CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2011; **23**:815–826
Published online 27 September 2010 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1658

# A new approach for sparse matrix vector product on NVIDIA GPUs

F. Vázquez, J. J. Fernández and E. M. Garzón*, †

*Department of Computer Architecture and Electronics, University of Almería, Ctra. Sacramento s/n, Almería 04120, Spain*

SUMMARY

The sparse matrix vector product (SpMV) is a key operation in engineering and scientific computing and, hence, it has been subjected to intense research for a long time. The irregular computations involved in SpMV make its optimization challenging. Therefore, enormous effort has been devoted to devise data formats to store the sparse matrix with the ultimate aim of maximizing the performance. Graphics Processing Units (GPUs) have recently emerged as platforms that yield outstanding acceleration factors. SpMV implementations for NVIDIA GPUs have already appeared on the scene. This work proposes and evaluates a new implementation of SpMV for NVIDIA GPUs based on a new format, ELLPACK-R, that 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, although the performance strongly depends on the specific pattern of the matrix, the implementation based on ELLPACK-R achieves higher overall performance. Moreover, a comparison with standard state-of-the-art superscalar processors reveals that significant speedup factors are achieved with GPUs. Copyright © 2010 John Wiley & Sons, Ltd.

Received 6 August 2009; Revised 9 July 2010; Accepted 16 July 2010

KEY WORDS: GPU; high performance computing; sparse matrix vector product

## 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 [1]. The relevance of this kind of an operation in computational sciences is supported by the constant effort devoted to optimize the computation of MV for the processors at the time, which range from the early computers in the 1970s to the last modern multi-core architectures [2–5]. A proof of this effort is the evolution of Basic Linear Algebra Subroutines (BLAS), where MV is considered a relevant operation, because it has constantly been improved and optimized as computer architectures have evolved [6–8].

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 [9]. For these problems the optimization of the sparse matrix vector product (SpMV) is a challenge because of the irregular computation

*Correspondence to: E. M. Garzón, Department of Computer Architecture and Electronics, University of Almería, Ctra. Sacramento s/n, Almería 04120, Spain.
†E-mail: gmartin@ual.es

Contract/grant sponsor: Spanish Ministry of Science and Innovation; contract/grant number: TIN2008-01117
Contract/grant sponsor: Junta de Andalucía; contract/grant numbers: P06-TIC01426, P08-TIC03518

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 as in the case of dense matrices (see [10]). 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 [2–5].

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 [12], that greatly facilitate the development of applications targeted at GPUs. Specifically, dense algebra operations are accelerated by GPU computing and the library CUBLAS [8] is now publicly available to get easily high performance with NVIDIA GPUs in these operations. Recently, several implementations of SpMV have also been developed with CUDA and evaluated on NVIDIA GPUs [13–15]. 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 NVIDIA GPUS which relies on a new storage format for sparse matrices, ELLPACK-R. This format is a GPU-friendly variant of the one previously designed for vector architectures, ELLPACK [16]. 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 ELLPACK-R outperforms the most efficient formats for SpMV on GPUs used so far.

Next, Section 2 summarizes the aspects related to GPU programming and computing. Then, Section 3 reviews the different formats to compress sparse matrices and the corresponding codes to compute SpMV, given that the selection of an appropriate format is the key to optimize SpMV on GPUs. Section 4 introduces the proposed format for computation of SpMV on GPUs. In Section 5 the performance measured on a NVIDIA Geforce GTX 295 with a set of representative sparse matrices belonging to diverse applications is presented. The results clearly show that the new storage format presented here, ELLPACK-R, gets the best performance for most of the test matrices. Finally, Section 6 summarizes the main conclusions.

## 2. COMPUTATIONAL KEYS TO EXPLOIT NVIDIA GPUs

Compute Unified Device Architecture (CUDA) provides a set of extensions to standard ANSI C for programming NVIDIA GPUs. It supports heterogeneous computations where applications use both the CPU and the GPU. Serial portions of applications are run on the CPU, and parallel portions are accelerated on the GPU. These portions executed in parallel by the GPU are called kernels [11]. 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. The number of SMs ranges from 16 (NVIDIA Tesla C870) to 30 in modern GPUs (NVIDIA Geforce GTX 295). The SPs in an SM share resources, such as registers and memory. The on-chip shared memory allows the parallel tasks running on these cores to share the data without the need of sending it over the system memory bus [11]. For an in-depth description of NVIDIA GPUs, the reader is referred to [11].

To develop codes for NVIDIA GPUs with CUDA, the programmer has to take into account several architectural characteristics, such as the topology of the multiprocessors and the management of the memory hierarchy. For the execution of the program, the CPU (called host in CUDA programming) performs a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks. The execution of every thread block is assigned to every SM. Moreover, every block is composed of several groups of 32 threads called warps. All threads belonging to a warp execute the same program over different data. The size of every thread block is defined by the programmer. The maximum instruction throughput is got when all threads of the same warp execute the same instruction sequence, given that any flow



DOI: 10.1002/cpe

control instruction can cause the threads of the same warp to diverge, that is, to follow different execution paths. If this occurs, the different execution paths have to be serialized, increasing the total number of instructions executed for this warp [11].

Another key to take advantage of NVIDIA GPUs is related to the memory management. There are several kinds of memory available on GPUs with different access times and sizes that constitute a memory hierarchy. The effective bandwidth can vary by an order of magnitude depending on the access pattern for each type of memory. There is a parallel memory interface between the global memory and every SM of the GPU. The GPU architecture performs the memory access in groups of 16 threads (called half-warps). If the 16 threads belonging to the half warp perform the access under certain memory access patterns (coalescence conditions), it can be done in parallel by all of them and the memory latency would be the same as that of a single access. However, if these patterns are not met, 16 separate accesses will be needed with a total latency of 16 times more than a single access [11]. Hence, the ordering of the data access chosen in an algorithm may have significant performance effects during GPU memory operations. Another point to take into consideration is the memory write conflicts which are produced when several threads attempt to write the same memory location. With atomic functions it is ensured that only one thread updates a memory location at a time, although at the expense of a penalty.

From the programmer's point of view, the GPU is considered as a set of SIMD (Single Instruction stream, Multiple Data streams) multiprocessors with shared memory. Therefore, the SPMD (Single Program Multiple Data) programming model is offered by CUDA. Moreover, in order to optimize the GPU performance, the programmer has to consider two main goals: (1) to balance the computation of the sets of threads and (2) to optimize the data access through the memory hierarchy. Specifically, to optimize SpMV on GPUs, both 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.

## 3. FORMATS TO COMPRESS SPARSE MATRICES. AN OVERVIEW OF SPMV AND ITS CHALLENGES

Several formats have been proposed in the literature 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.

The pattern of memory access to read the elements of the sparse matrix has a strong impact in the performance of SpMV. Thus, every specific algorithm to compute SpMV exploiting a particular architecture is related to a specific format to store the sparse matrix. Then, the key to increase the performance of SpMV with GPUs is the development of an appropriate algorithm with its format.

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.

### *3.1. 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 corresponding code to compute SpMV can be seen in Figure 1(left). The performance of SpMV may be penalized by COO 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.



DOI: 10.1002/cpe

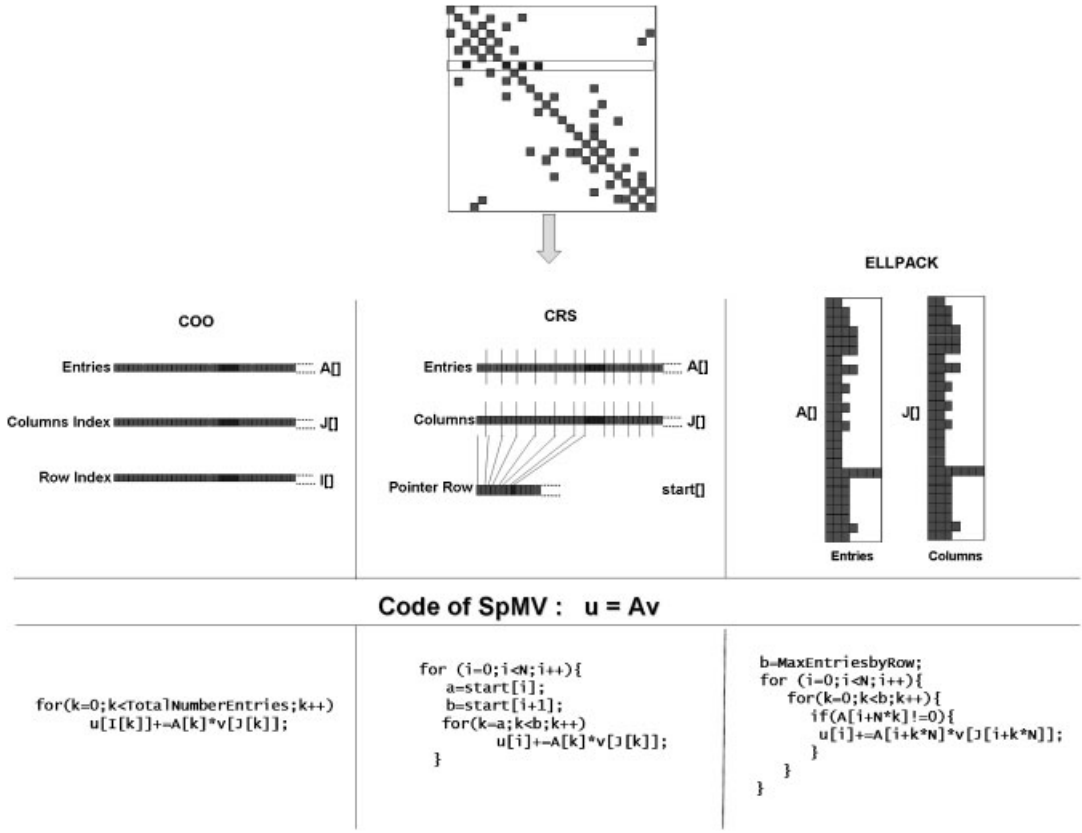


Figure 1. Different storage formats for sparse matrices and the corresponding codes to compute SpMV.

### 3.2. *Compressed row storage* (*CRS*)

Compressed row storage (CRS) is the most commonly known format to store sparse matrices on superscalar processors. Figure 1(middle) illustrates the CRS details. 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 out by row index.

The code to compute SpMV based on CRS can be seen in Figure 1(middle). There are several drawbacks that hamper the optimization of the performance of this code on superscalar architectures. First, the locality of access to 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 [10, 17]. Despite these drawbacks, several optimizations have made it possible to improve the performance of sparse computation on current processors [5, 18]. 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 fine grained parallelism on Intel processors. According to our experience, the SpMV can be accelerated with MKL over the standard implementation (and gcc compiler) up to $3\times$ factors.

### 3.3. *ELLPACK*

ELLPACK or ITPACK [16] 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



DOI: 10.1002/cpe

integer ($J[\ ]$), to save the column index of every entry. Both arrays are, at least, of dimension $N \times MaxEntriesbyRows$, where $N$ is the number of rows and *MaxEntriesbyRows* is the maximum number of nonzeros 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, as seen in Figure 1(right). 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 $0 \leq k < MaxEntriesbyRows$ 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 blocks of threads. Every block of threads can complete its computation without synchronization with others blocks, because every thread computes one element of the vector $u$ (i.e. the result of the SpMV operation), and there are no data dependencies in the computation of different elements of $u$.

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, as described in Figure 1(right). This is because to compute every $u[i]$, with $0 \leq i \leq N$, the $k$-loop must iterate until $k = MaxEntriesbyRows$ and the conditional branch is executed in every iteration; hence in order to reduce the memory access and activity of arithmetic units, the computation is penalized with $N \times MaxEntriesbyRows$ executions of the conditional branch.

### *3.4. Recent proposals for NVIDIA GPUs*

Recently, different proposals of kernels to compute SpMV on GPUs have been described and analyzed [13–15]. They can be classified into two groups according to their relationship with CRS or ELLPACK formats.

On the one hand, the kernel called CRS(vector) evaluated in [13] is based on a format which can be considered as derived from CRS format. This kernel computes every output vector element with the collaboration of the 32 threads of every warp. Thus, 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 for 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 [15]. 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 in the SpMV4GPU library [19], and hereinafter will be referred to in the same name. SpMV4GPU generally reaches better performance than CRS(vector). The computational load to compute $u[i]$ is proportional to $2rl[i]$ (i.e. the double of $i$th row entries number), if $T(\leq rl[i])$ threads collaborate to compute $u[i]$, every thread has a load proportional to $2rl[i]/T \geq 2$. However, if $rl[i] < T$, then, several threads are stalled when they are collaborating to compute $u[i]$, that is, there is a load imbalance. This penalty is more relevant if $T$ is higher and/or the number of rows with few entries is higher. Thus, CRS(vector) includes more imbalance than SpMV4GPU because $T_{\mathrm{CRS(vector)}}(=32) > T_{\mathrm{SpMV4GPU}}(=16)$, especially for matrices with low number of entries by rows, as analyzed in Section 5.



DOI: 10.1002/cpe

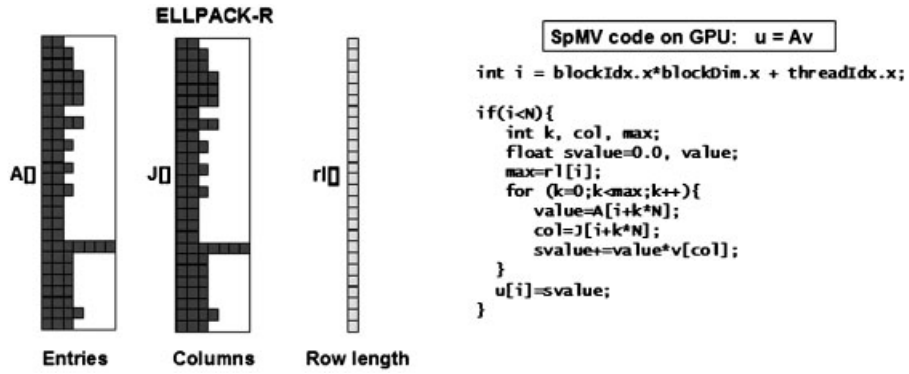


Figure 2. ELLPACK-R format and kernel to compute SpMV on GPUs.

On the other hand, the kernels related to the format called HYB (which stands for hybrid) proposed by [13] 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 to store it with HYB format, a preprocessing step is required in order to compute: (1) parameter *MaxEntriesbyRows*, (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 [13], and its corresponding parameter *MaxEntriesbyRows*$'$, and, finally, (4) two data structures to store $A$. *MaxEntriesbyRows*$'$ 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 that overfill some rows and storing them with COO format. Thus, the corresponding computation of SpMV based on GPU is split into several kernels related to the different formats, hopefully with an appropriate value of *MaxEntriesbyRows*$'$ 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 *MaxEntriesbyRows*$'$ does not reach optimum value.

## 4. ELLPACK-R, A FORMAT TO OPTIMIZE SpMV ON NVIDIA GPUs

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 $N \times MaxEntriesbyRows$ and an additional integer array called $rl[\,]$ of dimension $N$ (i.e. the number of rows) with the purpose of storing the actual length of every row, regardless of the number of the zero elements padded. As seen in Figure 2, the SpMV based on ELLPACK-R computed on GPUs takes advantage of

1. The exploitation of GPU architecture due to the coalesced global memory access, thanks to the column-major ordering used to store the matrix elements, and the non-synchronized execution between different blocks of threads, as the SpMV based on ELLPACK format.
2. Avoiding the penalties of the execution of conditional branches. Thanks to the inclusion of the array $rl[\,]$, the $k$-loop to compute SpMV based on ELLPACK-R does not include conditional branches and, additionally, the memory access and activity of arithmetic units are not overloaded with useless computation with padding zeros.
3. Homogeneous computing within the threads in 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



DOI: 10.1002/cpe

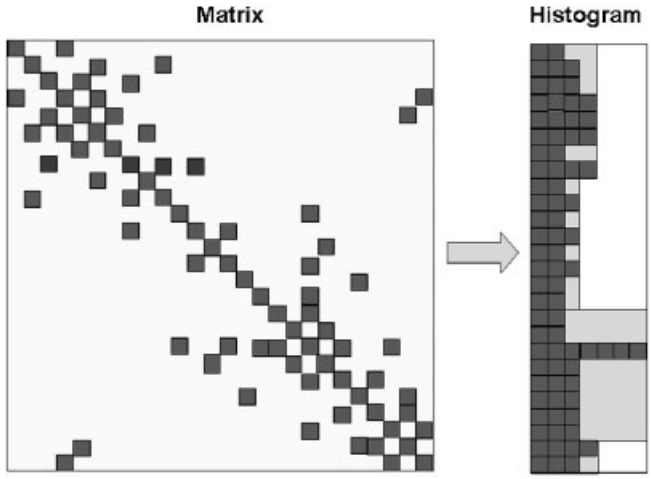


Figure 3. Histogram of a simple example with a tiny sparse matrix and assuming a hypothetic small warp of eight threads. The dark area is related to the runtimes of every thread belonging to every warp, and the gray area is related to the waiting times of the same thread.

other threads in the warp continue with the execution (see Figure 3). Furthermore, coalesced memory access is possible. This characteristic has a significant impact on the performance.

4. Reduction of useless computation and imbalance of the threads of one warp. If $x$ is the identifier of every thread, the $k$-loop reaches the maximum value of $k = rl[x] \leq MaxEntriesbyRows$ for specific threads into the warp. Then, the run-time of every warp is proportional to the maximum element of the sub-vector $rl[x]$ related to every warp, and it is not necessary that the $k$-loop for all threads into every warp reaches $k = MaxEntriesbyRows$, then, there is no useless iterations and the control of loops of this implementation is reduced comparing with SpMV based on ELLPACK. With the goal of illustrating this advantage of ELLPACK-R, Figure 3 shows an example of a histogram of a tiny matrix, and a hypothetic small warp of eight threads is considered. The runtime of every warp of eight threads is different and it is proportional to the longest row in the corresponding subset of rows of the matrix. Bearing in mind the kernel of SpMV with ELLPACK-R, the dark area is proportional to the runtime of every thread, and the gray area is proportional to the waiting time of every thread. Therefore, only the warps related to rows of very different lengths are penalized with longer waiting times, as can be seen in Figure 3.

## 5. EVALUATION

A comparative analysis of the performance of different kernels to compute SpMV on NVIDIA GPUs has been carried out in this work. The following formats to store the matrix have been evaluated: CRS, CRS(vector), SpMV4GPU, ELLPACK, HYB, and ELLPACK-R. This analysis is based on the runtimes measured on a GeForce GTX 295 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-zero elements (*Entries*), average number of entries per row ($Av$), the difference between the maximum number of entries in a row and $Av$ (*Max-Av*), standard deviation of the number of entries per row ($\sigma$), percentage of relative standard deviation of entries by row ($\%(\sigma/Av)$). Additionally, the parameter named $\%ELL$(HYB) is included in Table I, which denotes the percentage of entries stored with the ELLPACK format when HYB format is applied. 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



DOI: 10.1002/cpe

Table I. Set of test matrices and the characteristic parameters related with their distribution of entries on the rows: number of rows ($N$), total number of *Entries*, average number of entries per row ($Av$), the difference between maximum number of entries in a row and $Av$($Max-Av$), standard deviation of entries per row ($\sigma$), percentage of relative standard deviation of entries by row ($\%(\sigma/Av)$). Additionally, the column on the right shows the percentage of entries stored with the ELLPACK format when HYB format is applied ($\%ELL$(HYB)).

| Matrix | $N$ | *Entries* | $Av$ | $Max-Av$ | $\sigma$ | $\%\frac{\sigma}{Av}$ | $\%ELL$ (HYB) |
|---|---|---|---|---|---|---|---|
| qh1484 | 1484 | 6110 | 4.12 | 8.88 | 1.60 | 38.87 | 86.8 |
| dw2048 | 2048 | 10114 | 4.94 | 3.06 | 0.61 | 10.24 | 98.7 |
| rbs480a | 480 | 17087 | 35.60 | 0.40 | 0.49 | 1.38 | 100.0 |
| gemat12 | 4929 | 33111 | 6.72 | 37.28 | 3.01 | 44.79 | 90.7 |
| dw8192 | 8192 | 41746 | 5.10 | 2.90 | 0.61 | 12.05 | 96.9 |
| mhd3200a | 3200 | 68026 | 21.26 | 11.74 | 5.83 | 27.41 | 93.6 |
| e20r4000 | 4241 | 131556 | 31.02 | 30.98 | 15.40 | 49.63 | 86.9 |
| bcsstk24 | 3562 | 159910 | 45.00 | 12.00 | 11.49 | 25.59 | 100.0 |
| mac_econ | 206500 | 1273389 | 6.17 | 37.83 | 4.43 | 71.92 | 81.1 |
| qcd5_4 | 49152 | 1916928 | 39.00 | 1.00 | 0.00 | 0.00 | 100.0 |
| mc2depi | 525825 | 2100225 | 3.99 | 0.01 | 0.08 | 1.91 | 100.0 |
| rma10 | 46835 | 2374001 | 50.69 | 94.31 | 27.78 | 56.11 | 81.3 |
| cop20k_A | 121192 | 2624331 | 21.65 | 59.35 | 13.79 | 63.70 | 82.8 |
| wbp128 | 16384 | 3933095 | 240.05 | 15.95 | 34.78 | 14.49 | 100.0 |
| dense2 | 2000 | 4000000 | 2000.00 | 0.00 | 0.00 | 0.00 | 100.0 |
| cant | 62451 | 4007383 | 64.17 | 13.83 | 14.06 | 21.90 | 99.7 |
| pdb1HYS | 36417 | 4344765 | 119.30 | 84.70 | 31.86 | 26.70 | 96.0 |
| consph | 83334 | 6010480 | 72.13 | 8.87 | 19.08 | 26.45 | 100.0 |
| shipsec1 | 140874 | 7813404 | 55.46 | 46.54 | 11.07 | 19.96 | 93.1 |
| pwtk | 217918 | 11634424 | 53.39 | 127.61 | 4.74 | 8.88 | 99.4 |
| wbp256 | 65536 | 31413932 | 479.34 | 32.66 | 70.53 | 14.71 | 100.0 |

number of entries by row of the matrices. Most of the considered matrices belong to collections of the Matrix Market repository [20]. 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.

All kernels have been evaluated using the texture memory. This memory is bound to the global memory and plays the role of a cache level within the memory hierarchy, and its use improves the performance if there is data reuse. The programming interface, CUDA, allows the programmer to specify which variables are to be stored in the texture cache within the memory hierarchy. 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. Our experience shows that this use of texture memory improves the performance of SpMV, as also shown in [11].

An experimental evaluation of the coalesced memory access has shown that it is correlated with the performance achieved by the different approaches.

Figure 4 shows the performance (GFLOPs) of the SpMV kernels based on the formats that have been evaluated: CRS, CRS(vector), SpMV4GPU, ELLPACK, HYB, and ELLPACK-R. 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 coalesced.
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. $\%(\sigma/Av)$ is lower. SpMV4GPU reaches higher performance than CRS(vector) because



DOI: 10.1002/cpe

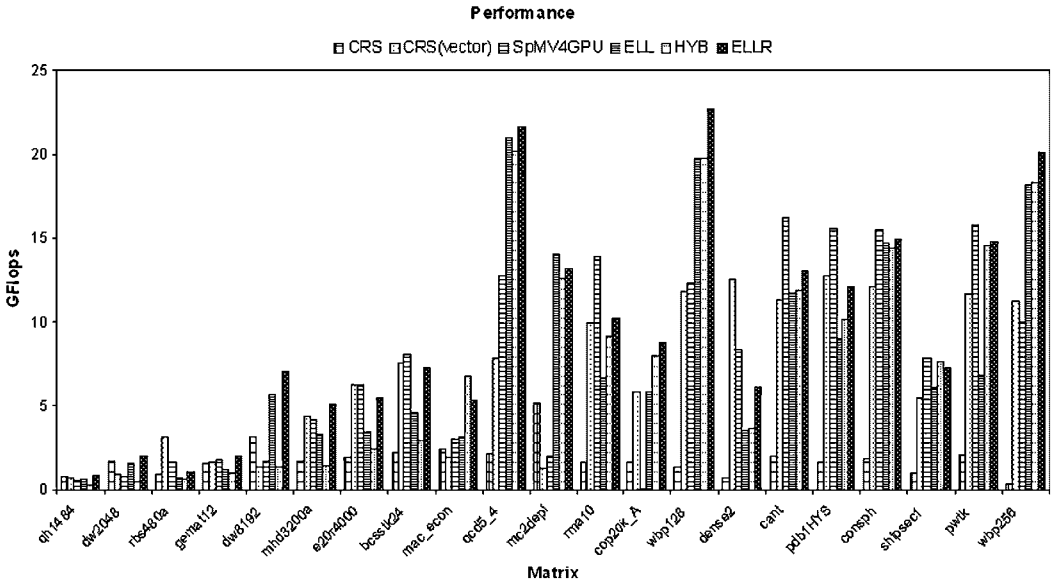


Figure 4. Performance of SpMV based on different formats on GPU GeForce GTX 295 with the set of test matrices, using the texture cache memory.

it better exploits (1) the power of threads, since the load balance is better achieved by $T_{\text{SpMV4GPU}}(=16)$ threads than $T_{\text{CRS(vector)}}(=32)$ collaborating to compute every $u$ element and (2) the coalesced memory access because the memory alignment requirements are met. Although SpMV4GPU gets the best performance for a subset of matrices with higher and regular filling of rows (e20r4000, bcsstk24, rma10, cant, pdb1HYS, consph, shipsec1, pwtk), it however achieves poor performance for another subset of tested matrices (qcd5_4, mc2depi, cop20k_A, wbp128, wbp256). It can thus be concluded that the SpMV based on SpMV4GPU outperforms CRS(vector), but the performance is dependent on the matrix pattern.

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. Thus, for matrices with a high value of the parameter $Max - Av$, such as rma10, pdb1HYS, shipsec1, pwtk (see Table I), the useless computation is more relevant because only few threads complete the inner k-loop with useful computation, due to only few rows including $b =$ *MaxEntriesbyRows* entries (see Figure 1). Then, for these matrices the implementation of SpMV based on format ELLPACK-R reaches better performance since it does not include these penalties.
5. The performance obtained by HYB is, in general, higher than that for the four previous formats, but it is remarkable that its poorer results for smaller matrices is 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 %, as shown by the parameter $\%ELL$(HYB) of Table I.
6. Finally, ELLPACK-R achieves the best average performance for the set of matrices considered in this work. In particular, it achieves the highest performance with matrices of high dimensions and higher values of parameters $Max - Av$, $\sigma$ and $\%(\sigma/Av)$. However, the performance is lower with matrices of small dimensions. ELLPACK or HYB reach slightly higher performance than ELLPACK-R with a few matrices of higher dimension (mc2depi, shipsec1), with small $\sigma$. ELLPACK-R achieves the more regular performance if the whole set of test matrices is considered, as clearly shown by the average performance in Figure 5.



DOI: 10.1002/cpe

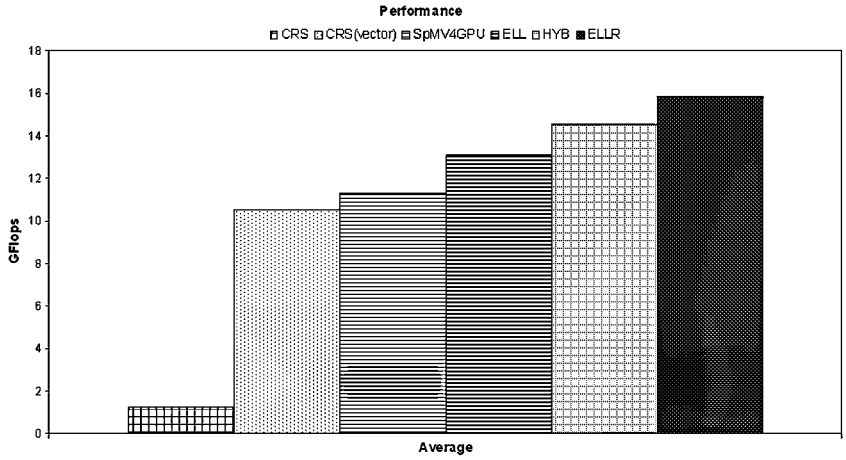


Figure 5. Average performance weighted according to the number of entries of test matrices, when different SpMV kernels are executed on GPU GeForce GTX 295 with the set of test matrices.

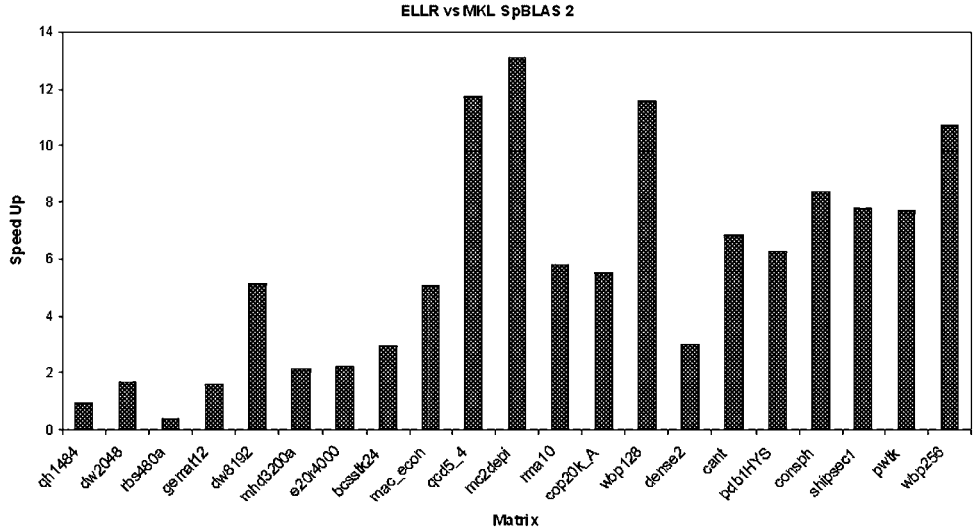


Figure 6. Speed-up of SpMV on GPU GeForce GTX 295 for the set of test matrices in Table I, taking as a reference the runtimes of SpMV on a Intel Core 2 Duo E8400. The storage format that provided the best performance for GPU was used, ELLPACK-R, and the MKL implementation of SpMV for The superscalar core was considered.

Figure 5 plots the average performance weighted according to the number of entries of the test matrices and obtained for the six formats evaluated. As seen, the best average performance is got by ELLPACK-R, followed by HYB and ELLPACK, and the worst average performance is obtained by CRS, CRS(vector), and SpMV4GPU. Therefore, these results confirm that ELLPACK-R is superior to the sparse matrix storage formats used thus far. The algorithm for computing SpMV using ELLPACK-R includes neither flow control instructions that serialize the execution of the warp, nor complex pre-processing steps to reorder the matrix rows; moreover, it allows coalesced matrix data access. Owing to these characteristics the performance of SpMV based on ELLPACK-R is predictably higher than that based on the other formats for sparse matrices in general. In conclusion, the simplicity of the SpMV computation based on ELLPACK-R allows full exploitation of the GPU architecture and its computing power.

The key of the success of GPUs in high performance computing comes from the outstanding speedup factors in comparison with standard computers or even clusters of workstations. In order to



DOI: 10.1002/cpe

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 in Table I. Although MKL may not be portable to other architectures, we decided to take its implementation of SpMV as a reference because it turns out to be one of the best optimized SpMV implementations for modern superscalar processors. For the GPU GeForce GTX 295, we used the ELLPACK-R format, which is the best for the GPU according to the results presented above. Figure 6 shows the speedup factors obtained for the SpMV operation on the GPU against one superscalar core, for all the test matrices considered in this work. The speedup ranges from a modest $2\times$ factor to 13, $1\times$ factor. The plot shows that the speedup depends on the matrix pattern, though in general it increases with the number of non-zero entries. In view of these results, we can conclude that the GPU turns out to be an excellent accelerator of SpMV.

## 6. CONCLUSIONS

In this paper a new approach to compute the sparse matrix vector on GPUs has been proposed and evaluated, ELLPACK-R. The simplicity of the SpMV implementation based on ELLPACK-R makes it well suited for GPU computing. The comparative evaluation with other proposed formats has shown that the average performance achieved by ELLPACK-R is the best after an extensive study on a set of representative test matrices. Therefore, ELLPACK-R has proven to be superior to the other approaches used thus far, because it combines several advantages to exploit the GPU architecture: (1) the improvement of memory management by means of the coalesced memory access to read the sparse matrix and (2) the reduction of useless computation and imbalance of the threads, since the actual number of non-zero entries in every row of the sparse matrix is taken into account. Moreover, the fact that this approach for SpMV does not require any preprocessing step makes it especially attractive to be integrated on sparse matrix libraries currently available. A comparison of the GPU implementation of SpMV based on ELLPACK-R on a GeForce GTX 295 has revealed that acceleration factors of up to $13\times$ can be achieved in comparison to optimized implementations of SpMV which exploit state-of-the-art superscalar processors. Our future plans include improvement of the performance of SpMV by fulfilling the memory alignment requirements and the use of several threads collaborating in the computation related to every row of the matrix. 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.

ACKNOWLEDGEMENTS

The authors thank Dr Enrique Quintana Ortí for helpful suggestions and Dr José Antonio Martínez García for technical support and discussions. This work has been partially supported by Spanish Ministry of Science and Innovation under grant number TIN2008-01117 and Junta de Andalucía under grant numbers P06-TIC01426 and P08-TIC03518.

REFERENCES

1. Bisseling RH. *Parallel Scientific Computation*. Oxford University Press: Oxford, 2004.
2. Ogielski AT, Aiello W. Sparse matrix computations on parallel processor arrays. *SIAM Journal on Scientific Computing* 1993; **14**:519–530.
3. Toledo S. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development* 1997; **41**(6):711–725.
4. Mellor-Crummey J, Garvin J. Optimizing sparse matrix-vector product computations using unroll and jam. *International Journal of High Performance Computing Applications* 2004; **18**:225–236.
5. 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.
6. Lawson C, Hanson R, Kincaid D, Krogh F. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software* 1979; **5**:308–325.



DOI: 10.1002/cpe

7. Baldeschwieler J, Blumofe R, Brewer E. ATLAS: An infrastructure for global computing. *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, Connemara, Ireland, 1996.
8. NVIDIA. CUDA CUBLAS Library. PG-00000-002.V2.1 September 2008. Available at: http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUBLAS_Library_2.1.pdf [June 2009].
9. Vazquez F, Garzon EM, Fernandez JJ. A matrix approach to tomographic reconstruction and its implementation on GPUs. *Journal of Structural Biology* 2010; **170**:146–151.
10. Kurzak J, Alvaro W, Dongarra J. Optimizing matrix multiplication for a short-vector SIMD architecture—CELL processor. *Parallel Computing* 2009; **35**(3):138–150.
11. NVIDIA. CUDA Programming guide, Version 2.3, August 2009. Available at: http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf [August 2009].
12. Kronos Group. OpenCL—The open standard for parallel programming of heterogeneous systems. Available at: http://www.khronos.org/developers/library/overview/opencl_overview.pdf [June 2009].
13. Bell N, Garland M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proceedings of the Conference on High Performance Computing Networking*, *Storage and Analysis*, vol. 18, 2009. Available at: http://www.nvidia.com/object/nvidia_research_pub_013.html [December 2009].
14. Buatois L, Caumon G, Lvy 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.
15. Baskaran MM, Bordawekar R. Optimizing sparse matrix-vector multiplication on GPUs. *IBM Research Report RC24704*, April 2009.
16. Kincaid DR, Oppe TC, Young DM. ITPACKV 2D User's Guide. *CNA-232*, 1989. Available at: http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/.
17. Goumas G, Kourtis K, Anastopoulos N, Karakasis V, Koziris N. Performance evaluation of the sparse matrix-vector multiplication on modern architectures. *Journal of Supercomputing* 2009; **50**:36–77.
18. Intel. Math Kernel Library. Reference Manual. Available at: http://software.intel.com/sites/products/documentation/hpc/mkl/mklman.pdf [June 2009].
19. Baskaran MM, Bordawekar R. Sparse matrix-vector multiplication toolkit for graphics processing units, April 2009. Available at: http://www.alphaworks.ibm.com/tech/spmv4gpu [June 2009].
20. *The Matrix Market*. Available at: http://math.nist.gov/MatrixMarket [June 2009].



DOI: 10.1002/cpe