2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)

Improving the performance of the sparse matrix vector product with GPUs

F. Vázquez, G. Ortega, J.J. Fernández, E.M. Garzón
Dpt Computer Architecture. Almeria University
Cra Sacramento s/n Almeria 04120 Spain
{f.vazquez, gol315, jjfdez, gmartin}@ual.es

***Abstract*—Sparse matrices are involved in linear systems, eigensystems and partial differential equations from a wide spectrum of scientific and engineering disciplines. Hence, sparse matrix vector product (SpMV) is considered as key operation in engineering and scientific computing. For these applications the optimization of the sparse matrix vector product (SpMV) is very relevant. However, the irregular computation involved in SpMV prevents the optimum exploitation of computational architectures when the sparse matrices are very large. Graphics Processing Units (GPUs) have recently emerged as platforms that yield outstanding acceleration factors. SpMV implementations for GPUs have already appeared on the scene. This work proposes and evaluates new implementations of SpMV for GPUs called ELLR-T. They are based on the format ELLPACK-R, which allows storage of the sparse matrix in a regular manner. A comparative evaluation against a variety of storage formats previously proposed has been carried out based on a representative set of test matrices. The results show that: (1) the SpMV is highly accelerated with GPUs; (2) the performance strongly depends on the specific pattern of the matrix; and (3) the implementations ELLR-T achieve higher overall performance. Consequently, the new implementations of SpMV, ELLR-T, described in this paper can help to exploit the GPUs, because, they achieve high performance and they can be easily joined in the engineering and scientific computing.**

I. Introduction

The Matrix-Vector product (MV) is a key operation for a wide variety of scientific applications, such as image processing, simulation, control engineering and so on [14]. For many applications based on MV, the matrix is large and sparse, i.e. the dimensions of matrix are large ($\geq 10^5$) and the percentage of non-zero components is very low ($\leq 1 - 2\%$). Sparse matrices are involved in linear systems, eigensystems and partial differential equations from a wide spectrum of scientific and engineering disciplines. For these problems the optimization of the sparse matrix vector product (SpMV) is a challenge because of the irregular computation of large sparse operations. This irregularity arises from the fact that the data access locality is not maintained and that fine grained parallelism of loops is not exploited [6]. Therefore, additional effort must be spent to accelerate the computation of SpMV. This effort is focused on the design of appropriate data formats to store the sparse matrices, since the performance of SpMV is directly related to the used format as shown [12], [13], [15].

Currently, Graphics Processing Units (GPUs) offer massive parallelism for scientific computations. The use of GPUs for general purpose applications has exceptionally increased in the last few years thanks to the availability of Application Programming Interfaces (APIs), such as Compute Unified Device Architecture (CUDA) [11] and OpenCL [9], that greatly facilitate the development of applications targeted at GPUs. Recently, several implementations of SpMV have been developed with CUDA and evaluated on GPUs [1], [3]–[5], [10]. Devising GPU-friendly matrix storage formats has been key in these implementations.

This work aims at presenting and evaluating a new approach to increase the performance of SpMV on GPUS which relies on a new storage format for the sparse matrix, ELLPACK-R. This format is a GPU-friendly variant of one previously designed for vector architectures, ELLPACK [7]. An extensive performance evaluation of this new approach has been carried out based on a representative set of test matrices. The comparative study has drawn the conclusion that the implementations ELLR-T based on ELLPACK-R proves to outperform the most common and efficient formats for SpMV on GPUs used so far.

Next, Section II summarizes the aspects related to GPU programming and computing. Then, Section III reviews the different formats to compress sparse matrices, given that the selection of an appropriate format is the key to optimize SpMV on GPUs. Section IV introduces the proposed format and algorithms for computation of SpMV on GPUs. In Section V the performance measured on a NVIDIA Geforce GTX 285 with a set of representative sparse matrices belonging to diverse applications is presented. The results clearly show that the new algorithms for computation of SpMV with GPUs presented here, ELLR-T, get the best performance for all the test matrices. Finally, Section VI summarizes the main conclusions.

II. Computational keys to exploit GPUs

From a programmer's point of view, the GPU is considered as a set of SIMD (Single Instruction stream, Multiple Data streams) multiprocessors. Each kernel (parallel code) is executed as a batch of threads organized as a grid of thread blocks. For the execution, each block is assigned to a Streaming Multiprocessor (SM) composed by eight cores called Scalar Processors (SP). The blocks in turn are divided into sets of 32 threads called warps.

978-0-7695-4108-2/10 $26.00 © 2010 IEEE
DOI 10.1109/CIT.2010.208

Each SM has an on-chip memory area containing a set of 32-bit registers and a low latency memory shared between all threads belonging to the block called shared memory. There is also an off-chip memory area consisting of a larger size and latency memory known as device memory. It is addressed by all threads declared for the execution, and, moreover, there are two low latency cache memories called constant memory and texture memory. The device memory access is performed by groups of 16 threads called half-warps. If the access pattern of the different threads belonging to every warp verifies the coalescence conditions, then, it can be performed in parallel by all of them and the memory latency would be the same as that of a single access. On the modern GPUs, coalescing is achieved by any pattern of accesses that fits into a segment size of 32, 64 or 128 bytes for 8, 16 or 32 and 64-bit words respectively. Memory segments must be aligned to 16 and 32 memory words in order to reduce the number of memory accesses. The use of texture memory improves the performance when the searched word is located within it.

The ratio between the number of active warps per multiprocessor and the maximum number of active warps is called the multiprocessor occupancy. The occupancy determines how effectively the hardware is kept busy with the goal of hide latencies, by switching between active warps, due to memory operations and paused warps. Occupancy is closely related to the thread block size (BS) and the number of registers and shared memory size used by a kernel. Therefore a good choice of BS will improve the performance.

On the other hand, the maximum instruction throughput is achieved when all threads of the same warp execute the same instruction sequence, given that any flow control instruction can cause the threads of the same warp to diverge, that is, to follow different execution paths that will be serialized.

Specifically, to optimize SpMV on GPUs, these goals have to be taken into account when devising appropriate formats to store the sparse matrix since the parallel computation and the memory access are tightly related to the storage format of the sparse matrix.

III. An overview of SpMV and its challenges. Formats to compress sparse matrices.

The pattern of memory access to read the elements of the sparse matrix has a strong impact in the performance of SpMV. So, every specific algorithm to compute SpMV (i.e. $u = Av$ where A is the sparse matrix, u and v are the output and input vectors respectively) exploiting a particular architecture is related to a specific format to store the sparse matrix. Next, the main formats to compress sparse matrices and their corresponding algorithms are described, focusing on the formats specifically designed for SIMD architectures such as vector architectures and GPUs.

A. Coordinate storage (COO)

The coordinate storage scheme (COO) to compress a sparse matrix is a direct transformation from the dense format. Let Nz be the total number of non-zero entries of the matrix. A typical implementation of COO uses three one-dimensional arrays of size Nz. One array, $A[\,]$ of floating-point numbers (hereafter referred to as floats), contains the non-zero entries. The other two arrays of integer numbers, $I[\,]$ and $J[\,]$, contain the corresponding row and column indices for each non-zero entry. The performance of SpMV based on COO may be penalized because it does not implicitly include the information about the ordering of the coordinates, and, additionally, for multi-threaded implementations of SpMV atomic data access must be included when the elements of the output vector are written.

B. Compressed Row Storage (CRS)

Compressed Row Storage (CRS) is the most extended format to store sparse matrices on superscalar processors. Let N and Nz be the number of rows of the matrix and the total number of non-zero entries of the matrix, respectively; the data structure consists of the following arrays: (1) $A[\,]$ array of floats of dimension Nz, which stores the entries; (2) $J[\,]$ array of integers of dimension Nz, which stores their column index; and (3) $start[\,]$ array of integers of dimension $N+1$, which stores the pointers to the beginning of every row in $A[\,]$ and $J[\,]$, both sorted by row index.

The code to compute SpMV based on CRS has several drawbacks that hamper the optimization of the performance of this code on superscalar architectures. First, the access locality of vector $v[\,]$ is not maintained due to the indirect addressing. Second, the fine grained parallelism is not exploited because the number of iterations of the inner loop is small and variable [6]. Despite these drawbacks, several optimizations have made possible to improve the performance of sparse computation on current processors [8], [15]. In particular, the Intel Math Kernel Library (MKL) improves the performance of sparse BLAS operations, based on CRS, by optimizing the memory management and exploiting the ILP on Intel processors.

C. ELLPACK

ELLPACK or ITPACK [7] was introduced as a format to compress a sparse matrix with the purpose of solving large sparse linear systems with ITPACKV subroutines on vector computers. This format stores the sparse matrix on two arrays, one float $A[\,]$, to save the entries, and one integer $J[\,]$, to save the column index of every entry. Both arrays are of dimension $N \times Max\_nzr$ at least, where N is the number of rows and Max_nzr is the maximum number of non-zeros per row in the matrix, with the maximum being taken over all rows. Note that the size of all rows in these compressed arrays $A[\,]$ and $J[\,]$ is the same, because every row is padded with zeros. Therefore, ELLPACK can be considered as an approach to fit a sparse matrix in a regular data structure similar to a dense matrix. Consequently, this format is appropriate to compute operations with sparse matrices on vector architectures.

Focusing our interest on the GPU architecture and if every element i of vector u is computed by a thread identified by index $x = i$ and the arrays store their elements in

column-major order, then the SpMV based on ELLPACK can improve the performance due to: (1) the coalesced global memory access, thanks to the column-major ordering used to store the matrix elements into the data structures. Then, the thread identified by index x accesses to the elements in the x row: $A[x + k * N]$ with $\{0 \leq k < Max\_nzr\}$ where k is the column index into the new data structures $A[\ ]$ and $J[\ ]$. Consequently, two threads x and $x + 1$ access to consecutive memory address, thereby fulfilling the conditions of coalesced global memory access; (2) non-synchronized execution between different thread blocks. Every thread block can complete its computation without synchronization with others blocks.

However, if the percentage of zeros is high in the ELLPACK data structure and there is a relevant amount of padding zeros, then the performance decreases. This penalty even remains when conditional branches are included to avoid the memory access and arithmetic operations with padding zeros, because to compute every $u[i]$, with $0 \leq i \leq N$, the k-loop must iterate until $k = Max\_nzr$ and the conditional branch is executed in every iteration; so in order to reduce the memory access and activity of arithmetic units, the computation is penalized with $N \times Max\_nzr$ executions of the conditional branch.

D. Recent Proposals for GPUs

Recently, different proposals of kernels to compute SpMV on GPUs have been described and analysed [1], [3]–[5], [10]. They can be classified in two groups according to their relationship with CRS or ELLPACK formats.

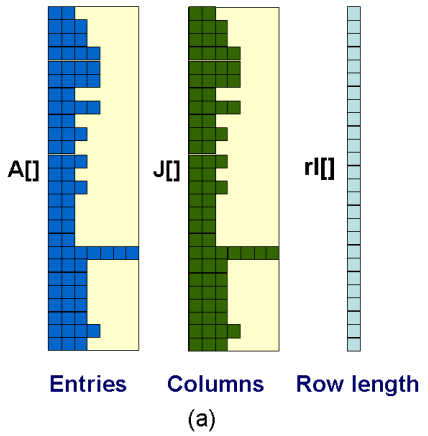
On the one hand, the kernel called CRS(vector) evaluated in [3] is based on CRS format. This kernel computes every output vector element with the collaboration of the 32 threads of every warp. So, one warp computes the float products related to the entries of one row in a cyclic fashion, followed by a parallel reduction in shared memory in order to obtain the final result of output vector element. Then, if the number of elements by row is lower than 32 the performance reached by CRS(vector) will decrease and the best performance will be achieved by matrices of rows with high number of elements. Similarly, another kernel to compute SpMV on GPUs based on CRS format has been recently proposed in [1]. Here the collaboration of 16 threads (half warp) computes every output vector element doing a zero-padding of rows to complete a length multiple of 16, in order to fulfill the memory alignment requirements and improve the coalesced memory access. It has been included on the SpMV4GPU library [2], and hereinafter it will be referred to the same name. Thus, it reaches better performance when the number of elements by rows is lower, but its performance decreases when the rows have a very high number of entries, if compared with CRS(vector), as analyzed in Section V. On the other hand, the kernels related to the format called HYB (which stands for hybrid) proposed by [3] seem to yield the best performance on GPUs so far. This format combines the ELLPACK and COO formats with the goal of improving the performance of ELLPACK. Let A be a sparse matrix stored with CRS format, then a preprocessing step is required to store it with HYB format in order to compute: (1) parameter Max_nzr, (2) distribution function of rows according to their number of entries, (3) subset of a specific percentage of rows with less entries, for example $2/3$ [3], and its corresponding parameter Max_nzr', and, finally, (4) two data structures to store A. Max_nzr' entries of every row are stored in ELLPACK format, and if any entries remain, they are stored with COO format. In other words, HYB stores the sparse matrix with ELLPACK avoiding the elements which overfill some rows and storing them with COO format. So, the corresponding computation of SpMV based on GPU is split in several kernels related to the different formats, hopefully with an appropriate value of Max_nzr' the main kernel related to ELLPACK can reach high performance on GPU, but the kernels related to COO format adds relevant penalties due mainly to un-coalesced memory access and the need to use atomic functions for the write memory operations. This drawback could be relevant especially for any kind of patterns of sparse matrices where the computation of Max_nzr' does not reach optimum value.

Recently, the format called *Sliced ELLPACK* has been proposed and evaluated in [10]. In order to compress the matrix, the N rows of A are partitioned in sets of S rows and every set is stored with ELLPACK format. Moreover, the τ threads into every block collaborate in the computation related to every set of rows. It achieves high performance when a preprocess with reordering of rows is considered and the optimum values of the parameters S and τ are selected. Other format, called BELLPACK, has been proposed in [5], this proposal compresses the sparse matrix by small dense entries blocks. Then, this approach reaches better performance for those sparse matrices with their pattern including small blocks of entries. Both approaches, *Sliced ELLPACK* and BELLPACK, include complex pre-processing of the sparse matrix.

IV. SpMV based on ELLPACK-R

We propose the ELLPACK-R format, a variant of ELLPACK, to further improve the performance reached by ELLPACK on GPUs. ELLPACK-R consists of two arrays, $A[\ ]$ (float) and $J[\ ]$ (integer) of dimension $N \times Max\_nzr$; and, moreover, an additional integer array called $rl[\ ]$ of dimension N (i.e. the number of rows) is included with the purpose of storing the actual length of every row, regardless of the number of the zero elements padded.

According to the mapping of threads in the computation of every row, several implementations of SpMV based on ELLPACK-R can be developed. Thus, when T threads compute the element $u[i]$ accessing to the i-th row, the implementation is referred as ELLR-T. So, the i-th row is split in sets of T elements. Then, in order to compute the element $u[i]$, T threads compute $rl[\ ]$ iterations of the inner loop of SpMV, every thread stores its partial computation in the shared memory. Finally, to generate the value of $u[i]$, one reduction of the T values computed and stored in shared memory has to be included. The value of parameter T can be




```
idx=(blockIdx.x*blockDim.x+threadIdx.x) # Thread index (BS=blockDim.x)
idb = threadIdx.x;     # Thread index into the block of index blockIdx.x
idp = idb % T;         # Thread index into S_i (set of T Threads related to the row of index i)
i = idx / T;           # Row index

if (i < N) {
   svalue=0.0;
   max=ceil(rl[i]/T);
   for(int k = 0;k < max;k++){
        value=A[k*(N'*T)+(i*T)+idp];# N'=new dimension to fulfill the memory alignment requirements
        col=J[k*(N'*T)+(i*T)+idp];
        svalue+=value*v[col];
   }
   shared[idb]=svalue;
   if(idp < 16){ # Reduction on shared memory to compute u[i] (Code for T = 32)
       shared[idb]+=shared[idb+16];
       if (idp < 8) shared[idb]+=shared[idb+8];
       if (idp < 4) shared[idb]+=shared[idb+4];
       if (idp < 2) shared[idb]+=shared[idb+2];
       if(idp == 0) u[i]=shared[idb]+shared[idb+1];
   }
}
```

(b)

Fig. 1. (a) ELLPACK-R Format and (b)ELLR-T code to compute SpMV on GPUs

explored in order to obtain the best performance with every kind of sparse matrices. Figure 1 illustrates the code of ELLR-T algorithm. The algorithms ELLR-T to compute SpMV with GPUs take advantage of: (1) *Coalesced and aligned global memory access*. The access to read the elements of A, J and rl are coalesced and aligned thanks to the column-major ordering used to store the matrix elements and the zeros-padding to complete the length of every row as multiple of 16. Consequently, the highest possible memory bandwidth of GPU is exploited. (2) *Homogeneous computing within the warps*. The threads belonging to one warp do not diverge when executing the kernel to compute SpMV. The code does not include flow instructions that cause serialization in warps since every thread executes the same loop, but with different number of iterations. Every thread stops as soon as its loop finishes, and the remaining threads continue the execution.(3) *Reduction of useless computation and unbalance of the threads of one warp*. Let S_i be the set of T threads which are collaborating on the computation of $u[i]$, the k-loop reaches the maximum value of $k = rl[i]/T \leq Max\_nzr/T$ for specific sets, S_i, into the warp. Then, the run-time of every warp is proportional to maximum element of the sub-vector $rl[i]/T$ related with every warp, and it is not necessary that the k-loop for all threads reaches $k = Max\_nzr/T$, then, there are not useless iterations and the control of loops of this implementation is reduced comparing with SpMV based on ELLPACK. (4) *High occupancy*. High occupancy levels are reached as it will be shown in next section, if optimal value of threads block size (BS) is used.

V. Evaluation

A comparative analysis of the performance of different kernels to compute SpMV on GPUs has been carried out in this work. The SpMV computations with GPU based on the following formats to store the matrix have been evaluated: CRS, CRS(vector), SpMV4GPU, ELLPACK, HYB and ELLR-T* where $\star$ denotes that ELLR-T is evaluated for optimum values of T and BS.

This analysis is based on the run-times measured on a GeForce GTX 285 with a set of test sparse matrices from different disciplines of science and engineering. Table I summarizes the test matrices used in this work and the characteristic parameters related to their specific pattern: number of rows (N), total number of non-zeros elements (*Entries*), average number of entries per row (Av), the difference between the maximum number of entries in a row and Av (IAv), percentage of relative standard deviation of entries by row($\frac{\sigma}{Av}$). Moreover Table I shows the bandwidth (BW) and speed-up (sp) reached with ELLR-T* on the GPU GeForce GTX 285. The values of these parameters are key to justify the differences between the performance achieved by SpMV with the different formats, which are primarily related to the variability or dispersion of the number of entries by row of the matrices. All matrices are real of dimensions $N \times N$.

The programming interface, CUDA, allows the programmer to specify which variables are to be stored in the texture cache within the memory hierarchy [11]. Here, the vector v has been stored binding to the texture memory for all kernels evaluated, since in the computation of $u = Av$ only the vector v is reused throughout the products with the different rows of the matrix.

The evaluation results show that the best average performance is got by ELLR-1 followed by HYB and ELLPACK, and the worst average performance is obtained by CRS, CRS(vector) and SpMV4GPU. However, the performance of ELLR-T can be highly increased if the values of two parameters are appropriately selected: the thread block size, BS, before mentioned in Section II, and T recently mentioned. The possible values of BS are the powers of two from 16 to 512. The experimental results have shown that only for $BS = 128, 256, 512$ the kernels ELLR-T reach 100% occupancy of GPU, and for $BS = 16, 32, 64$ the occupancy is equal to or less than 50%. Then, it is predictable that the

TABLE I
SET OF TEST MATRICES, THEIR CHARACTERISTIC PARAMETERS RELATED TO THEIR DISTRIBUTION OF ENTRIES ON THE ROWS; EFFECTIVE BANDWIDTH OF MEMORY ACCESS (BW) AND NET SPEED-UP (sp) OF SPMV WITH ELLR-T* ON THE GPU GEFORCE GTX 285

Matrix	*N*	*Entries*	Av	IAv	$\frac{\sigma}{Av}$	*BW*	*sp*
qh1484	1484	6110	4	9	39	6.	1.
dw2048	2048	10114	5	3	10	11.	2.
rbs480a	480	17087	36	0	1	14.	1.
gemat12	4929	33111	7	37	45	20.	4.
dw8192	8192	41746	5	3	12	35.	5.
mhd3200a	3200	68026	21	12	27	41.	4.
e20r4000	4241	131556	31	31	50	53.	5.
bcsstk24	3562	159910	45	12	26	70.	14.
mac_econ	206500	1273389	6	38	72	38.	8.
qcd5_4	49152	1916928	39	1	0	120.	16.
mc2depi	525825	2100225	4	0	2	118.	23.
rma10	46835	2374001	51	94	56	99.	14.
cop20k_A	121192	2624331	22	59	64	70.	21.
wbp128	16384	3933095	240	16	14	111.	14.
dense2	2000	4000000	2000	0	0	121.	15.
cant	62451	4007383	64	14	21	121.	32.
pdb1HYS	36417	4344765	119	85	27	121.	32.
consph	83334	6010480	72	9	26	120.	33.
shipsec1	140874	7813404	55	46	20	122.	32.
pwtk	217918	11634424	53	128	9	128.	33.
wbp256	65536	31413932	479	33	15	95.	13.

performance decreases with the smaller values of BS. On the other hand, focussing our interest on the other parameter T, the values of T are divisors of BS, then $T = 2^l < BS$; our experimental results have shown that ELLR-T does not reach the highest performance for $T \geq 16$. Then, the kernel ELLR-T can achieve better performance if $BS = 128, 256, 512$ and $T = 1, 2, 4, 8$.

Consequently, in order to optimize the performance of ELLR-T twelve combinations of values of BS and T parameters have to be evaluated. Many examples of applications include the computation of SpMV hundred of times with the same pattern of a large sparse matrix A. Then, the selection of parameters BS and T can be carried out in a pre-process stage which includes twelve computations of SpMV and it will require no relevant percentage of run time to optimize the performance of ELLR-T.

Figure 2 shows the performance (GFLOPs) of the SpMV kernels based on the formats that have been evaluated: CRS, CRS(vector), SpMV4GPU, ELLPACK, HYB and moreover the kernel ELLR-T* with optimal values of BS and T. The results shown in that figure allow us to highlight the following major points: (1) Like any parallel implementation of SpMV, the performance obtained by most formats increases with the number of non-zero entries in the matrix, since small matrices do not generate a relevant computational load to reach high parallel performance. Thus, in general, as the dimension of matrices increases, the performance improves. (2) In general, the CRS format yields the poorest performance because the pattern of memory access is not coalescent; (3) The CRS(vector) and SpMV4GPU formats achieve better performance than CRS with most matrices, specially when Av is higher and the distribution of entries is more regular, i.e. $\frac{\sigma}{Av}$ is lower. SpMV4GPU reaches higher performance than CRS(vector) because it better exploits the power of threads, as sixteen threads collaborate to compute every u element, and perform a total coalesced memory access. (4) In general, ELLPACK outperforms both CRS-based formats, however its computation is penalized for some particular matrices, mainly due to the relevance of useless computation of the warps when the matrix histogram includes rows with very uneven length. (5) The performance obtained by HYB is, in general, higher than the four previous formats, but it is remarkable its poorer results for smaller matrices due to the penalty introduced by the call to three different kernels necessary to compute SpMV. Moreover, with specific matrices of higher dimension (qcd5_4, mc2depi, cop20k_A, wbp128, consph, wbp256) it reaches lower or similar performance than ELLPACK, because the percentage of entries stored with ELLPACK format is near to 100 $\%$.(6) Finally, the kernel ELLR-T* based on the format ELLPACK-R clearly achieves the best performance for all matrices considered in this work. In particular, it achieves the highest performance with matrices of high dimensions and higher values of parameters IAv, and $\frac{\sigma}{Av}$. It is remarkable the improvement of the performance reach by ELLR-T* for matrices with large dimension.

Memory optimizations are very relevant to maximize the performance of the GPU. The goal is to maximize the use of the hardware by maximizing bandwidth. The effective bandwidth memory access is a parameter to estimate the level of memory optimization while a specific kernel is executed on the GPU [11]. Table I shows the effective bandwidth achieved when SpMV is computed with ELLR-T* on the GPU GeForce GTX 285 for the set of test matrices in Table I. The effective bandwidth reached by ELLR-T* is high specially for matrices with large dimension. So, for these matrices the effective bandwidth ranges from 90 to 128 GBps, that is 57-80$\%$ of the peak bandwidth (159 GBps) for this card.

In order to estimate the net gain provided by GPUs in the SpMV computation, we have taken the best optimized SpMV implementations for modern processors and for GPUs. For the former, we have considered the MKL implementation of SpMV for a computer based on a state-of-the-art superscalar core, Intel Core 2 Duo E8400, and evaluated the computing times for the set of test matrices. For the GPU GeForce GTX 285, we used the ELLR-T*, which is the best for the GPU according to the results presented above. Table I shows the speedup factors obtained for the SpMV operation on the GPU against one superscalar core, for all the test matrices. The results show that the speedup depends on the matrix pattern, though in general it increases with the number of non-zero entries. The speedup achieves values higher than 30$\times$ for matrices of large dimensions and higher number of entries. In view of the results related to the effective bandwidth and the speed-up achieved by ELLR-T*, we can conclude that the GPU turns out to be an excellent accelerator of SpMV.

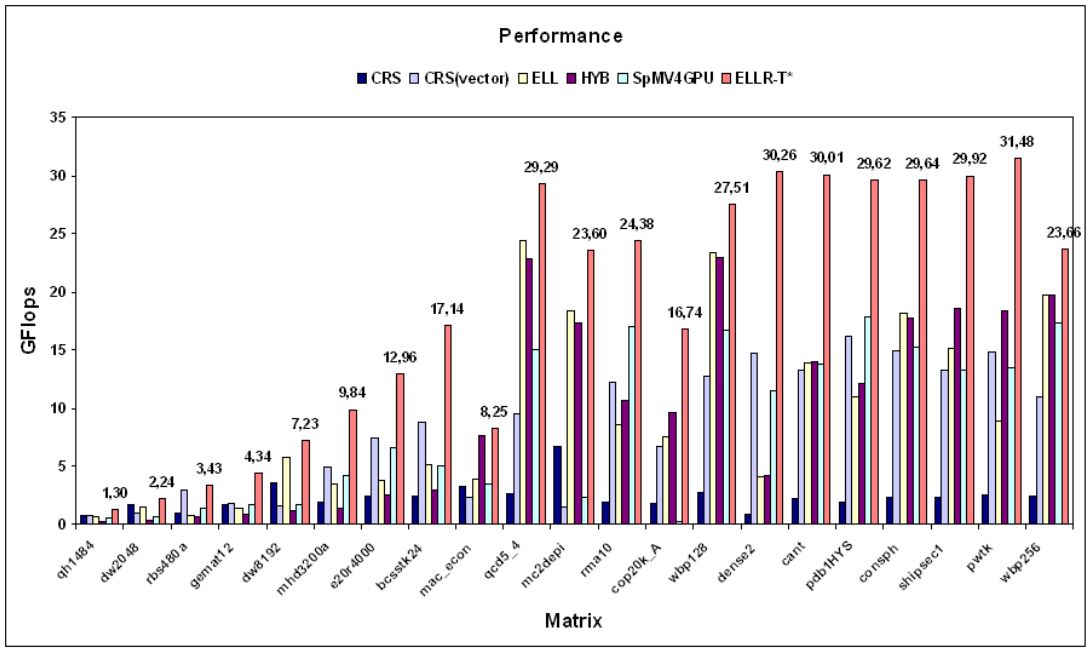


Fig. 2. Performance of SpMV based on different formats on GPU GeForce GTX 285 with the set of test matrices, using the texture cache memory.

VI. Conclusions

In this paper a new approach to compute the sparse matrix vector on GPUs has been proposed and evaluated, ELLR-T. The specific characteristics of ELLR-T based on ELLPACK-R format makes it well suited for GPU computing. The comparative evaluation with other proposals has shown that the performance achieved by ELLR-T is the best after an extensive study on a set of representative test matrices. Therefore, ELLR-T has proven to be superior to the other approaches used thus far. Moreover, the fact that this approach for SpMV does not require any row reordering preprocess makes it specially attractive to be integrated on sparse matrix libraries currently available. A comparison of ELLR-T on a GeForce GTX 285 has revealed that acceleration factors of up to 30× can be achieved in comparison to optimized implementations of SpMV which exploit state-of-the-art superscalar processors. Therefore, GPU computing is expected to play an important role in computational science to accelerate SpMV, especially dealing with problems where huge sparse matrices are involved.

Acknowledgment

This work has been funded by grants from the Spanish Ministry of Science and Innovation TIN2008-01117 and Junta de Andalucia (P06-TIC-01426, P08-TIC-3518)

References

[1] Baskaran MM, Bordawekar R. Optimizing Sparse Matrix-Vector Multiplication on GPUs. *IBM Research Report RC24704*. April 2009.

[2] Baskaran MM, Bordawekar R. Sparse Matrix-Vector Multiplication Toolkit for Graphics Processing Units. April, 2009 http://www.alphaworks.ibm.com/tech/spmv4gpu

[3] Bell N, Garland M. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors Proceedings of SC'09. http://www.nvidia.com/object/nvidia_research_pub_013.html

[4] Buatois L, Caumon G, Levy B. Concurrent number cruncher - A GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems* 2009; **24**(3):205–223

[5] Choi JW, Singh A, Vuduc RW. Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs *Proceedings of PPoPP10, 2010*

[6] Kurzak J, Alvaro W, Dongarra J. Optimizing matrix multiplication for a short-vector SIMD architecture - CELL processor. *Parallel Computing* 2009; **35**(3):138–150

[7] Kincaid DR, Oppe TC, Young DM. ITPACKV 2D User's Guide. *CNA-232* 1989. http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/

[8] Intel. Math Kernel Library. Reference Manual http://software.intel.com/sites/products/documentation/hpc/mkl/mklman.pdf

[9] Kronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*. http://www.khronos.org/developers/library/overview/opencl_overview.pdf

[10] Monakov,A; Lokhmotov, A; and Avetisyan, A. Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures Proceedings of HiPEAC 2010, LNCS 5952, pp. 111- 125, 2010

[11] NVIDIA, CUDA Programming guide. Version 2.3, August, 2009. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf

[12] Ogielski AT, Aiello W. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing* 1993; **14**:519–530.

[13] Toledo S. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*. 1997; **41**(6):711–725

[14] Vázquez F, Garzón EM, Fernández JJ. A matrix approach to tomographic reconstruction and its implementation on GPUs. *Journal of Structural Biology*, 2010; **170**:146–151

[15] Williams S, Oliker L, Vuduc R, Shalf J, Yelick K, Demmel J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing* 2009; **35**(3):178–194