by rows, followed by sorted columns. This detail creates some problems when inserting values in the **CSR** format, since it is required to be represented them in order of columns as described previously. A special data structure was used to keep the values sorted by columns when representing them in memory, a **map**. This data structure is an implementation of a Red-Black tree, where is possible to know the correct order of an element in an $O(\log n)$ time. Notice that it is not important to keep values sorted by columns with these representations, therefore a vector of **pairs** was chosen to be the data structure in the map value. The first element of a given pair described a column, and the second element described a value. The values were then added to the correct representations.

Average non-zero deviation

OpenMP

Solution Design

Results & Discussion

Conclusions

References

- [1] Raphael Yuster and Uri Zwick, *Fast sparse matrix multiplication*, Available at: <<http://www.cs.tau.ac.il/~zwick/papers/sparse.pdf>> [Accessed 28 March 2017]
- [2] B. Neelima1 and Prakash S. Raghavendra, April 2012, *Effective Sparse Matrix Representation for the GPU Architectures*, Available at: <<https://pdfs.semanticscholar.org/2d15/dd5d0975fff797397ad31059ec097b659e00.pdf>> [Accessed 28 March 2017]
- [3] University of Florida, *Sparse Matrix Collection*, Available at: <<https://sparse.tamu.edu/>> [Accessed 28 March 2017]
- [4] Matrix Market, *ANSI C library for Matrix Market I/O*, Available at: <<https://math.nist.gov/MatrixMarket/mmio-c.html>> [Accessed 28 March 2017]
- [5] B. N. Manjunatha Reddy, Dr. Shanthala S., Dr. B. R. VijayaKuma, February 2017 *Performance Analysis of GPU V/S CPU for Image Processing Applications*, Available at: <<https://www.ijraset.com/files/serve.php?FID=6250>> [Accessed 28 March 2017]

Appendices

Source Code