Small Scale for Parallel Programming

Sparse Matrix-Vector Product Kernel

April 9, 2018

António Pedro Araújo Fraga

Student ID: 279654

Cranfield University

M.Sc. in Software Engineering for Technical Computing

Contents

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,

$$A = \begin{pmatrix} \mathbf{3} & \mathbf{4} & 0 & 0 & 0 \\ 0 & \mathbf{5} & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & \mathbf{2} & 0 & 0 \\ 0 & 0 & \mathbf{2} & \mathbf{3} & 0 \\ 0 & 0 & 0 & \mathbf{1} & \mathbf{6} \end{pmatrix} \qquad \begin{matrix} m = 5 \\ n = 5 \end{matrix}$$

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,

$$nz = 10$$

$$irp = \begin{pmatrix} 0 & 2 & 4 & 6 & 8 & 11 \end{pmatrix}$$

$$as = \begin{pmatrix} 3 & 4 & 5 & 1 & 1 & 2 & 2 & 3 & 1 & 6 \end{pmatrix}$$

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

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

```c
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,

$$maxnz = 2$$

$$as = \begin{pmatrix} 3 & 4 \\ 5 & 1 \\ 1 & 2 \\ 2 & 3 \\ 1 & 6 \end{pmatrix} \qquad ja = \begin{pmatrix} 0 & 1 \\ 1 & 2 \\ 1 & 2 \\ 2 & 3 \\ 3 & 4 \end{pmatrix}$$

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.

The performance of the methods using the CSR format can vary with the deviation of amount of *non-zero* values. Thereby, the percentage of the average amount of non-zero values per row was calculated. This value can be obtained by the sum of absolute differences between each number of non-zero values per row and the mean number of non-zero values per row. Followed by a division by its mean, and multiplying by one hundred.

$$AverageDeviation = \frac{1}{n}\sum_{i=1}^{n}|x_i - mean|$$

$$PercentAvgDev = 100 \times \frac{AverageDeviation}{mean}$$

With this information, it was possible to verify whether this factor influences CSR matrices vector product efficiency or not.

CPU vs GPU

OpenMP, is a technology that makes use of threads running in **C**entral **P**rocessing **U**nits. These processors, are often few compared to the number of cores present in a **G**raphics **P**rocessing **U**nits 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.

With OpenMP, one expects to declare a **directive**, that indicates which configurations to follow in a given parallelized section. These directives are of the form,

```
#pragma omp (...)
```

, *pragma* indicates to the compiler that the following statement is not part of the C-like syntax. And *omp* indicates that the following configuration fields are part of the OpenMP technology.

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] in order to manage such large amount of threads, the **GPU** uses an Single Instruction Multiple Thread architecture. In this architecture threads are executed in groups of **32** called **warps**. A warp executes one instruction at a time, thus, a model of one instruction and multiple threads is achieved. Thread operations are called **coalesced** when threads within the same block are accessing memory within the same segment. But accessing scattered locations results in **memory divergence**, and requires a larger number of memory transactions. These transactions are time expensive, therefore, it is important to avoid them.

The basic usage of CUDA implies that the computational effort is divided into threads, threads are divided into thread blocks and thread blocks are divided into grids.[6]

Crescent

Crescent is the new machine designated to **H**igh **P**erformance **C**omputing in Cranfield University. Students have access to this machine when requested, therefore all the results in this document are related with this machine. Since a single program was developed to solve the problem with the several methods, the GPU queue was used. The node specifications are,

- Two intel E5-2698 v3 (Haswell) CPUs giving 32 CPU cores;
- 256GB of shared memory;
- Four Tesla K40 GPU cards.

The theoretical peak processor performance, including GPUs is estimated to be 28.2 TFlops.

Design Specification

This project followed an Object-Oriented programming design pattern.

Solution

A class was created to manage **input** and **output** information, *IOmanager*. This object was responsible to read information from the file system, converting them to data structures which were capable to represent them in memory.

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 represented them by sorted rows as described previously. A special data structure was used to keep the values sorted by rows 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 objects, CSR and Ellpack. These objects were inherited by a **Matrix** class.

These objects were responsible to keep track of the average of execution time, and to calculate the average deviation of *non-zeros* per row. After performing calculations with each method, the *IOmanager* object would export these results into **csv** files.

Bash scripts were used to exchange information with Crescent using the **S**ecure **C**opy **P**rotocol. These scripts were able to send updated source code to the mentioned machine, and to receive the obtained results. Having this process automated allowed to concentrate efforts in development and testing. A makefile and a **PBS** script were developed as well.

The class diagram regarding this project is available in the **Appendices** section.

Standards

This project versions were controlled by making use of GitHub. The project contained two main branches, with extra branches created with the purpose of developing a new feature.

Naming conventions were adopted in order to obtain a coherent and readable code. This project followed Google C++ naming conventions.

Test Plan

A test plan was developed in order to perform both correctness and performance tests.

Correctness was tested by making use of **Catch**, which is a tool that allows to require certain boolean expressions to be true. Certain matrices are downloaded with a **b** and **x** file. These files include arrays that are solutions to the

$$A.b = x$$

equation. These files of matrix A and array **b** were read and the result was compared with the file containing the array **x**. The results were compared to an order of $1.e-05$. The reason of establishing such order is due to **round-off** errors. **Unit tests** were developed in order to test each method correctness. A possible output of this tool can be observed below.

```
========================================================================
All tests passed (30 assertions in 5 test cases)
```

A printing method was developed for each format in order to guarantee that each array was stored correctly. This method was specially useful when reading small matrices. Taking the matrix previously described as an example, the *irp*, *ja* and *as* objects could be printed to the standard output.

The program should be able to run with a **memory check** tool. **Valgrind** and **cuda-memcheck** were an example of tools that allowed to verify whether *out of bounds* accesses were being done or not. These tools were useful to avoid memory leaks as well. **cuda-memcheck** is able to spot such problems inside a CUDA kernel, whereas Valgrind is able to do it with code running in CPU cores.

Performance testing was firstly made by changing the type of arguments in both OpenMP directives, and CUDA kernel launchers. These changes allowed to verify whether the performance measurements fluctuated or not.

The second approach was to make use of profilers, these tools are useful when dealing with problems of load balancing between threads. Flame Graphs, a tool developed by Brendan Greg that makes use of the **perf** command on linux, was able to export a more user-friendly perf output[7].

Memory bandwidth variations were tested as well. By improving variable locality, the code could improve its performance. One of the examples would be to avoid reductions among threads.

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. The methods were executed 20 times, and the computational time average was taken from there, attenuating possible fluctuations. All the three methods were capable of computing matrices in the CSR and Ellpack formats.

The reason why an extra sequential method was developed, was due to the fact that compilers might skip optimizations with code inside an OpenMP block. Thereby, it would be unfair to declare the OpenMP method running with one thread as sequential.

Parsing

Parsing was performed 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 matrix that is 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 since the set contained square matrices only, this process is not needed if a value is part of the main diagonal of a matrix. The *non-zero* values were incremented in these type of matrices.

The last type of matrices is declared as **coordinate pattern general**, where the values were assumed to be **1.0**. Thus, the body contained information about the **row** and **column** of a given value only. These matrices are often used to represent graphs.

OpenMP

The approach taken when using this technology was to launch a thread responsible to compute a given number of rows. The matrix was split into **t** partitions, and it could be graphically represented at **figure 1**.

In the previously described example, assuming that $t = 4$, the CSR arrays would be divided into four equal parts. Since the example matrix has 5 rows, one of the threads would compute

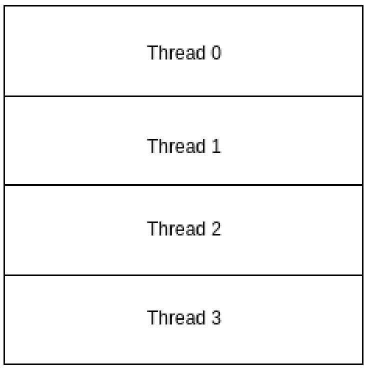

Figure 1: Threads split through the several matrix rows.

an extra row. This division is usually responsibility of the OpenMP technology, unless the user specifies the size of chunks to be computed per each thread.

$$irp = \begin{pmatrix} 0 & 2 & 4 & 6 & 8 & 11 \end{pmatrix}$$

$$as = \begin{pmatrix} \boxed{3 \quad 4} & \boxed{5 \quad 1} & \boxed{1 \quad 2} & \boxed{2 \quad 3 \quad 1 \quad 6} \end{pmatrix}$$

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

The Ellpack format would be divided in a similar way.

$$as = \begin{pmatrix} \boxed{3 \quad 4} \\ \boxed{5 \quad 1} \\ \boxed{1 \quad 2} \\ \boxed{\begin{matrix} 2 & 3 \\ 1 & 6 \end{matrix}} \end{pmatrix} \begin{pmatrix} \boxed{0 \quad 1} \\ \boxed{1 \quad 2} \\ \boxed{1 \quad 2} \\ \boxed{\begin{matrix} 2 & 3 \\ 3 & 4 \end{matrix}} \end{pmatrix} = ja$$

The parsing was done with only one thread, therefore some matrix values were already in the reading processor cache memory. Hence, in order to produce "fair" results, the first run was not included in the average of computational times. If it was included, the sequential execution would have a clear advantage on this method. CSR computations could be done with,

```
int i;
#pragma omp parallel for private(i) schedule(static) num_threads(t)
for (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;
}
```

and taking the same approach, one could compute matrices in the Ellpack format with,

```
int i;
#pragma omp parallel for private(i) schedule(static) num_threads(t)
for (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;
}
```

These methods were executed from **2** to **16** threads. The method described in this section can be referred as the scalar method.

CUDA

The first approach to solve this problem with CUDA was to launch a **kernel** per row. This method is very similar to the scalar method previously described, and might be the most intuitive and simple solution, but it creates some problems for matrices in the CSR format. Memory access from threads within the same *warp* is not coalesced, thus, like explained before, it is expected that this will become an overhead.

The following code sample can be used as a CSR scalar approach,

```
__global__ void scalarCSR(int * m, int * irp, int * ja, double * as, double *
    x, double * y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < *m) {
        double temp = 0.0;
        for (int j = irp[i]; j < irp[i + 1]; ++j) {
          temp += as[j] * x[ja[j]];
        }
        y[i] = temp;
  }
}
```

and the the memory access pattern can be described as follows, where the number in the iterations arrays indicates the row to be accessed,

$$irp = \begin{pmatrix} 0 & 2 & 4 & 6 & 8 & 11 \end{pmatrix}$$
$$as = \begin{pmatrix} 3 & 4 & 5 & 1 & 1 & 2 & 2 & 3 & 1 & 6 \end{pmatrix}$$
$$ja = \begin{pmatrix} 0 & 1 & 1 & 2 & 1 & 2 & 2 & 3 & 3 & 4 \end{pmatrix}$$

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

Thereby, by launching a warp per row, is possible to avoid this situation. This methods was denominated as **Vector-Strip Mining**. It was created an array as shared memory among threads within the same block. In the end, **sequential** reduction was done among threads within the same **warp**, reducing the value to the first warp thread. The following piece of code would be an implementation of such strategy [6],

```
__global__ void vectorMiningCSR(int * m, int * irp, int * ja, double * as,
    double * x, double * y) {
  extern __shared__ volatile double sdata[];

  int i = blockIdx.x * blockDim.x + threadIdx.x;;
  int warp = i / 32;
  int lane = i & (32 - 1);
  int row = warp;

  sdata[threadIdx.x] = 0;

  if (row < *m) {
     for (int j = irp[row] + lane ; j < irp[row + 1]; j += warp_size)
         sdata[threadIdx.x] += as[j] * x[ja[j]];

     if (lane < 16) { sdata[threadIdx.x] += sdata[threadIdx.x + 16]; }
     if (lane < 8) { sdata[threadIdx.x] += sdata[threadIdx.x + 8]; }
     if (lane < 4) { sdata[threadIdx.x] += sdata[threadIdx.x + 4]; }
     if (lane < 2) { sdata[threadIdx.x] += sdata[threadIdx.x + 2]; }
     if (lane < 1) { sdata[threadIdx.x] += sdata[threadIdx.x + 1]; }

     if (lane == 0)
       y[row] += sdata[tid];
  }
}
```

The memory access pattern is described as follows,

$$irp = \begin{pmatrix} 0 & 2 & 4 & 6 & 8 & 11 \end{pmatrix}$$
$$as = \begin{pmatrix} 3 & 4 & 5 & 1 & 1 & 2 & 2 & 3 & 1 & 6 \end{pmatrix}$$
$$ja = \begin{pmatrix} 0 & 1 & 1 & 2 & 1 & 2 & 2 & 3 & 3 & 4 \end{pmatrix}$$

$$\textit{Iteration 0} = \begin{pmatrix} 0 & 0 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \end{pmatrix}$$
$$\textit{Iteration 1} = \begin{pmatrix} \_ & \_ & 1 & 1 & \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ \end{pmatrix}$$
$$\textit{Iteration 2} = \begin{pmatrix} \_ & \_ & \_ & \_ & 2 & 2 & \_ & \_ & \_ & \_ & \_ & \_ \end{pmatrix}$$
$$\textit{Iteration 3} = \begin{pmatrix} \_ & \_ & \_ & \_ & \_ & \_ & 3 & 3 & \_ & \_ & \_ & \_ \end{pmatrix}$$
$$\textit{Iteration 4} = \begin{pmatrix} \_ & \_ & \_ & \_ & \_ & \_ & \_ & \_ & 4 & 4 & 4 & 4 \end{pmatrix}$$

In this approach is essential that each warp computes only one row, and it will produce better results if the minimum number of *non-zero* values per row is greater than 32. If so, it will minimize the number of threads idling during the execution of such kernel. With the scalar kernel avoiding this problem becomes more complicated, but it would be minimized if a matrix had the same number of non-zero values per row.

The strategy to solve the Sparse Matrix-Vector product when dealing with matrices stored in the Ellpack format was a scalar approach. One kernel per row was launched. Notice that this format has a major drawback of having to perform multiplications by *zero*.

Results & Discussion

Analysing the produced results was not trivial. One can consider several metrics in order to obtain a proper analysis. As referred before, two main metrics were used. The calculated value of *FLOPS* and the *average non-zero deviation* per row. One should keep in mind that memory may have a big impact in the calculations of this specific problem. The size of *Sparse Matrices* representations in memory may get to **GigaBytes**, therefore the procedure may achieve a high number of transactions between the several memory levels. Memory transactions are quite expensive, and they should be avoided.

CSR & Ellpack

By taking OpenMP as an example, it could be seen that by having matrices with a very attenuated number of non-zero values per row, the performance of both formats could increase at a similar ratio when the number of threads was increased.

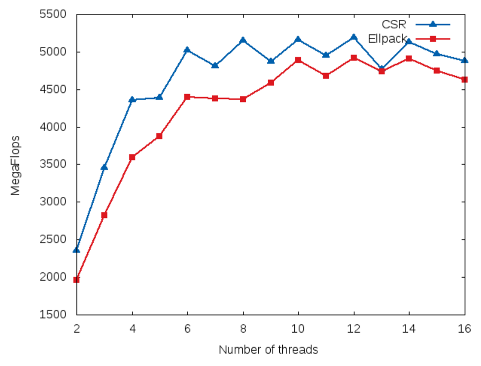

Figure 2: ML_ Laplace (1.45% of nz deviation).

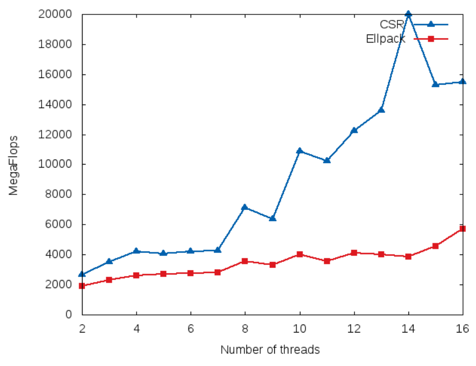

Figure 3: cant (20.2% of nz deviation).

By observing **figure 2**, it is possible to observe that this phenomenon occurs with matrix *ML_ Laplace*, a matrix with 1.45% of average non-zero deviation. But by analysing **figure 3**, one can determine that such behaviour is not true for matrices with a larger value of *average* *non-zero deviation* per row. One example of this scenario is matrix *cant*, with 20.2 % of this value.

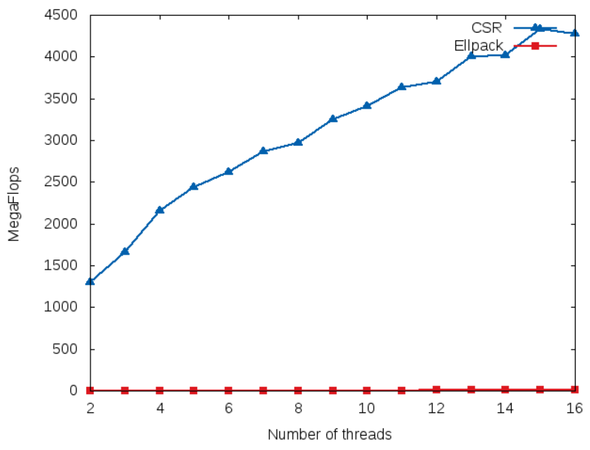

Figure 4: 1M (60.12% of nz deviation).

This behaviour is even more radical when dealing with matrices with a higher deviation. The matrix which results can be seen at **figure 4**, not only has a high deviation of non-zero values per row, but the number of elements to be processed is significantly huge. This is a major overhead for the *Ellpack* format, since multiplications by zero are performed every time a row does not have the maximum number of *non-zero* values.

OpenMP & CUDA

Parallelization becomes a overhead when one tries to launch threads that will be idle. Such behaviour can be easily seen with small matrices. By analysing **figure 5**, one can deduce that the number of calculations to be done within this matrix is relatively low. Hence, the performance of parallelization decreases when the number of threads is increased. The overhead of launching threads becomes obvious in these situations.

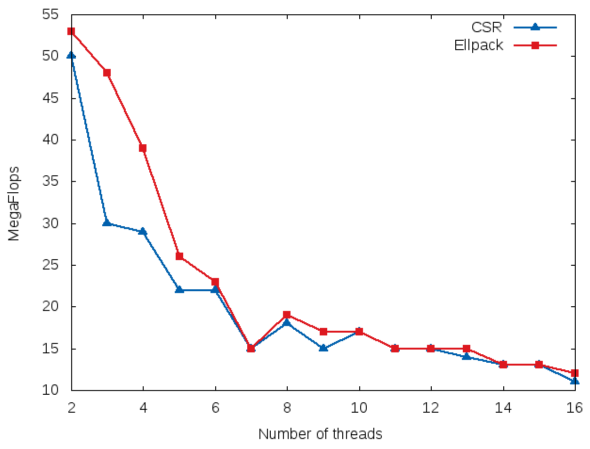

Figure 5: OpenMP results of Cage_ 4, 9 rows, 9 columns, 49 non-zero values.

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 2018]

[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 2018]

[3] University of Florida, *Sparse Matrix Collection*, Available at: <https://sparse.tamu.edu/> [Accessed 28 March 2018]

[4] Matrix Market, *ANSI C library for Matrix Market I/O*, Available at: <https://math.nist.gov/MatrixMarket/mmio-c.html> [Accessed 28 March 2018]

[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/fileserve.php?FID=6250> [Accessed 28 March 2018]

[6] Nathan Bell and Michael Garland, December 11, 2008 *Efficient Sparse Matrix-Vector Multiplication on CUDA*, Available at: <http://wnbell.com/media/2008-12-NVR-SpMV/nvr-2008-004.pdf> [Accessed 8 April 2018]

[7] Brendan Greg, *CPU Flame Graphs*, Available at: <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html> [Accessed 9 April 2018]

Appendices

Project Plan

Class Diagram

Source Code