Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach

F. Vázquez[a], J.J. Fernández[a,b], E.M. Garzón[a]

[a] *Almeria University Dpt Computer Architecture and Electronics. Ctra San Urbano s/n Cañada. 04120 Almeria Spain*
[b] *National Biotechnology Center (CSIC). C. Darwin, 3. Campus de Cantoblanco. 28049 Madrid Spain*

Abstract

A wide range of applications in engineering and scientific computing are involved in the acceleration of the sparse matrix vector product (SpMV). 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 is focussed on the ELLR-T algorithm to compute SpMV on GPU architecture, its performance is strongly dependent of the optimum selection of two parameters. Then, taking account that the memory operations dominate the performance of ELLR-T, an analytical model is proposed in order to obtain the auto-tuning of ELLR-T for particular combinations of sparse matrix and GPU architecture. The evaluation results with a representative set of test matrices show that the average performance achieved by auto-tuned ELLR-T by means of the proposed model is near to the optimum. A comparative analysis of ELLR-T against a variety of previous proposals shows that ELLR-T with the estimated configuration reaches the best performance on GPU architecture for the representative set of test matrices.

Keywords:
Sparse matrix vector product, GPU computing, GPU performance modeling

1. 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. For many applications based on MV, the matrix is large

and sparse. Sparse matrices are involved in linear systems, eigensystems and partial differential equations from a wide spectrum of scientific and engineering disciplines[1]. For these problems the optimization of the sparse matrix vector product (SpMV) is a challenge because of the irregular computation of large sparse operations. 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 in [2, 3, 4].

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) [5] and OpenCL [6] that greatly facilitate the development of applications targeted at GPUs.

Recently, several implementations of SpMV have been developed with CUDA and evaluated on GPUs [7, 8, 9, 10, 11]. Devising GPU-friendly matrix storage formats has been a key in these implementations. This work is focused on the ELLR-T algorithm which relies on the storage format for the sparse matrix, ELLPACK-R [12, 13]. This format is a GPU-friendly variant of one previously designed for vector architectures, ELLPACK [14]. 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 implementation based on ELLR-T proves to outperform the most common and efficient formats for SpMV on GPUs used so far. However, the ELLR-T performance strongly depends on the values of two parameters related to the the configuration of ELLR-T according to the particular combination of input sparse matrix/GPU architecture. The goals of this work are: (1) to analyze the ELLR-T algorithm; (2) to propose a model to optimize the ELLR-T performance and (3) to evaluate the performance achieved by ELLR-T algorithm with the modeled configuration.

Currently there are several proposals to model the performance of GPUs executing general applications. Thus, an analytical model of GPU performance based on the estimation of the memory warp parallelism is analyzed in [15]. However the complexity of the management memory in the GPU architecture introduces a large number of factors to instantiate the model on specific applications and GPU architectures. On the other hand, a tool to predict the GPU performance is proposed on [16]. It automatically generates mini-kernels from specific applications for benchmark the GPU architecture

and lately introduces the experimental measures on their model, taking account the symbolic evaluation of the application code. More closely to our proposal, a model to estimate the performance of SpMV on GPUs based on the blocked ELLPACK implementation is introduced in [17]. However, bearing in mind that the ELLR-T performance is strongly determined by its memory access, this work proposes a analytical model based on the evaluation of the GPU memory operations when ELLR-T is executed on a specific GPU architecture for a particular matrix, in order to obtain the values of the parameters that optimize the performance. The evaluation of the proposed model proves its high accuracy, and it can be included as a previous stage of ELLR-T in order to get its auto-tuning on run time, since it takes as inputs the row lengths of the matrix and two parameters of GPU architecture.

The remainder of the paper is structured as follows. Next Section 2 describes the main characteristics of the computational GPU model focusing the interest on the main factors which impact on the GPU performance. Then, Section 3 starts with a review of the different formats to compress sparse matrices, following the details of ELLR-T algorithm are analyzed, this section ends proposing a model to optimize the performance of ELLR-T. In Section 4 the evaluation results show the high performance of ELLR-T achieved by a NVIDIA Geforce GTX 285 with a set of representative sparse matrices when the proposed model is used to configure it. The results clearly show that the ELLR-T with the proposed model gets the best performance for all the test matrices. Finally, Section 5 summarizes the main conclusions.

2. Computational GPU model

GPUs have hundreds of cores that can collectively run thousands of computing threads. Each core, called Scalar Processor (SP), belongs to a set of multiprocessors units called Streaming Multiprocessors (SM) that compose the device.

Using the CUDA interface, the GPU is considered by the programmer as a set of SIMT (Single Instruction, Multiple Threads) multiprocessors [5]. Each kernel (parallel code) is executed as a batch of threads organized as a grid of thread blocks whose configuration is defined by the programmer setting up specific parameters. One of these parameters is the threads block size, hereinafter it is denoted as BS. On run time, the blocks are cyclically mapped on the SMs, as it is illustrated in Figure 1. The blocks in turn are divided into sets of threads called warps, the warp size (denoted here as

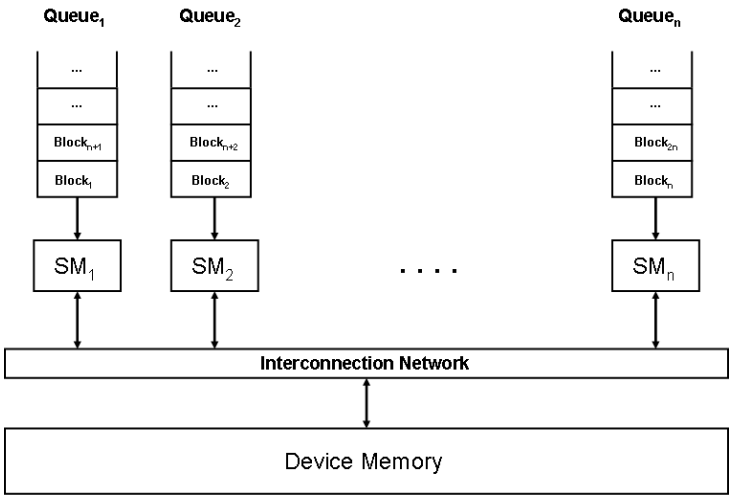


Figure 1: Threads Blocks Mapping on the GPU architecture

$ws = 32$) is defined by the architecture. Currently, the SMs are composed by eight (or thirty-two) SPs on the most extended NVIDIA GT (or Fermi) GPU architectures [5, 18].

According to the memory resources in every SM and the particular kernel memory requirements, every SM can simultaneously execute a set of maximum active warps. Thus, the blocks queue on every SM is executed in pipeline, then, if one warp executes a slow memory access, the execution of the following warp is started, so that the slow memory access can be hidden by the computation if the pipeline is fed [19]. In order to optimize the exploitation of the NVIDIA GPU architecture the programmer has to attend to maximize the *bandwidth memory*, the memory management can be optimized if the access pattern of the different threads belonging to every half-warp (16 threads) verifies the coalescence and alignment 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. Moreover, the use of texture memory improves the performance when the searched word is located within it [5].

3. Accelerating the SpMV on GPUs. ELLR-T approach

3.1. Formats to compress sparse matrices

Several formats have been proposed to optimize the computation with sparse matrices for a specific architecture. These formats define the locality or the coalescence of memory access for the SpMV, which are essential to optimize the performance on CPU or GPU architectures. Before describing the details of ELLR-T kernel let us review the more used formats to store sparse matrices.

Let $u = Av$ be a sparse matrix vector product where A is the sparse matrix, v and u are the input and output vectors respectively, every specific algorithm to compute $u = Av$, exploiting a particular architecture, is related to a specific format to store A.

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 [20]. Despite these drawbacks, several optimizations have made possible to improve the performance of sparse computation on current processors [4, 21]. 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.

ELLPACK or ITPACK [14] 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 the SpMV based on ELLPACK can improve the performance due to the coalesced global memory access, thanks to the column-major ordering used to store the matrix elements into the data structures. 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.

Recently, different *kernels to compute SpMV on GPUs* have been proposed [7, 8, 9, 10, 11, 22]. On the one hand, the kernel called CRS(vector) evaluated in [7] is based on CRS format. It computes every output vector element with the collaboration of the 32 threads of every warp. Similarly, another kernel based on the CRS format has been proposed in [9] and it has been included on the SpMV4GPU library [23]. Here the collaboration of 16 threads (half warp) computes every output vector element, and padding-zeroes are added to every row to complete a length multiple of 16, in order to fulfill the memory alignment requirements and improve the coalesced memory access. On the other hand, the kernels related to the format called HYB (which stands for hybrid) proposed by [7] seem to yield high performance on GPUs. With the goal of improving the performance of ELLPACK it combines the ELLPACK and coordinate storage scheme (COO) formats, this last format consists of three linear arrays to store the entries, columns indexes and row indexes of the sparse matrix. Recently, the format called *Sliced ELLPACK* has been proposed and evaluated in [11]. 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 [10], 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.

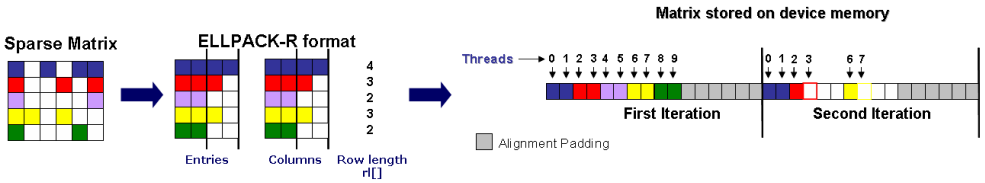
Moreover, we have devised the kernel based on the format ELLPACK-R, it has proved achieve better performance on GPUs for a high percentage of the representative test matrices [12, 22]. Recently, we have proposed the kernel ELLR-T with the same kind of format to store the sparse matrix. It reaches the best performance if optimum values of two parameters are selected [13]. Next, this kernel is described in depth, with the aim of analyzing the model for the selection of the optimum parameters.

3.2. Computing SpMV with ELLR-T algorithm 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 to 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 $\lceil rl[i]/T \rceil$ iterations of the inner loop of SpMV. Every thread stores its partial computation in the shared memory of the GPU. 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 explored in order to obtain the best performance with every kind of sparse matrices. Figure 2 illustrates the characteristics of the code of ELLR-T, underlining on top the specific storage for the sparse matrix on the device memory to warranty that the device memory access of every set of $T = 2$ threads is coalescent and aligned. This characteristic is very relevant for ELLR-T due to the high memory access related to computation of the SpMV. 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.



(a)

```
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 i-th row )
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 for 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] 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)

Figure 2: (a) The ELLR-T memory storage of the sparse matrix fulfilling the coalescence and alignment conditions for $T = 2$ and (b) ELLR-T code to compute SpMV on GPUs including the reduction on share memory for $T = 32$

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 $k = \lceil rl[i]/T \rceil \leq \lceil Max\_nzr/T \rceil$ for specific sets, S_i, into the warp. Then, the run-time of every warp is proportional to the maximum element of the sub-vector $\lceil rl[i]/T \rceil$ related to every warp, and it is not necessary for the k-loop reaches $k = \lceil Max\_nzr/T \rceil$ for all threads, then, there are not useless iterations and the control of loops of this

implementation is reduced comparing with SpMV based on ELLPACK. However, if the value of T excessively increases, relevant number of threads are unloaded, the unbalance increases and the kernel achieves a poor performance.

Consequently, ELLR-T is devised to exploit the GPU architecture computing the SpMV operation, however, in order to reach the best performance it is very relevant to select the optimum values of two parameters: (1) T, that is, the number of threads which collaborate to compute one element of output vector and related to every matrix row, then, it is an specific parameter of ELLR-T; and (2) BS, that is, the block size of the CUDA code, is a general parameter to optimize the CUDA programs.

3.3. Model for optimizing ELLR-T

Our goal is to define a model to predict the values of BS and T which optimize ELLR-T for the specific combination of a sparse matrix and a GPU architecture. Then, it is not necessary an accurate estimation of GPU performance, since our aim is just to determine the values of the parameters which maximize the performance, knowing the particularities of a sparse matrix and a GPU platform.

According to our experience and the results described in [16, 17], the performance reached by SpMV on GPUs is dominated by the memory operations. They are strongly related to the pattern of the sparse matrix and, additionally, the memory broadband is related to the resources of the GPU and the kind of memory access included in the kernel. More specifically for ELLR-T algorithm, the number of memory operations of every set of T threads reading the i-th row is proportional to its length, that is $rl[i]$, and every thread accesses to the device memory $rl[i]/T$ times. From other point of view, the broadband memory is proportional to the number of stream multiprocessors (SM) of the GPU denoted by n. It is reasonable to think that the run-time of ELLR-T is proportional to the number of memory access executed by every SM. Moreover, there will is an unbalance between the different SMs of GPU due to (1) the irregularities of the matrix row length, and (2) the mismatches between the number of SMs, n, and the number of threads blocks $\lceil \frac{N*T}{BS} \rceil$. Consequently, the performance of ELLR-T is proportional to the run-time of the most loaded SM, i. e. to the maximum number of memory access executed by one SM on the GPU.

Bearing in mind the above considerations and the threads mapping on GPU architecture, described in Section 2, let us define the model for optimizing ELLR-T. We suppose that the value of block size, BS, lets to active enough number of threads in every SM, so that the arithmetic computation is hidden by the memory access on the pipeline of GPU computation and it is not necessary to include it in the model. Then, it is coherent to establish that the best performance of ELLR-T is reached when the number of memory access on the most loaded SM is minimum.

Let B_{sm} be the number of blocks for the SM indexed by sm, we suppose that the run time is proportional to the number of the memory access for every SM and it can be estimated as:

$$M_{sm} = \sum_{b=0}^{b=B_{sm}-1} M_{sm}^{b} \tag{1}$$

where b is the index of block related to the smth-multiprocessor and M_{sm}^{b} is proportional to the memory access of the bth-block.

$$M_{sm}^{b} = \sum_{w=0}^{w=\lceil \frac{BS}{ws} \rceil - 1} M_{sm}^{b}(w) \tag{2}$$

where ws is the warp size ($ws = 32$ for NVIDIA GPU architectures) Analyzing the ELLR-T code, the number of access to device memory for the wth-warp can be estimated as:

$$M_{sm}^{b}(w) = MAX_1(rl, T, b, w) + MAX_2(rl, T, b, w) \tag{3}$$

where MAX_1 and MAX_2 represent the maximum row length into the set of rows related to the first and second half of the wth-warp respectively, with this warp belonging to the b-th block, that is

$$MAX_1(rl, T, b, w) = \{rl(ws/T * w + x_w) \mid 0 \leq x_w < (ws/2)/T\}$$

and

$$MAX_2(rl, T, b, w) = \{rl(ws/T * w + x_w) \mid (ws/2)/T \leq x_w < ws/T\}$$

where x_w is the thread identifier belonging to the wth-warp.

Then, according to the previous analysis the ELLR-T run-time is proportional to the maximum number of memory access by one SM, denoted by M_{ELLR-T}, which can be expressed as:

$$M_{ELLR-T}(rl, BS, T) = MAX_{sm=0..n-1} \left\{ \sum_{b=0}^{b=B_{sm}-1} \sum_{w=0}^{w=\lceil \frac{BS}{ws} \rceil - 1} M_{sm}^{b}(w) \right\} \quad (4)$$

Notice that the value of $M_{ELLR-T}(rl, BS, T)$ is not an estimation of the run-time, however it can be used to locate the optimum ELLR-T configuration because $M_{ELLR-T}(rl, BS, T)$ and ELLR-T run-time achieve their extreme values for the same BS and T values approximately.

The analytical expression defined by Equation 4 is the key of our model and it establishes the relation between the memory activity and the parameters BS and T for one particular combination of matrix/GPU architecture before that ELLR-T starts its execution. The matrix characteristics are introduced on the model with the $rl[]$ vector and the GPU architecture by means of the the number of SMs, n, and the warp size ws which are static parameters of the architecture. So that, this model can be integrated as the first stage of ELLR-T without previous benchmark of the GPU architecture and without relevant overload in order to determine the optimum configuration of the ELLR-T, that is, the values of BS and T which achieve the minimum of M_{ELLR-T}.

4. Evaluation

This section is intended to evaluate: (1) the accuracy of the proposed model to predict the optimum configuration of the ELLR-T kernel when the input matrix and GPU architecture are known; and (2) the performance achieved by the ELLR-T kernel when it is configured according to the prediction of proposed model.

The evaluation results have been obtained on a GeForce GTX 285 as the test architecture. Moreover, the set of test sparse matrices described in Table 1 has been considered. They are related to different disciplines of science and engineering. Table 1, on the left, summarizes the characteristic parameters associated to specific patterns of the set of test matrices: number of rows (N), total number of non-zeros elements (*Entries*), average number of entries per row (Av), percentage of relative standard deviation of entries

Table 1: Set of test matrices. Characteristic parameters related to entries distribution on the rows (on the left). Actual and estimated optimum parameters (on the right)

Matrix	N	*Entries*	Av	$\frac{\sigma}{Av}$	BS^O	T^O	$BS^\star$	$T^\star$
qh1484	1.484	6.110	4,1	38,9	64	8	128	4
dw2048	2.048	10.114	4,9	10,2	64	4	128	2
rbs480a	480	17.087	35,6	1,4	64	16	128	8
gemat12	4.929	33.111	6,7	44,8	256	4	128	2
dw8192	8.192	41.746	5,1	12,0	64	1	128	1
mhd3200a	3.200	68.026	21,3	27,4	128	8	128	8
e20r4000	4.241	131.556	31,0	49,6	256	4	128	8
bcsstk24	3.562	159.910	45,0	25,6	128	8	128	4
mac_econ	206.500	1.273.389	6,2	71,9	512	4	512	2
qcd5_4	49.152	1.916.928	39,0	0,0	128	4	128	1
mc2depi	525.825	2.100.225	4,0	1,9	128	1	128	1
rma10	46.835	2.374.001	50,7	56,1	128	8	128	8
cop20k_A	121.192	2.624.331	21,6	63,7	128	4	128	1
wbp128	16.384	3.933.095	240,1	14,5	128	1	128	8
dense2	2.000	4.000.000	2.000	0,0	256	8	128	32
cant	62.451	4.007.383	64,2	21,9	512	4	128	4
pdb1HYS	36.417	4.344.765	119,3	26,7	128	8	256	8
consph	83.334	6.010.480	72,1	26,4	512	2	128	1
shipsec1	140.874	7.813.404	55,5	20,0	128	4	128	2
pwtk	217.918	11.634.424	53,4	8,9	512	2	128	1
wbp256	65.536	31.413.932	479,3	14,7	128	2	128	8

by row($\frac{\sigma}{Av}$). These parameters display the variability or dispersion of the number of entries by row of the matrices. Their diversity in the set of test matrices is significant. All matrices are real of dimensions $N \times N$. Although some of them are symmetric, they all have been considered as general to compute SpMV. The remainder information included on Table 1 will be described on the posterior analysis.

The programming interface, CUDA, allows the programmer to specify which variables are to be stored in the texture cache within the memory hierarchy [5]. Here, vector v has been stored binding to the texture memory for all kernels evaluated, since only the vector v is reused throughout the products with the different rows of the matrix, in the computation of $u = Av$.

In order to illustrate the impact of the parameters BS and T on the performance of ELLR-T, Figure 4 shows the performances achieved by ELLR-T with the optimum and worst configurations (denoted by ELLR-T^O and ELLR-T^W respectively) for the set of test matrices. Both configurations have been defined by means of exhaustive search. As it can be seen, there are relevant differences between the performance reached by both configurations. So, it underlines that the selection of appropriate values for BS and T has a strong impact on the performance of ELLR-T. Table 1, on the right, shows the optima (BS^O and T^O) and the values estimated ($BS^\star$ and $T^\star$) by the proposed model. So $BS^\star$ and $T^\star$ minimize M_{ELLR-T} subject to the mapping restrictions according to the GPU resources. The optima and the estimated values match for few matrices, this result could indicate that the proposed model fails. However, the performance achieved by the optima and the estimated values are very near as shown in Figure 4, where ELLR-T* is referred to the ELLR-T kernel with the estimated configuration.

In order to analyze this fact, Figure 3 shows M_{ELLR-T} as function of the parameters T and BS for four illustrative examples of test matrices: rma10, pdb1HYS, mac_econ and dense2. As it is shown, very different plots of M_{ELLR-T} are related to every matrix. So, if the interest is focused on rma10, the minimum of M_{ELLR-T} is well-pronounced for the specific combination of the parameters $BS^\star = 128$ and $T^\star = 8$ and the optimum performance is achieved by these values, that is $BS^\star = BS^O$ and $T^\star = T^O$. However, the plot of M_{ELLR-T} for pdb1HYS exhibits almost the same minimum value for $BS = 128, 256, 512$ and $T = 8$, thus the performance achieved by $BS^\star = 256$ and $T = 8$ is almost the optimum, although $BS^\star \neq BS^O = 128$. This plot for mac_econ is very different because it shows that M_{ELLR-T} does not depend on BS and almost the same minimum of M_{ELLR-T} is achieved by two values of $T = 2, 4$, then, the estimated values by the model $BS^\star = 512$ and $T^\star = 2$ get almost the optimum performance of $BS^O = 512$ and $T^O = 4$. However, the plot for the matrix dense2 shows that M_{ELLR-T} decreases as T increases and several combinations of BS and T achieve nearly the same minimum value due the regular pattern of this matrix, so, the values defined by the model $BS^\star = 128$ and $T^\star = 32$ get nearly the optimum performance although $BS^O = 256$ and $T^O = 8$. Then, it can be concluded that the proposed model helps to define one combination of both parameters which nearly achieves the optimum performance.

In order to evaluate the accuracy of the proposed model, Table 2 shows the distribution of the test matrices according to the Matching Percentage, i.e.

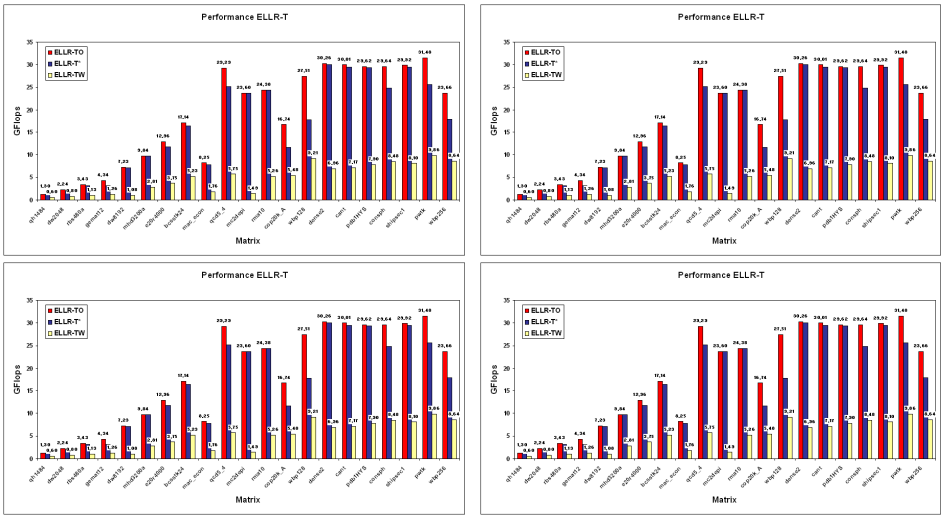


Figure 3: M_{ELLR-T} as function of the parameters T and BS for the test matrices rma10, pdb1HYS, mac_econ and gemat12.

the ratio between the performances achieved by ELLR-T* and ELLR^O. So, data on left column point that ELLR-T* and ELLR-T^O reach the same performance for twelve matrices, and data on right column show that ELLR-T* reaches between 70% and 60% of the optimum performance for two matrices. Moreover, the average Matching Percentage in the set of test matrices is 91,6%. Then, the results show that the configuration of ELLR-T defined by the proposal model achieves a performance close to the optimum, although the estimated parameters differ to the optima.

Additionally, both configurations have been evaluated without to use the texture memory as cache for the v vector storage, and the matching percentage achieved is nearly 99%. These results are coherent with the model hypothesis, since they do not consider the GPU caches management and they confirm the accuracy of the model. Notice that the performances of ELLR-T* and ELLR-T^O differ more relevantly for the matrices cop20k_A, wbp128, pwtk and wbp256; the entries location exhibits high regularity for these matrices. Consequently, the corresponding SpMV has a higher locality level, so the cache management has more relevant impact on the performance. This fact could justify the inaccuracy of the model for these matrices.

Next, a comparative evaluation of the performance achieved by ELLR-T* and the kernels CRS, CRS(vector), SpMV4GPU, ELLPACK, HYB will

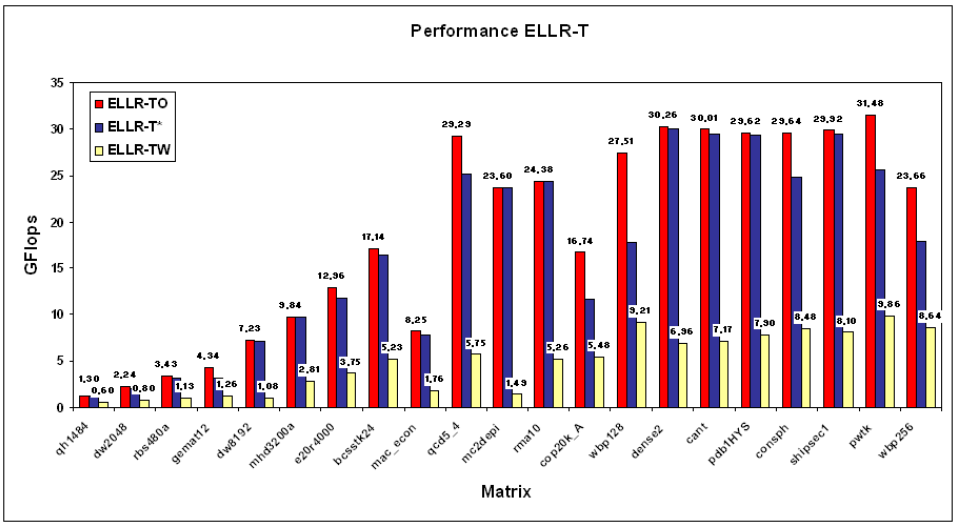


Figure 4: Performances achieved by ELLR-T with the optimum, modeled and worst configurations (denoted by ELLR-T^O, ELLR-T* and ELLR-T^W respectively) for the set of test matrices

Matching Percentage	100%	90%	80%	70%	60%
Number of test matrices	12	2	3	2	2

Table 2: Distribution of test matrices

be developed. Figure 5 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 the estimated values of BS and T. The results shown allow us to highlight the following major points:

1. As 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

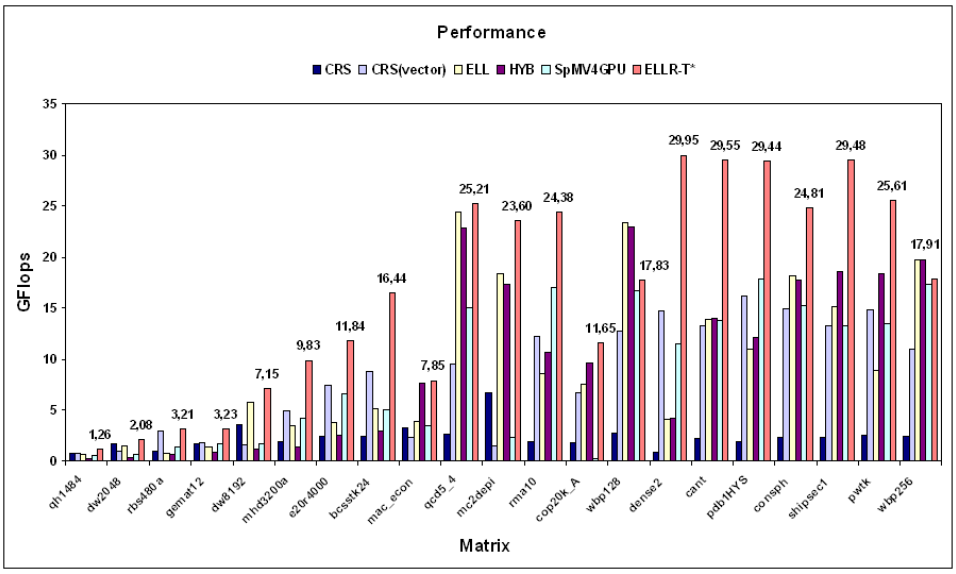


Figure 5: Comparing the ELLR-T* performance with the SpMV based on different formats on GPU GeForce GTX 285 with the set of test matrices

reaches higher performance than CRS(vector) because it better exploits the power of threads.

4. In general, ELLPACK outperforms previous 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.
6. Finally, the kernel ELLR-T* based on the format ELLPACK-R achieves the best performance for all matrices considered, except for the matrices cop20k_A, wbp128, pwtk and wbp256. In particular, it achieves the highest performance with matrices of higher dimension and higher value $\frac{\sigma}{Av}$ and a random pattern.

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. Table 3 shows the effective bandwidth achieved when SpMV is computed with ELLR-T* on the GPU GeForce GTX 285 for the set of test

Table 3: Effective Bandwidth of memory access (*Bandwidth*) of SpMV with ELLR-T* on the GPU GeForce GTX 285 for the set of test matrices

Matrix	*Bandwidth*	Matrix	*Bandwidth*	Matrix	*Bandwidth*
qh1484	6,5	bcsstk24	69,9	dense2	121,1
dw2048	10,8	mac_econ	38,1	cant	121,4
rbs480a	14,0	qcd5_4	120,2	pdb1HYS	119,5
gemat12	19,9	mc2depi	118,0	consph	120,0
dw8192	34,6	rma10	99,5	shipsec1	121,8
mhd3200a	41,2	cop20k_A	70,1	pwtk	128,3
e20r4000	53,5	wbp128	110,5	wbp256	94,8

matrices in Table 1. It 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. Consequently, these results show that in spite of the irregularity of the SpMV, the high percentage of coalescent and aligned memory access of ELLR-T allows to achieve a high effective bandwidth.

In order to estimate the net gain provided by GPUs over modern processors in the SpMV computation, we have taken the best optimized SpMV implementations for both kind of architectures. 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 latter, we used the ELLR-T*, which is the best for the GPU according to the results presented above. Figure 6 shows the acceleration factors obtained by the SpMV operation on the GPU against one superscalar core for all the test matrices. The results show that the acceleration depends on the matrix pattern, though, in general, it increases with the number of non-zero entries. The acceleration factor 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 with the predicted configuration by means of the proposed model, we can conclude that the GPU turns out to be an excellent accelerator of SpMV by means of the ELLR-T* algorithm.

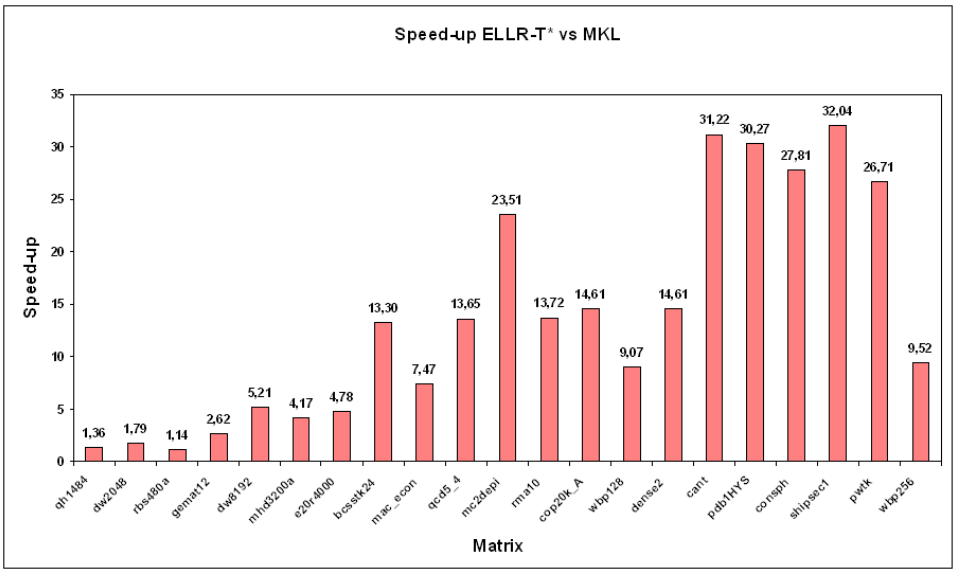


Figure 6: Acceleration factor achieved by ELLR-T* on GPU GeForce GTX 285 over one superscalar core of Intel Core 2 Duo E8400 with the set of test matrices

5. Conclusions

In this paper a new approach to compute the sparse matrix vector product on GPUs has been evaluated. This approach is based on the kernel ELLR-T whose performance depends on the optimum selection of two parameters; one is particular for ELLR-T (the number of Threads collaborating to compute the same output vector element) and the other is general for all CUDA kernel (the threads block size). A model to auto-tune the ELLR-T kernel has been proposed. It is based on the evaluation of memory access by the stream multiprocessors of GPU architecture, knowing the rows length of a particular matrix. The estimated configurations by the model are close to the optima, so the ELLR-T achieves 91% of optimum performance when the model estimations are considered. The comparative evaluation with other proposals has shown that the performance achieved by ELLR-T with the configuration estimated by the model is the best. Therefore, ELLR-T combined with the proposed model 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 and any benchmark process, 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 with optimized implementations of SpMV which exploit state-of-the-art superscalar processors.

Acknowledgment

This work has been funded by grants from the Spanish Ministry of Science and Innovation TIN2008-01117 and Junta de Andalucia (P10-TIC-6002, P08-TIC-3518).

[1] F. Vázquez, E. Garzón, J. Fernández, A matrix approach to tomographic reconstruction and its implementation on GPUs, Journal of Structural Biology 170 (2010) 146–151.

[2] A. Ogielski, W. Aiello, Sparse matrix computations on parallel processor arrays, SIAM J. Sci. Comput 14 (1992) 519–530.

[3] S. Toledo, Improving memory-system performance of sparse matrix-vector multiplication, in: IBM Journal of Research and Development, 1997.

[4] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, Parallel Comput. 35 (3) (2009) 178–194. doi:http://dx.doi.org/10.1016/j.parco.2008.12.006.

[5] NVIDIA, CUDA Programming guide. Version 2.3, 2009.

[6] Kronos, Group, OpenCL - the open standard for parallel programming of heterogeneous systems.

[7] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ACM, New York, NY, USA, 2009, pp. 1–11. doi:http://doi.acm.org/10.1145/1654059.1654078.

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

[9] M. Baskaran, R. Bordawekar, Optimizing sparse matrix-vector multiplication on GPUs, Tech. Rep. Research Report RC24704, IBM (April 2009).

[10] J. Choi, A. Singh, R. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, in: Proceedings of PPoPP10, 2010.

[11] A. Monakov, A. Lokhmotov, A. Avetisyan, Automatically tuning sparse matrix-vector multiplication for GPU architectures, in: Proceedings of HiPEAC 2010, LNCS 5952, 2010, pp. 111–125.

[12] F. Vázquez, E. Garzón, J. Martínez, J. Fernández, Accelerating sparse matrix vector product with GPUs, in: 9th International Conference on Computational and Mathematical Methods in Science and Engineering. CMMSE, 2009, pp. 1081–1092.

[13] F. Vázquez, G. Ortega, J. Fernández, E. Garzón, Improving the performance of the sparse matrix vector product with GPUs, in: 10th IEEE International Conference on Computer and Information Technology. CIT 2010, IEEE Computer Society, 2010, pp. 1146–1151. doi:http://dx.doi.org/10.1109/CIT.2010.208.

[14] D. Kincaid, T. Oppe, D. Young, ITPACKV 2D User's guide, Tech. Rep. CNA-232, Center for Numerical Analysis. University of Texas at Austin (1989).

[15] S. Hong, H. Kim, An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness, in: SIGARCH Comput. Archit. News 37, 3, ACM, New York, NY, USA, 2009, pp. 152–163. doi:http://doi.acm.org/10.1145/1555815.1555775.

[16] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, W. Hwu, An adaptive performance modeling tool for gpu architectures, in: PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM, New York, NY, USA, 2010, pp. 105–114. doi:http://doi.acm.org/10.1145/1693453.1693470.

[17] J. Choi, A. Singh, R. Vuduc, Model-driven autotuning of sparse matrix-vector multiply on GPUs, in: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel program-

ming, PPoPP '10, ACM, New York, NY, USA, 2010, pp. 115–126. doi:http://doi.acm.org/10.1145/1693453.1693471.

[18] NVIDIA, Next generation CUDA architecture. Fermi Architecture, 2010.
URL http://www.nvidia.com/object/fermi_architecture.html

[19] NVIDIA, CUDA C programming. Best practices guide. CUDA Toolkit 2.3, July 2009.

[20] J. Kurzak, W. Alvaro, J. Dongarra, Optimizing matrix multiplication for a short-vector SIMD architecture - CELL processor, Parallel Computing 35 (3) (2009) 138 – 150, revolutionary Technologies for Acceleration of Emerging Petascale Applications. doi:DOI: 10.1016/j.parco.2008.12.010.
URL http://www.sciencedirect.com/science/article/B6V12-4VDY7V0-2/2/2fc7165879200d9298ebe290b285bbdd

[21] INTEL, Math kernel library. reference manual, 2009.

[22] F. Vázquez, J. Fernández, E. Garzón, A new approach for sparse matrix vector product on NVIDIA GPUs, Concurrency and Computation: Practice And Experience. In Press.

[23] M. Baskaran, R. Bordawekar, Sparse matrix-vector multiplication toolkit for graphics processing units.
URL http://www.alphaworks.ibm.com/tech/spmv4gpu