Impact of Row Sorting on the Sparse Matrix-Vector Product on GPU

[1]Paula Prata, [2]João Muranho, [3]Abel Gomes

[2]University of Beira Interior, Department of Informatics, Portugal1,
[1,3]Instituto de Telecomunicações (IT), Portugal2
Email: [1]pprata@di.ubi.pt, [2]muranho@di.ubi.pt, [3]agomes@di.ubi.pt

Abstract— Modern graphics processing units (GPUs) allow for a high throughput in processing a massive amount of floating-point operations. The sparse matrix-vector (SpMV) product is of particular interest, and subject to intensive research in the GPGPU (general-purpose GPU) computation community, because it is repeatedly used in numerous scientific and engineering applications such as, for example, circuits simulation and ground-water model simulation among others.

Bearing this in mind, we have studied the impact of the sorting of matrix rows, according to their length (that is, by the number of nonzero values), on the GPU performance in the computing of the SpMV product, using synthetic matrices and benchmark matrices of different dimensions. Although sorting the rows is a commonly used technique in matrix operations, our study shows that sorting the rows by their length is useless regarding time performance gains in current GPU architectures.

Index Terms— CUDA, data parallelism, general purpose GPU, sparse matrix-vector product.

I. INTRODUCTION

This article describes a study about the impact of reordering the rows of a sparse matrix on the performance of the sparse matrix-vector (SpMV) product operation using Nvidia graphics processing units (GPUs). Albeit the SpMV product is a very simple operation, we have to be aware that it is a critical operation to the overall performance of a number of applications in science and engineering [11,16] because it is executed thousands or even millions of times, as happens in iterative methods for solving sparse linear systems [14].

GPUs are especially useful to carry out numerically intensive computations with data parallelism, where analogous calculations are performed on large quantities of data that are regularly organized (for example, vectors and matrices). Several studies show that GPUs can deliver a very high performance in computations involving dense matrices [1,19]. However, unlike dense matrices, the sparse matrices have not a regular structure, what does affect the performance of the solver of sparse matrix linear systems. But, as the practice shows, the performance of a solver of this kind also depends on further three important cornerstones:

- matrix storage formats;
- SpMV product algorithms;
- reordering algorithms.

Note that the focus of this paper is on the impact of row reordering on the overall performance of sparse matrix-vector (SpMV) product algorithms using different storage formats. More specifically, our work studies the impact of reordering the matrix rows on the SpMV product operation, using the length of each row as a reordering criterion. The idea behind this procedure of reordering is to get a better load balancing across the threads of each scheduling unit, even if that is obtained at the expense of some loss of locality of the rows and this is the main contribution of this paper.

In this paper we start by briefly describing the parallel computing architecture of GPU/CUDA in Section II. Afterwards, we proceed to Section III, where the storage formats for sparse matrices are also briefly described. The SpMV algorithms are approached in Section IV. Next, in Section V, we present the experimental results obtained in studying the impact of sorting the matrix rows on the performance of the SpMV product. Section VI discusses those results obtained in testing. Finally, in Section VII we present the conclusions and some hints for future work.

II. GPU/CUDA PARALLEL COMPUTING ARCHITECTURE

A.Nvidia GPU Architecture

An Nvidia GPU is an array of a number of multithreaded streaming multiprocessors with a global memory space. Each multiprocessor consists of a set of scalar processor cores (each with a set of registers) communicating via shared memory [21]. These many-core processors are capable of very high throughput computations, but they do not possess large caches that allow for memory access optimisation. To hide the high latency in global memory access, and take advantage of GPU processing power, it is necessary to running thousands of threads. It is also necessary to distribute the workload over the threads of warps (i.e., scheduling units in CUDA) in a balanced manner. Each warp only terminates when all its threads got concluded.

B. CUDA Parallel Computing Model

The Compute Unified Device Architecture (CUDA) was designed to take advantage of the many-core processors of Nvidia GPUs [12]. Basically, a CUDA application consists of a process running on host side (CPU) that launches functions (kernels) on device side (GPU). Each kernel running on GPU side is simultaneously executed by a high number of threads. The threads are organized in blocks, and the set of blocks constitutes an execution grid. We can say that one thread is associated to one scalar processor core, a block of threads is associated to one multiprocessor, and a grid of blocks is mapped onto the GPU (see Figure 1). Once a block of threads is assigned to a multiprocessor, it becomes responsible by splitting them into warps that get scheduled across several cores. Threads that belong to the same block may exchange and share data via the multiprocessor shared memory. Blocks waiting to be processed are automatically launched as soon as one of the multiprocessors becomes available.

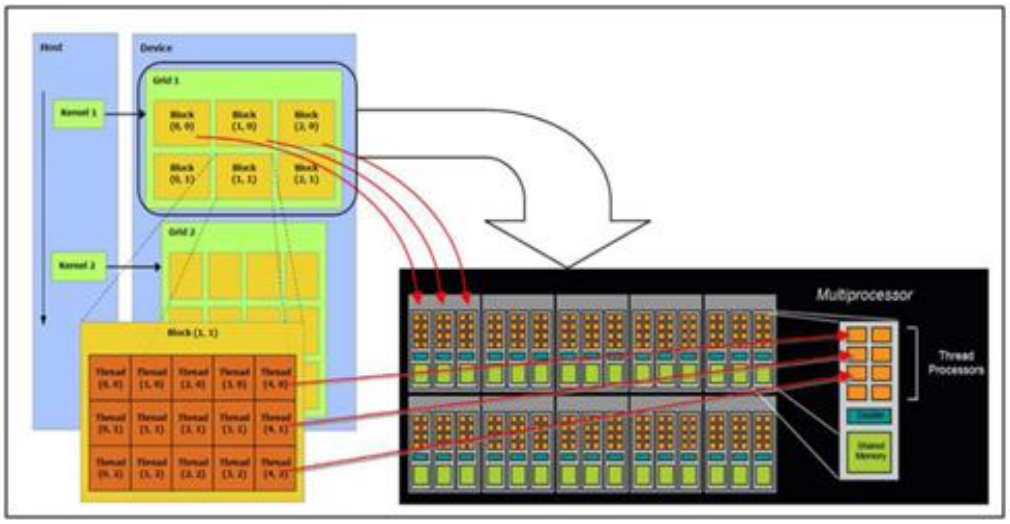

Fig. 1 Association between the structure of the software and the architecture of the hardware (adapted from [12])

Global memory can be accessed by any thread in the program, but has high access latency. Beyond global memory, the GPU has two additional (read-only) memories, texture memory and constant memory. Whenever an element is stored in the texture cache, it passes to be accessed in texture memory, which is far faster than global memory, in subsequent requests.

It is clear that, before launching a kernel on GPU side, the data must be copied into the global memory of GPU. Terminated the execution of the kernel, the output results must be copied back to CPU memory.

III. SPARSE MATRIX STORAGE FORMATS

A sparse matrix is usually stored in a compact format in order to just keep the nonzero values and its indexed location. The most common matrix storage formats in GPU-based sparse linear systems are the following:

- the coordinate format, also known as COO format [14].
- the compressed sparse row (CSR) format [14];
- the ELL format and its variants [7].

For a visual representation of these storage formats, the reader is referred to [2], [17].

A. The COO Format

The COO format is the reference storage format for sparse matrices. Let N be the number of nonzero elements of the sparse matrix A. In the COO format, the matrix A translates itself into three vectors (or one-dimensional arrays) of size N: the row indexing vector, where the row indices of the nonzero elements are stored; the column indexing vector, where the column indices of the nonzero elements are stored; the data vector that stores the nonzero elements themselves.

This format is used to initially store all the matrices studied in this work. Usually the matrices are converted to compressed formats which can lead to no negligible savings in storages [14]. The COO format implies a number of GPU global memory accesses larger than the use of compressed formats which results in a performance penalization.

B. The CSR Format

The CSR format extends the COO format. The main difference between these two formats is that the row-indexing array of COO format is replaced by a shorter array in the CSR format. The i-th component of this array contains the number of nonzero elements of the 0- to i-th rows of the sparse matrix. Therefore, this shorter array is an array of row offsets, as needed to index the

elements of the second and third arrays, say the column indexing array and the data array, respectively.

C. The ELL, HYB and ELL-r Formats

The ELLPACK/ITPACK or ELL format uses two vectors, one for the nonzero values and another for the column indices. Supposing a matrix with M rows where K is the maximum number of nonzero elements per row, the nonzero values are stored by columns assuming that each row has length K. In rows with a number of nonzero elements less than K the final positions will be zero-padded to length K. Then, the first M elements of the data vector will be the first nonzero element of the first row, the first nonzero element of the second row and so on until the first nonzero element of the M^{th} row. The second vector, the column indices vector, corresponds to a M-by-K matrix stored in column major order, where each position contains the column of the value that is at the corresponding position of the data vector.

For matrices where the length of the rows has high variability, the percentage of zeros in the data vector is high and the performance decreases. For those cases, the hybrid format (HYB) proposed in [2] combines ELL and COO formats. When a matrix has a certain number of rows with length greater than an empirical threshold, the typical number of nonzero values is stored in the ELL format and the remaining entries of larger rows are stored in the COO format.

Finally, the ELLPACK-r (ELL-r) format optimises the ELL format by using a third vector to store the number of nonzeros of each row [17], [18].

IV. SPARSE MATRIX-VECTOR PRODUCT ON GPU

Now, let us see how the SpMV operation can be implemented on GPU. In fact, a number of sparse matrix-vector product algorithms can be classified in the basis of the dichotomy between the number of threads and the structuring element of the matrix as follows:

- 1 thread per nonzero matrix element (TpE algorithm);
- 1 thread per matrix row (TpR algorithm);
- 1 warp (actually, 32 threads) per matrix row (WpR algorithm).

Moreover, according to the work presented in [2], the results produced by combining those three storage matrix formats and those three SpMV product algorithms suggest that the best performance of the SpMV product is attained in two circumstances:

- using TpR algorithm together with the ELL format.
- using WpR algorithm together with the CSR format;

The WpR algorithm is the most efficient algorithm only when the length of the rows is long enough to feed the entire warp of threads. With the WpR algorithm each matrix row is processed by one warp, thus sorting rows is useless.

A. Thread per Row Algorithm Using the ELL Format

Before proceeding any further with the TpR algorithm to carry out the SpMV product within the ELL format, we need more context about the ELL format itself.

Indeed, a previous study presented in [2], [3] shows that the CUDA implementation of the SpMV product produces the best performance using the ELL storage format, provided that the maximum number of nonzero elements per row does not substantially differ from the average. This format allows that the matrix data to be processed by one warp is stored in contiguous memory positions and, therefore, the memory access times can be shortened. If the matrix rows handled by each warp have approximately the same length (that is, the same number of nonzero elements) then all threads will be simultaneously occupied without wasting of computing capacity. But, if the number of nonzero elements per row varies significantly, those authors propose the hybrid format (HYB) described in Section III. C.

Recently, some proposals on automatically tuning the SpMV operations on GPU have appeared in the literature [5], [6], [10]. In [10], a new modified version of ELL, the sliced ELL format is proposed. In this format, the matrix is divided into slices, that is, in strips of a number of adjacent rows, being each slice separately stored in the ELL format. A variable number of threads can be assigned to each slice of matrix rows. In [5] another modification to ELL, the blocked ELL (BELLPACK), is proposed. The matrix is divided into small dense blocks that were stored contiguously. In that way, we can reduce the amount of storage needed for matrices having a block structure. In an analogous manner to the sliced ELL format, the matrix in the blocked ELL format is also divided into slices. In both cases, the slicing process is combined with a row reordering process in order to bring together rows of similar length, as well as to reduce storage. But, as referred in [10], reordering rows may result in degrading the performance. Adjacent rows usually have closer nonzero elements than unrelated rows, so that the shuffling of rows can increase the number of cache misses on accesses to vector to be multiplied by the sparse matrix.

Some works on integer sparse-matrix product in GPUs also propose reordering the rows by their length before building sliced formats [15] and blocked formats [4]. Finally, row sorting is also used in [9], having these authors proposed a new storage format with the intent of reducing the memory footprint of the ELL format.

On the other hand, performing the SpMV product on GPU using the TpR (one thread per row) algorithm on a

ELL-format matrix makes the threads that belong to the same warp to access contiguous memory positions at each iteration. When the thread i processes the row i, it accesses the j-th component of the row i in the j-th iteration. As the matrix is stored in the column major order, all these elements are stored in consecutive memory positions. In the same warp, a single instruction is executed at a time across all its threads, following a Single Instruction Multiple Thread (SIMT) model. Thus, we have all threads in the warp accessing contiguous memory positions. This pattern of memory access, known in CUDA as "coalesced memory access", is that one that allows for the best performance [8].

If the matrices have rows with quite a variable number of nonzero elements, the same warp can process rows with very different lengths. In other words, we have threads with little work and threads with a lot of work. According to the SIMT execution model, the warp just finishes when all the threads have terminated. Therefore, we can expect that sorting the rows by their length leads to a more balanced work on GPU, resulting from it a possibly better performance. In the ELL-r format, the use of a vector with a number of nonzero elements per row, as referred before, allows stopping the computation when a thread has not more elements to process, without changing the memory access pattern (even if one or more threads do not access the memory, the scheme of contiguous access is not broken).

B. Warp per Row and Thread per Row Algorithms Using the CSR Format

With the CSR format, a good performance can be obtained using the WpR (warp per row) algorithm, where all warp threads process the same matrix row using a parallel reducing algorithm [2]. In this case, the execution model in the several multiprocessors of a GPU is SPMD (single program multiple data), which means that a new warp is launched when the execution of a warp terminates, and this is so regardless of what happens in other multiprocessors. Concerning the memory access, the WpR algorithm is more efficient than the TpR algorithm, because it accesses data and indices contiguously (data is stored in row major order and all threads access elements from the same row). This does not happen in the TpR algorithm with the CSR format because the contiguous elements are not accessed simultaneously by all threads.

The main drawback of the WpR algorithm is that it needs to have enough work for each warp, which requires that the matrix rows have a number of nonzero elements greater than the warp size (32). Then, as all the threads of a given warp are processing the elements of the same row, sorting the rows ends up having no impact on performance. In the case of the TpR algorithm with the CSR format, when the matrix has a highly variable number of nonzero values per row, each warp just finishes when the longest row finishes, and thus as with ELL format sorting the rows may improve performance.

V. TESTING RESULTS

A. Experimental Testbed

The host machine used in this work was an Intel Core 2 Quad Q9550 at 2.83 GHz with 4 GB of RAM and running the operating system Microsoft Windows XP Professional 64-bits. We also used two graphics cards, the Nvidia GeForce GTX 295 and the GeForce GTX 590 programmed with CUDA.

The GeForce GTX 295 (base on the GT200 architecture) used with synthetic matrices, has 30 multiprocessors, each one with 8 cores (at 1.24 GHz). This GPU has 1GB of global memory and a "compute capability" of 1.3.

The GeForce GTX 590, used in the last experiments of our work, is equipped with a two full size GF110 Fermi GPU's. Each GPU has 16 multiprocessors, each one with 32 cores (at 1,214 GHz), 1.6 GB of global memory and a "compute capability" of 2.0.

This GPU based on Fermi architecture has a new memory hierarchy in which it is possible to configure different cache levels; unlike previous architectures different kernels can execute concurrently; a new dual warp scheduler is introduced; now each multiprocessor has two warp schedulers and two instruction dispatch units, allowing two warps to be issued and executed concurrently [13].

B. Thread per Row versus Warp per Row Algorithms on Synthetic Matrices

We used real-valued square matrices to evaluate the performance of the SpMV product. For that purpose, we randomly generated single precision sparse matrices of order 4096, 8192 and 16384. More specifically, about 1% to 20% of nonzero elements were generated for each matrix row. Also, the column index of each nonzero value was obtained randomly. Doing so, each matrix ended up having about 10% of nonzero values. However, randomly generated matrices have a high variability in respect to row length, assuming here that the length of a row is the number of its nonzero elements. Complete results of these experiments can be seen in [22]. With respect to the TpR algorithm, we compared the kernel execution times for both formats, with and without sorting of rows, considering three block sizes: 32 (1 warp), 64 and 128 threads. We have concluded that row sorting in the CSR format can give a small advantage (about 12% in the best case of having the smaller matrix order and the block size of 128).

For the ELL format, we used the ELL-r version and the results are much better than for CSR format. For the studied matrices, the gain obtained with row sorting can

be about 30% (to the matrix of order 16384 and a block size of 32). Increasing the block size causes in some cases a very slight increase in the performance, but because this is not a general rule the following experiments were carried out using a block size of 32.

Table 1 shows the five testing setups involving CSR and ELL-r formats, with and without row sorting. The first four setups made usage of the TpR algorithm, while the last setup used the WpR algorithm. As the results show, this last setup consisting of the CSR format plus WpR algorithm surpasses the other setups in terms of timing performance.

Considering that in many applications the matrices have higher percentages of rows with small length, we generated matrices with the same structure as before, but decreasing the percentage of nonzero values. The results of the SpMV product for the matrix with order of 16384, using a block size of 32, are shown in Table 2. The execution times obtained in CPU are also presented as a baseline that allows getting an insight of the advantage of using the GPU. The CSR format is here considered because it is much faster than the ELL format in CPU.

As can be seen, when decreasing the percentage of nonzero values, the "ELL-r sorted" plus TpR surpasses the WpR algorithm in performance when such a percentage goes down to 2%. The column GPUsor/GPU*100 indicates the percentage of the execution time of the product with row sorting in relation to the execution time of the same operation without row sorting.

Table 1. Best execution times (in milliseconds) and the corresponding block sizes for each GPU case studied applied to synthetic matrices with 10% of nonzero values

Matrix order	GPU, CSR (Thread per Row)				GPU, ELL-r (Thread per Row)				GPU, CSR	
			row sorted				row sorted		Warp per Row	
	time (ms)	block size	time	block size	time (ms)	block size	time (ms)	block size	time (ms)	block size
4096	4.24	32	3.72	128	1.40	32	1.23	32	0.83	32
8192	20.22	128	18.80	64	4.64	64	3.89	64	2.98	32
16384	93.37	128	88.14	128	17.55	128	14.11	32	11.45	32

Table 2. Execution times (in milliseconds) when decreasing the percentage of nonzero values (synthetic matrix of order 16384, block size = 32)

% of Nonzero	CPU CSR (ms)	GPU, ELL-r, Thread per Row			GPU, CSR WpR (ms)	Maximum row length
		Without row sorting (ms)	With row sorting (ms)	GPUsor/GPU*100		
5%	45.48	9.77	7.11	72.8%	6.77	1638
2%	18.50	3.98	2.90	72.9%	3.06	655
1%	9.39	2.07	1.58	76.3%	1.67	327
0.1%	1.23	0.32	0.27	90.0%	0.47	32

Table 3. Execution times (in milliseconds) when increasing the percentage of rows with length <= 32 (synthetic matrix of order = 16384, block size = 32)

% of row length <= 32	% of nonzero elements	CPU	GPU		
		CSR (ms)	ELL-r TpR (ms)	ELL_R sorted TpR (ms)	CSR WpR (ms)
10%	4.5%	42.03	9.22	6.46	6.35
30%	3.6%	32.74	8.32	5.25	5.22
50%	2.6%	23.35	7.30	3.40	4.31
70%	1.6%	14.64	6.21	2.74	3.63
90%	0.6%	5.77	5.09	1.94	3.11
100%	0.1%	1.25	0.32	0.29	0.66

Sorting the rows is less advantageous when the percentage of nonzero elements and the maximum row length decreases, but the results point to that for matrices with a small percentage of nonzero values the best algorithm is the "one thread per row" for the ELL-r format with row sorting.

Furthermore, we studied what happens when increasing the percentage of rows with length smaller than the warp size. Table 3 shows the results for the matrix of order 16384, where a percentage of rows have a random length smaller than the warp size (32) and the others have a length between 32 and 10% of the matrix order. Again, we considered a block size of 32, and the execution times are in milliseconds. As can be seen, when the percentage of rows with length smaller than 32 reaches the 70%, the ELL-r TpR algorithm with sorting surpasses the WpR algorithm in performance. Again, when the percentage of nonzero values is below the 2%, the ELL-r TpR algorithm with sorting becomes faster than the WpR algorithm.

In summary, we conclude that when the percentage of nonzero values decreases, the ELL-r TpR algorithm with sorting performs better than the CSR WpR. The same happens when the percentage of rows with length smaller than 32 increases.

C. Benchmarking Matrices using the GTX 295 GPU

We have also studied the impact of sorting the rows for the matrices used in the studies [2], [3], [17], [18] which were originally used in the multi-core benchmarking study of Williams et al. [20]. From this benchmark we did not consider four matrices: "dense", "QCD", "webbase" and "LP". In the first two matrices all the rows have the same length, thus sorting the rows is not applicable. With the last two matrices we had memory restrictions to implement the ELL format without partitioning the matrix. When studying all the remaining 10 matrices, we also used a block of 32 threads and the average execution times (in milliseconds) presented in Table 4 were obtained with 500 runs of each kernel in the same GPU as in the previous experiments, the GTX 295.

In this table we can compare the CPU execution times (CSR format) with the GPU execution times obtained using six setups. The first four GPU setups use global memory on GPU side, while the last two use GPU texture memory to access the vector X. Besides, the first three and the last two setups make usage of the ELL formats together with the TpR algorithm to accomplish, while the fourth setup tries to take advantage of the CSR format and of the WpR algorithm. As can be seen, row sorting only has advantage in three cases (Economics, Circuit and Harbor). For the other matrices, the execution time after sorting the rows is worse than before.

To better understand the results we have analysed the longest row of each warp before and after sorting the rows. We will call LRW (longest row of the warp) to the size of the longest row processed by the warp. Especially in the Economics and Circuit matrices after sorting, the number of warps with a small LRW grows significantly (decreasing the number of warps with big LRW). This happens in the matrices with high row length variability inside each warp. After sorting, small rows were not mixed with big rows. Figure 2 shows a graph with the number of nonzero elements per row to the Circuit matrix, which is the one that presents the biggest advantage when sorting the rows. As can be observed in the Circuit matrix, there is high variability of row lengths. Figure 3 shows the graph for the Wind Tunnel matrix. Here the variability of row lengths is clearly smaller.

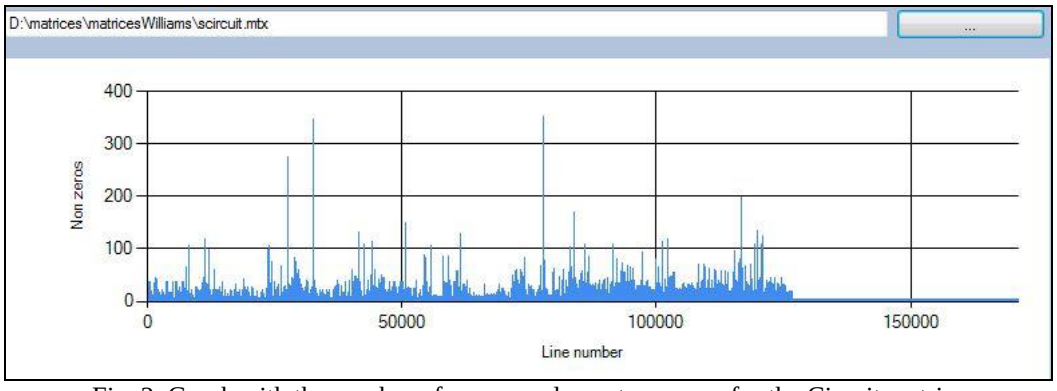

Fig. 2. Graph with the number of nonzero elements per row for the Circuit matrix.

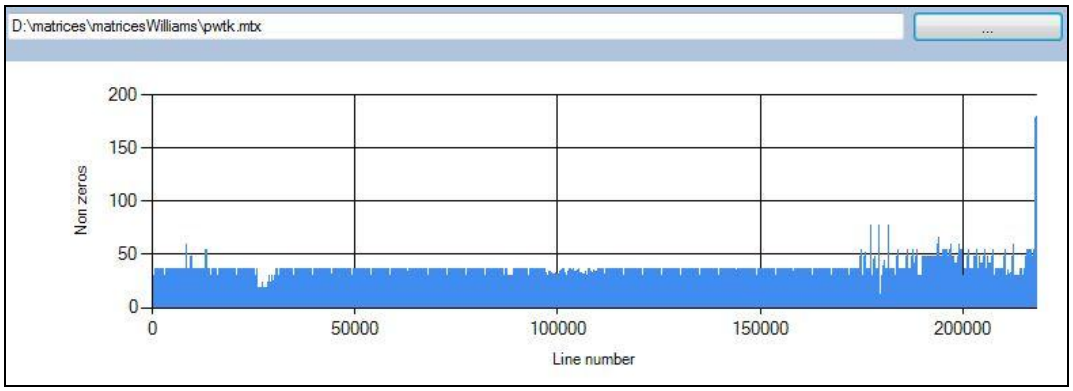

Fig. 3. Graph with the number of nonzero elements per row for the Wind Tunnel matrix.

Studying the memory accesses, we have verified that the difference in the execution times are mainly due to the access to the vector X. Table 5 shows the following for each matrix: characteristics of the matrix (the number of nonzero elements – nnz, the percentage of nonzero elements and the average number of nonzero elements per row); the time (in milliseconds) spent in sorting the rows (via the qsort C/C++ function); the performance in GFLOPS for the TpR algorithm when using the formats ELL-r and for ELL-r sorted when using the texture memory. It was calculated by 2*(number of nonzero values)/time required for one SpMV product. Concerning

the time needed to sort, it should be noted that usually real applications involve a great number of iterations with the rows being sorted just once in the beginning of the process. The last columns of Table 5 present the average of the LRWs before and after sorting. The last column is the percentage ratio of the average of the LRWs after sorting to the average before sorting. As can be seen, we get an advantage in sorting when that ratio is 66% or less. Computing this ratio for previous synthetic matrices, with 10% of nonzero elements, we get a value of 52.2%.

Analysing the LRWs before and after sorting, the results point to that with this GPU (GTX 295) we have advantage in sorting rows when the average of the LRWs decreases to about 66% or less in relation to the value before sorting.

Table 4. Execution times (in milliseconds) for matrices of Williams multi-core benchmarking [20] with a bock size of 32 running in a GTX 295

Matrix	Order	CPU	GPU				GPU texture	
		CSR (ms)	TpR ELL std (ms)	TpR ELL-r (ms)	TpR ELL-r sorted (ms)	WpR CSR (ms)	TpR ELL-r (ms)	TpR ELL-r sorted (ms)
Economics	206500	5.24	2.10	1.60	0.945	3.81	1.21	0.708
Accelerator	121192	5.27	0.825	0.665	0.830	2.25	0.504	0.685
Cantilever	62451	7.10	0.867	0.751	0.976	1.35	0.588	0.622
Epidemiology	525825	7.02	0.776	0.745	0.753	7.54	0.685	0.687
Protein	36417	7.76	1.658	1.08	1.36	1.26	0.787	0.857
Spheres	83334	10.12	1.41	1.02	1.45	1.98	0.827	0.915
Ship	140874	14.35	2.26	1.54	2.84	2.87	1.23	2.13
Wind Tunnel	217918	20.60	6.56	2.01	3.56	4.27	1.55	2.14
Circuit	170998	4.619	9.90	2.11	0.919	3.07	1.78	0.793
Harbor	46835	4.578	1.69	1.27	1.03	1.41	0.811	0.614

Table 5. Features, sorting time, GFLOPS and average of warp lengths of Williams benchmarking matrices [20].

Matrix	Nº of nonzero elements (nnz)	% of nnz	Av. of nnz per row	Sorting Time (ms)	GFlops TpR ELL-r (Texture)	GFlops TpR ELL-r Sorted (Texture)	Average of LRWs		
							Before Sorting (1)	After Sorting (2)	Ratio (2) / (1)
Economics	1273389	0.003	6	12.2	2.1	3.6	22.4	6.18	27%
Accelerator	1362087	0.009	22	7.3	5.4	4.0	16.1	11.2	70%
Cantilever	2034917	0.050	65	3.9	6.9	6.5	37.6	32.6	87%
Epidemiology	2100225	0.0003	4	9.8	6.1	6.1	3.99	3.99	100%
Protein	2190591	0.165	119	3.6	5.6	5.1	90.3	60.3	67%
Spheres	3046907	0.044	72	5.0	7.4	6.7	42.2	36.6	87%
Ship	3977139	0.015	28	10.5	6.5	3.7	40.0	28.3	71%
Wind Tunnel	5926171	0.012	53	5.3	7.6	5.5	31.0	27.2	88%
Circuit	958936	0.003	6	9.0	1.1	2.4	14.1	5.66	40%
Harbor	2374001	0.110	50	3.5	5.8	7.7	76.5	50.7	66%

D. Benchmarking Matrices using the GTX 590 GPU

Finally we run the same code on the Fermi GPU (the GTX 590) for the same benchmarking matrices using just the global memory and the ELL format. The TpR algorithm is applied to the ELL standard format and to ELL-R with and without sorting. In Table 6 the execution time for each matrix when using a block size of 32 is presented. Comparing these results with the ones obtained with the GTX 295 card (Table 4) it can be seen that sorting the rows still having advantage for the matrices "Economics" and "Harbor", but now the difference to the case without sorting is quite smaller. For all the other matrices sorting the rows has no advantage.

For this card we also consider block sizes of 64, 128, 256 and 512. Table 7 presents just the best execution time obtained for each matrix and the corresponding block size. As can be seen, for all the studied matrices, the best execution time of the SpMV product is obtained with the ELL-R format without sorting.

In the Fermi architecture, each multiprocessor has 32 cores, a fourfold increase over the GT200 architecture, and allows two warps to be executed concurrently. Then considering blocks with just 32 threads in Fermi card is a wasting of resources and justifies the similar results in both GPUs when using a block of 32 threads. Increasing the block size reduces significantly the execution time in all the cases and sorting the rows in the benchmarking matrices does not present any advantage. As referred before the original positions of the rows present a better memory access pattern in accessing the correspondent vector values.

To measure the performance gain with sorting, independently of the cost in accessing the vector X, we changed the previous kernel in order to access sequentially the vector X, instead of accessing the right positions. With this new kernel that accesses the initial positions of X, a wrong result vector is produced. However it shows the net gain obtained when the work of each warp is balanced across its threads.

Table 8 shows the execution times for Williams's benchmark in a GTX 590 GPU when using this kernel for the TpR algorithm applied to ELL-R format with and without sorting. We consider block sizes of 32, 64, 128, 256, 384 and 512. In all the matrices and for all the block sizes, the best execution times when accessing sequentially the vector X were obtained with the version that sorts the matrix rows. Furthermore for all the matrices, the best execution time is obtained with a block

size of 256. That value (that is, 8 times 32) could be expected considering that this GPU, as was referred before, has four more cores than the GTX 295 and launches two warps at a time. In table 8, just the execution times for this block size are presented; column three presents the execution times before sorting and column four the execution times after sorting. The last column of table 8 presents the percentage ratio of the execution time after sorting to the execution time before sorting. As can be seen from Table 8, in all the cases it is advantageous to sort the rows. The biggest advantage is for the Economics matrix with a value of 59.3% (obtained from 100 minus 40.7) and for the Epidemiology matrix the advantage is almost non-existent. In this last case, most of the matrix rows have the same length.

Table 6. Execution times (in milliseconds) for matrices of Williams multi-core benchmarking [20] with a bock size of 32 running in a GTX 590 GPU

Matrix	Block size	GPU		
		TpR ELL standard (ms)	TpR ELL-r (ms)	TpR ELL-r Sorted (ms)
Economics	32	2.14	0.986	0.588
Accelerator	32	0.866	0.502	0.662
Cantilever	32	0.780	0.452	0.592
Epidemiology	32	0.779	0.571	0.575
Protein	32	1.57	0.667	1.13
Spheres	32	1.41	0.690	0.927
Ship	32	2.40	1.09	1.79
Wind Tunnel	32	7.00	1.35	1.93
Circuit	32	9.26	0.596	0.697
Harbor	32	1.66	0.732	0.707

Table 7. Execution times (in milliseconds) for matrices of Williams multi-core benchmarking [20] in a GTX 590 GPU using the block size that produces the best result for each matrix

Matrix	Block size	GPU		
		TpR ELL standard (ms)	TpR ELL-r (ms)	TpR ELL-r Sorted (ms)
Economics	256	0,669	0.415	0.598
Accelerator	256	0.407	0.346	0.655
Cantilever	512	0.257	0.224	0.507
Epidemiology	256	0.241	0.234	0.238
Protein	512	0.626	0.391	1.18
Spheres	128	0.533	0.348	0.792
Ship	512	0.732	0.534	1.92
Wind Tunnel	512	1.79	0.701	1.77
Circuit	128	2.90	0.428	0.635
Harbor	128	0.595	0.369	0.581

Table 8. Execution times (in milliseconds) for matrices of Williams multi-core benchmarking [20] in a GTX 590 GPU when sequentially accessing the vector X, for the block size that produces the best result for each matrix.

Matrix	Block size with the best result	GPU		
		TpR, ELL-r (ms) (1)	TpR ELL-r Sorted (ms) (2)	Ratio (2) / (1)
Economics	256	0.243	0.099	40.7%
Accelerator	256	0.111	0.082	73.9%
Cantilever	256	0.126	0.110	87.3%
Epidemiology	256	0.152	0.151	99.3%
Protein	256	0.210	0.171	81.4%
Spheres	256	0.186	0.168	90.3%
Ship	256	0.293	0.220	75.1%
Wind Tunnel	256	0.417	0.367	88.0%
Circuit	256	0.271	0.251	92.6%
Harbor	256	0.210	0.160	76.2%

We can conclude that for the new GPU architectures, with high processing power sorting the rows by its length has no advantage regarding time performance purposes. We should note that GTX 590 card is now a mid range GPU. The gain in sorting the rows by its length is surpassed by the loss in accessing the values of vector X by the new order. To take advantage in ordering the rows we need to search by an algorithm where the gain in computation surpass the lost in accessing the vector X.

VI. DISCUSSION

In this work we have explored the impact of sorting the rows of the sparse matrix (A) by their length when computing the SpMV product (A×X) in GPU for matrices in ELL-r and CSR formats using the TpR algorithm. The results were compared with the execution times obtained with the WpR algorithm with the CSR format.

To assess the impact on performance of row sorting, two sets of matrices were used. The first set is randomly generated (synthetic matrices) and the second set is formed by the benchmarking matrices proposed in [20]. The rows of each matrix were pre-processed in CPU, using the quick sort algorithm (C/C++ qsort function) in such a way that they become stored in memory by order of increasing length. To guarantee the result consistency we have created an additional array with the original row's positions.

The first part of the work was comparing the two algorithms that have shown to have better results in previous studies [2], [17] the TpR and the WpR algorithms when using synthetic matrices and a Nvidia GTX 295 GPU. The TpR algorithm was studied for the storage formats CSR and ELL-r with and without sorting the matrix rows. The execution times of these cases were compared with the WpR algorithm to the CSR format. Although sorting the rows is not applicable to the WpR algorithm it was also studied to compare the execution times and understand in what situations this algorithm is faster than the TpR algorithm.

Initially, synthetic matrices with 10% of nonzero float elements were used. However, for many real applications the sparse matrices present a percentage of nonzero elements much lower than 10%. Furthermore, in [2] is shown that for matrices where most of the rows contain a number of nonzero elements greater than the warp size, the algorithm of WpR to the CSR format, has the best results. Therefore the study continues by comparing the behaviour of the same algorithms firstly when decreasing the percentage of nonzero elements secondly when varying the percentage of rows with length smaller than the warp size (32). Now, in both situations we used just the ELL-r format to study the impact of row sorting and compare the results with the WpR + CSR algorithm. That decision was made because the TpR algorithm for the CSR format has shown worse performance than for the ELL-r format. More than that, it was concluded that the impact of row sorting in the CSR format is significantly smaller than in the ELL-r format.

In the second part of this study, we tested the approach of sorting the rows with most of the matrices used in [2,17]. For these benchmarking matrices we used two GPU's: a second generation Nvidia GPU, the GeForce GTX 295 and the GeForce GTX 590 based on Fermi, a more advanced Nvidia GPU architecture. To be able to compare with the execution times obtained with these matrices in previous works we have also used the TpR algorithm for the standard ELL format. We have adapted the kernels from [2]. Another variant studied was the use of the texture memory. Besides the case of storing the matrix and the vector in global memory as was done for synthetic matrices, for benchmarking matrices we also test the use of the texture memory to store the vector X. Using the texture memory improves the performance but does not change the trend of the results.

From this last study with benchmarking matrices we have concluded that, when using the GTX 295 reordering the rows is worthwhile to less than one third of the tested matrices and even that small advantage disappears when using a more advanced GPU such as the GTX 590. For this last card we have also measured the performance gain with sorting, independently of the cost in accessing the vector X. The results, from accessing sequentially the initial positions of X, show that sorting the rows is always advantageous. Thus to be able to a-priori decide if row sorting is worthwhile we also need to measure the lost of locality when accessing the values of X.

VII. CONCLUSION

We have studied the impact of sorting the matrix rows on the performance of the SpMV product for two storage formats, CSR and ELL-r, and we have compared the behaviour of different algorithms when varying the sparse structure of the matrices.

For matrices where the percentage of rows with length greater than the warp size is high, the best algorithm is the "one warp per row" to the CSR format, and in that case, sorting the rows is useless. Otherwise, for matrices with a high percentage of small rows, the algorithm of "one thread per row" with the ELL-r format is the best. In that case we have shown that for synthetic matrices with high row length variability, sorting the matrix rows by their length is advantageous. However, the study of a set of benchmarking matrices representing real-world applications has shown that when using the second generation Nvidia GTX 295 GPU just for less than a third of the matrices sorting the rows is advantageous.

Moreover, when using a more recent GPU architecture, the GTX 590 from Nvidia, the best performance is obtained without sorting the rows. Thus we can conclude that using current GPU's, for the studied matrices, the advantage obtained through sorting the rows by its length is overcome by the loss in performance caused by the additional time needed to access the input vector. Although the rows of a warp may have almost the same number of nonzero entries after row sorting, their positions in each row can be quite distant. In spite of sorting the rows can be useful for the purpose of saving memory it should be taken into account that sorting the rows by their length brings some penalization in most of

the cases.

As future work we will explore other approaches, namely based on metaheuristics, to find the best ordering that maximizes the load balancing in the scheduled warps and simultaneously minimizes the cost of memory accesses.

REFERENCES

[1] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and H. S. Qintana-Ortí, "Solving dense linear systems on graphics processors," Lecture Notes in Computing Science vol. 5168 (2008) pp 739-748.

[2] N. Bell, and M. Garland, "Implementing Sparse Matrix-Vector Multiplication," on Throughput-Oriented Processors, Proc. Supercomputing '09, 2009.

[3] N. Bell, and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," Technical Report NVR-2008-004, NVIDIA Corporation, Dec., 2008.

[4] B. Boyer, J. Dumas, and P. Giorgi, "Exact Sparse Matrix-Vector Multiplication on GPU's and Multicore Architectures," Proc. of the 4th Int'l Workshop on Parallel and Symbolic Computation (PASCO '10), Marc Moreno Maza and Jean-Louis Roch (Eds.). ACM, New York, NY, USA, pp. 80-88, 2010. DOI=10.1145/1837210.1837224.

[5] J.W. Choi, A. Singh, and R.W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," Proc. of the 15th ACM SIGPLAN, Symp. on Principles and practice of parallel programming (PPoPP '10), pp. 115-126, 2010.

[6] D. Grewe, and A. Lokhmotov, "Automatically generating and tuning GPU code for sparse matrix-vector multiplication from a high-level representation," Proc. of the Fourth Workshop on General Purpose Processing on GPU, ACM, Article 12, 8 pages, 2011.

[7] D.R. Kincaid, T.C. Oppe and D.M. Young, "ITPACKV 2D User's Guide," Report CNA-232, University of Texas at Austin, Centre for Numerical Analysis, May, 1989.

[8] D.B. Kirk, and W.W Hwu, "Programming Massively Parallel Processors," Morgan Kaufmann, Elsevier, 2010.

[9] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, A. Basermann, and A.R. Bishop, "Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation," in CoRR, abs/1112.5588, 2011. Available at: http://arxiv.org/abs/1112.5588.

[10] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures", Lecture Notes in Computer Science, Vol. 5952(2010) pp. 111-125.

[11] R.L. Naff, and E.R. Banta, "The U.S. Geological Survey modular ground-water model - PCGN: A preconditioned conjugate gradient solver with improved nonlinear control', in U.S." Geological Survey, Open-File Report 2008-1331, 2008. Available at: http://pubs.usgs.gov/of/2008/1331/pdf/OF08-1331_508.pdf.

[12] NVIDIA Corporation, "NVIDIA CUDA Programming guide," version 4.0, 2011.

[13] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2009. Available at: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[14] Y. Saad, "Iterative Methods for Sparse Linear Systems," 2nd Edition, SIAM, 2003.

[15] B. Schmidt, H. Aribowo, and H. Dang, "Iterative sparse Matrix-Vector multiplication for integer factorization on GPUs," Proc. of the 17th Int.l Conf. on Parallel processing - Volume Part II (Euro-Par'11), Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.) pp. 413-424, 2011.

[16] S. Thomas, "Preconditioned Conjugate Gradient Methods for Semiconductor Device Simulation on a CRAY C90 Vector Processor," Proc. of the Second International Conference on Vector and Parallel Processing (VECPAR '96), José M. L. M. Palma and Jack Dongarra (Eds.). Springer-Verlag, London, UK, pp. 154-167, 1996.

[17] F. Vázquez, J.J. Fernández and E.M. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs," Concurrency and Computation Practice and Experience, Vol. 23(2011) pp. 815–826. doi: 10.1002/cpe.1658v.

[18] F. Vázquez, E.M. Garzón, J.A. Martinez, and J.J. Fernández, "Accelerating sparse matrix vector product with GPUs," Proc. of the 2009 Int'l Conf. on Computational and Mathematical Methods in Science and Engineering, CMMSE, vol. 2, pp. 1081-1092, 2009.

[19] V. Volkov, and J.W. Demmel, Benchmarking, "GPUs to tune dense linear algebra," Proc. of the 2008 ACM/IEEE Conf. on Supercomputing, IEEE Press, USA, pp. 1-11, 2008.

[20] S. Williams, L. Olivier, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," Proc. of 2007 ACM/IEEE Conference on Supercomputing, 2007.

[21] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi and A. Moshovos, "Demystifying GPU Microarchitecture through Microbenchmarking," IEEE Int'l Symp. On Performance Analysis of Systems & Software, pp. 236-246, March 2010.

[22] P. Prata, G. Melfe, R. Pesqueira and J. Muranho, "Impacto da Organização dos Dados em Operações com Matrizes Esparsas na GPU," in Portuguese, INForum 2010- II Simpósio de Informática, Luís S. Barbosa, Miguel P. Correia (eds), pp. 255–266, , September 2010.