The sparse matrix vector product on GPUs

F. Vázquez, E. M. Garzón , J. A. Martínez, J. J. Fernández
{f.vazquez, gmartin, jamartine, jjfdez}@ual.es
Dpt Computer Architecture and Electronics.
University of Almeria

June 14, 2009

Abstract

The sparse matrix vector product (SpMV) is a paramount operation in engineering and scientific computing and, hence, has been a subject of intense research for long. 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. The Graphics Processing Units (GPUs) have recently emerged as platforms that yield outstanding acceleration factors. Currently, SpMV implementations for NVIDIA-GPUs have already appeared on the scene. This work proposes and evaluates a new implementation of SpMV for GPUs based on a new matrix storage format, called ELLPACK-R, and compares it against a variety of formats proposed elsewhere. The most important qualities of this new format is that (1) no preprocessing of the sparse matrix is required, and (2) the resulting SpMV algorithm is very regular. The comparative evaluation of this new SpMV approach has been carried out based on a representative set of test matrices. The results show that the SpMV approach based on ELLPACK-R turns out to be superior to the previous strategies used so far. Moreover, a comparison with standard state-of-the-art superscalar processors reveals that significant speedup factors are achieved with GPUs.

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 operation in computational sciences is supported by the constant effort devoted to optimise the computation of MV for the processors at the time, which range from the early computers in the seventies to the last modern multi-core architectures [2, 3, 4, 5]. In that sense, the fact that MV is a routine of Level 2 in BLAS (Basic Linear Algebra Subroutines) is remarkable because the BLAS library has constantly been improved

and optimized as the computer architectures have evolved [6, 7, 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. 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 [9]. 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.

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) [14] and OpenCL [10], 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 [11, 12, 13]. The aim of this work is to design and analyse GPU computing approaches of SpMV. This work covers a variety of formats to store the sparse matrix in order to explore the best possible use of the GPU for a variety of algorithmic parameters. A proposal for a new storage format is presented which proves to outperform the most common and efficient formats for SpMV used so far.

Next, Section 2 summarises 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 optimise SpMV on GPUs. Section 4

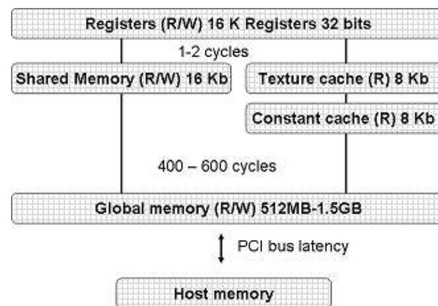

Figure 1: Different access times and sizes of GPU Memories

introduces a new format suitable for computation of SpMV on GPUs. In Section 5 the performance measured on a NVIDIA Geforce GTX 295 with a wide 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 summarises the main conclusions.

2 Computational keys to exploit GPUs

Compute Unified Device Architecture (CUDA) provides a set of extensions to standard ANSI C for programming NVIDIA GPUs. It supports heterogeneous computation where applications use both the CPU and 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 [14]. 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 eight (NVIDIA Tesla C870) to thirty in modern GPUs (NVIDIA Geforce GTX 295). The SPs in a 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 [14].

To develop codes for 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. The GPU architecture allows the host to issue 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 by 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 control instruction can cause the threads of the same warp to diverge, that is, to follow different execution paths. If this occurs, the different executions paths have to be serialized, increasing the total number of instructions executed for this warp [14].

Another key to take advantage of 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, as illustrated in Figure 1. 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 access to the global memory can be performed in parallel by all threads of a half-warp (16 threads), which is accelerated only for specific coalesced memory access patterns [14]. Hence the ordering of the data access chosen in an algorithm may have

significant performance effects during GPU memory operations.

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 optimise the GPU performance, the programmer has to consider two main goals: (1) to balance the computation of the sets of threads, and (2) to optimise the data access through the memory hierarchy. Specifically, to optimise SpMV on GPUs, both goals have to be taken into account in 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

3.1 Coordinate storage

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, of floating-point numbers (hereafter referred to as floats), contains the non-zero entries. The other two arrays, of integer numbers, contain the corresponding row and column indices for each non-zero entry. The performance of SpMV may be penalised by COO because it does not implicitly include the information about the ordering of the coordinates.

3.2 Compressed Row Storage (CRS) and some variants

Compressed Row Storage (CRS) is the most extended format to store sparse matrices on superscalar processors. Figure 2(left) 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, which stores the pointers to the beginning of every row in $A[\,]$ and $j[\,]$.

The code to compute SpMV based on CRS can be seen on Figure 2(left). There are several drawbacks that hamper the optimization of the performance of this code on superscalar architectures. First, the locality to 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 [9].

The Incremental Compressed Row Storage (ICRS) format [1] is a variant of CRS where the location of non-zero elements is encoded as a one-dimensional index. The underlying motivation is the following: if the entries within a row are ordered by increasing column index, the one-dimensional indices form a

monotonically increasing sequence. ICRS consists of two arrays: (1) $A[\,]$ array of floats of dimension Nz, which stores the entries; and (2) $inc[\,]$ array of integers of dimension Nz, which stores the increments of indexes of the vector $v[\,]$. More details about ICRS can be obtained in [1]. The performance obtained by ICRS is similar to CRS according to our experience with the set of matrices considered.

We have designed and evaluated a new format, called CRSN (CRS with Negative marks), which is a variant of CRS. The components of CRSN are illustrated in Figure 2(center). CRSN only requires two arrays of dimension Nz, which are equivalent to the arrays $A[\,]$ and $j[\,]$ of the original CRS. However, the beginning of every row is marked with a negative column index in $j[\,]$, and the corresponding code to compute SpMV includes one loop that contains a conditional branch. The performance obtained with CRSN is slightly better than with CRS and ICRS on superscalar cores included in current processors, such as Intel Core 2 Duo, Intel Xeon Quad Core Clovertown and AMD Opteron Quad Core according to our experience. Specifically, better performance is got on Intel Core 2 Duo E8400. In Section 5, CRSN on one core of Intel Core 2 Duo E8400 is considered as a reference, with the purpose of comparing the performance of the SpMV computation on two architectures, GPU and one superscalar core.

3.3 ELLPACK

ELLPACK or ITPACK [15] 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, to save the entries, and one integer, to save the columns of every entry. Both arrays are of dimension at least $N \times MaxEntriesbyRows$, where N is the number of rows and $MaxEntriesbyRows$ 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, as seen in Figure 2(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. However, if the percentage of zeros is high in the ELLPACK data structure and there is a very irregular location of entries in different rows, then the performance decreases.

4 ELLPACK-R, a format to optimize SpMV on GPUs

ELLPACK-R is a variant of the ELLPACK format. ELLPACK-R consists of two arrays, $A[\,]$ (float) and $j[\,]$ (integer) of dimension $N \times MaxEntriesbyRows$; and, moreover, an additional integer array called $rl[\,]$ of dimension N (i.e. the

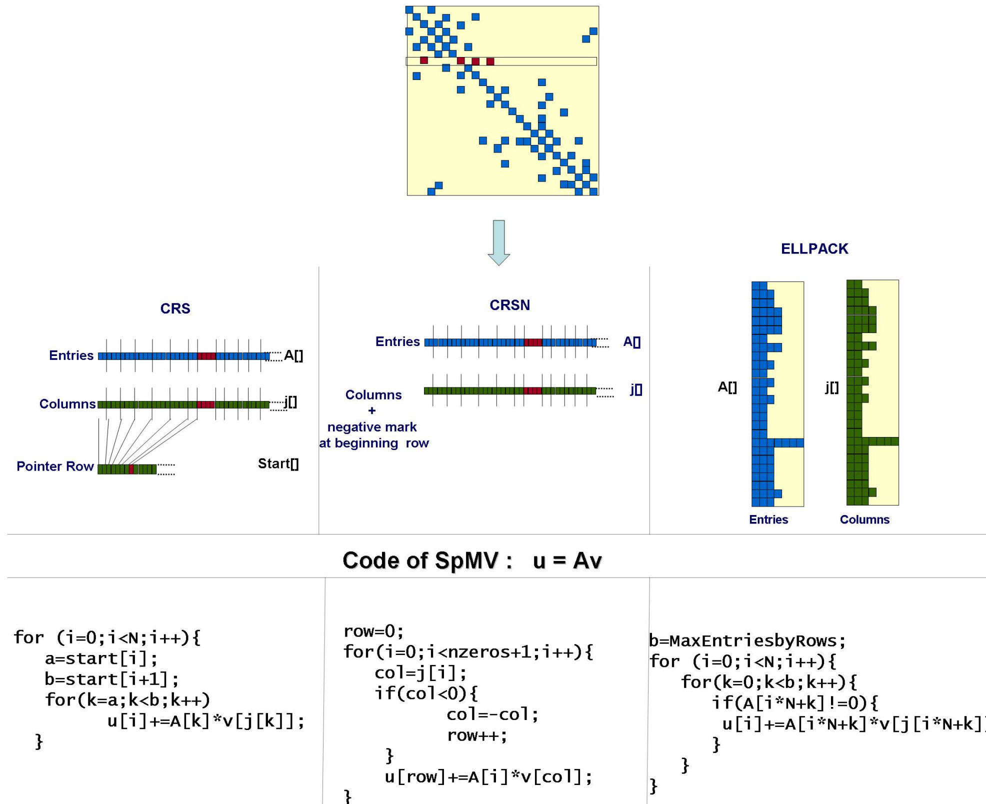

Figure 2: Different storage formats for sparse matrices and the corresponding codes to compute SpMV

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. An important point is the fact that the arrays store their elements in column-major order. As seen in Figure 3, these data structures take advantage of:

(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: $A[x + i * N]$ with $\{\ 0 \leq i < rl[x]\ \}$ where i is the column index and $rl[x]$ is the total number of non-zeros in row x. 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 dependences in the

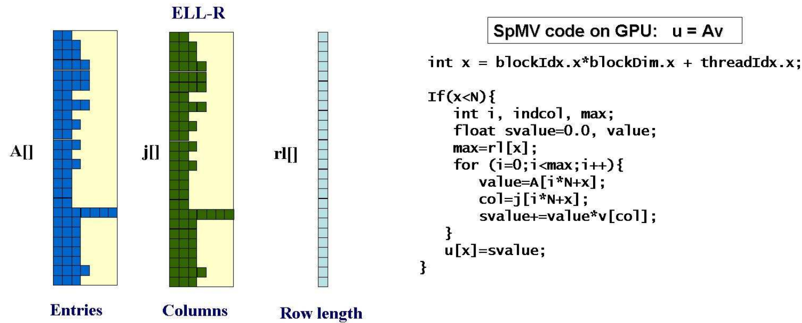

Figure 3: ELLPACK-R format and kernel to compute SpMV on GPUs

computation of different elements of u.

(3) The reduction of the waiting time or unbalance between threads of one warp. Figure 4 shows an example of histogram of a tiny matrix, and a hypothetic small warp of eight threads is considered with the goal of illustrating the advantage of ELLPACK-R. The computational load 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 grey area is proportional to the waiting time of every thread. Therefore, only the warps related to rows of very different length are penalised with longer waiting times, as can be seen in Figure 4.

(4) 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 other threads in the warp continue with the execution (see Figure 4). Furthermore, coalesced memory access is possible. This characteristic has a significant impact on the performance.

Recently, different proposals of kernels to compute SpMV have been described and analysed [11, 12, 13]. The kernels related to the format called HYB (which stands for hybrid) proposed by [11] seem to yield the best performance on GPUs so far. This format combines the ELLPACK and COO formats for different sets of rows. However, this format previously requires a preprocessing step consisting of reordering the rows in order to get a better performance. This preprocessing of the matrix is a drawback that may produce a significant penalization due to the calls/returns to/from kernels, especially in matrices where large sets of rows have to be divided and reordered. Other kernel called CRS(vector) has also been evaluated in [11]. This kernel computes every output vector element with the collaboration of the 32 threads of every warp. So, every

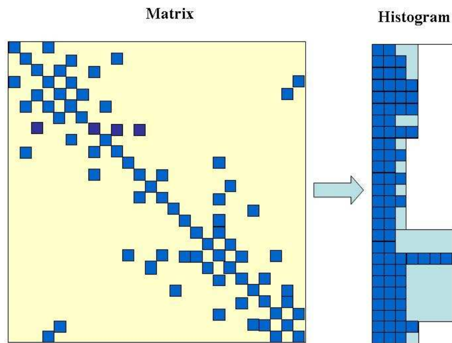

Figure 4: Histogram of a simple example with a tiny sparse matrix and assuming a hypothetic small warp of eight threads. The dark area is related to the runtimes of every thread belonging to every warp, and the grey area is related to the waiting times of the same thread.

warp computes the float products related to the entries of every row, followed by a parallel reduction in shared memory in order to obtain the final result of output vector element.

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), ELLPACK, ELLPACK-R and HYB. This analysis is based on the run-times measured on a GeForce GTX 295 with a set of test sparse matrices from different disciplines of science and engineering. Table 1 summarizes the test matrices used in this work and their important characteristics, such as the dimensions, the number of non-zero entries, etc. Most considered matrices belong to collections of the Matrix Market repository [16]. 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 [14].

Figure 5 shows the performance (GFLOPs) of the SpMV kernels based on the formats that have been evaluated: CRS, CRS(vector), ELLPACK, ELLPACK-R and HYB. The results shown in that figure allow us to highlight the following major points: (1) the performance obtained by most formats increases with the number of non-zero entries in the matrix; (2) in general, the CRS format yields the poorest performance because the pattern of memory access is not coalescent; (3) the CRS(vector) format achieves better performance than CRS in most cases

Table 1: Set of test matrices

Matrix	N	Entries	Type	Application area
qh1484	1484	6.110	Gen.	Power systems simulations
dw2048	2048	10.114	Gen.	Electrical engineering
rbs480a	480	17.087	Gen.	Robotic control
gemat12	4929	33.111	Gen.	Power flow modeling
dw8192	8192	41.746	Gen.	Electrical engineering
mhd3200a	3200	68.026	Gen.	Plasma physics
bcsstk24	3562	81.736	Sym.	Structural engineering
e20r4000	4241	131.556	Gen.	Fluid dynamics
mac_econ	206500	1.273.389	Gen.	Economics
cop20k_A	121192	1.362.087	Sym.	FEM/Accelerator
qcd5_4	49152	1.916.928	Gen.	QCD
cant	62451	2.034.917	Sym.	FEM/Cantiveler
mc2depi	525825	2.100.225	Gen.	Epidemiology
pdb1HYS	36417	2.190.591	Sym.	Biocomputation
rma10	46835	2.374.001	Gen.	FEM/Harbor
consph	83334	3.046.907	Sym.	FEM/Spheres
wbp128	16384	3.933.095	Gen.	Tomographic reconstruction
shipsec1	140874	3.977.139	Sym.	FEM/Ship
dense2	2000	4.000.000	Gen.	Dense
pwtk	217918	5.926.171	Gen.	Fluid dynamics
wbp256	65536	31.413.932	Gen.	Tomographic reconstruction

(even for matrices with high number of non-zero entries by row or nearly dense), despite the fact that a coalesced matrix data access is not possible with this format either; (4) in general, ELLPACK outperforms both CRS-based formats, however its computation is penalised for some particular matrices, mainly due to the divergence of the warps when the matrix histogram includes rows with very uneven length; (5) The performance obtained by HYB is, in general, higher than that for the three previous formats, but it is remarkable its poorer results for smaller matrices due to the penalty introduced by the preprocessing step; (6) finally, ELLPACK-R clearly achieves the best performance for most matrices considered in this work.

Figure 6 plots the average performance obtained for the five 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 and CRS(vector). 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 neither includes flow control instructions that serialise the execution of a warp of 32 threads, nor complex pre-processing steps to reorder the matrix rows; moreover, it allows coalesced matrix data access. In conclusion, the simplicity of the SpMV computation based on ELLPACK-R allows full exploitation of the GPU architecture and its computing power.

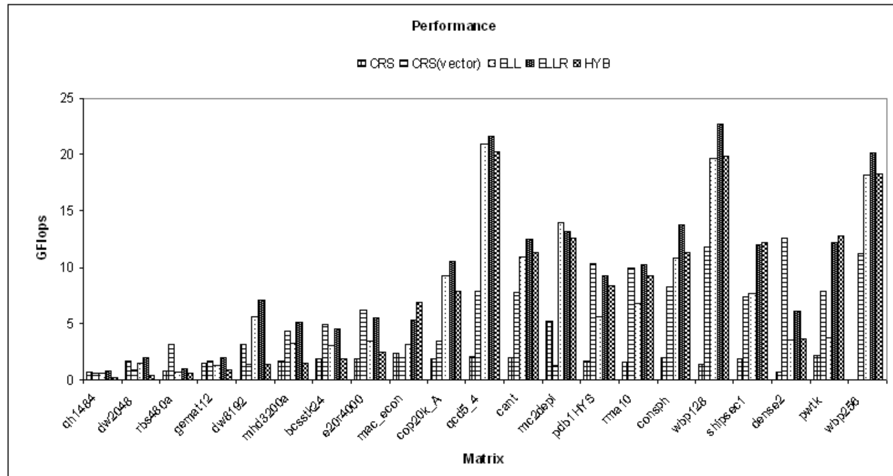

Figure 5: Performance of SpMV based on different formats on GPU GeForce GTX 295 with the set of test matrices, using the texture cache memory

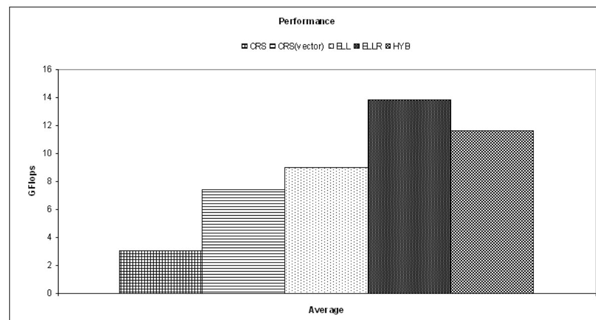

Figure 6: Average performance of SpMV on GPU GeForce GTX 295 and the set of test matrices

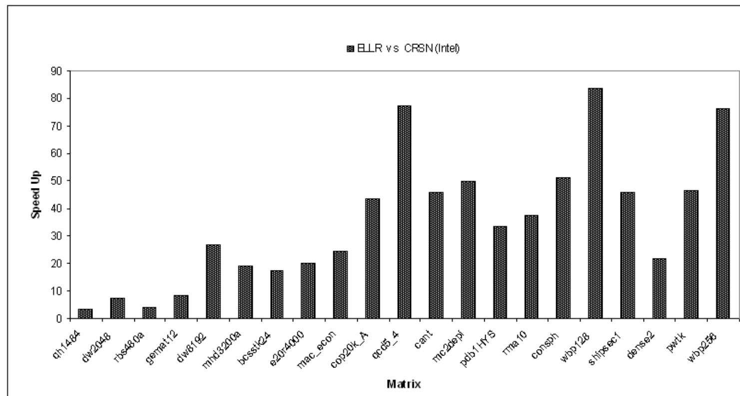

Figure 7: Speed-up of SpMV on GPU GeForce GTX 295 for the set of test matrices in Table 1, taking as a reference the runtimes of SpMV on a Intel Core 2 Duo E8400. The storage format that provided the best performance for each platform was used, ELLPACK-R for the GPU and CRSN for the superscalar core.

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 estimate the net gain provided by GPUs in the SpMV computation, we have implemented the 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 1. For the superscalar implementation, we chose the CRSN format as it provided the best performance for this platform (results not shown here). 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 7 shows the speedup factors obtained for the SpMV operation on the GPU against the superscalar core for all the test matrices considered in this work. The speedup ranges from a modest 5× factor to an exceptional 80× 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 proposals has shown that the average performance achieved by ELLPACK-R is the best after an extensive study on a wide set of test matrices. Therefore, ELLPACK-R has proven to be superior

to the other approaches used thus far. Moreover, the fact that this approach for SpMV does not require any preprocessing step makes it specially 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 80× can be achieved in comparison to 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.

Acknowledgements

This work has been partially supported by grants P06-TIC01426 and TIN2008-01117.

References

[1] *R.H. Bisseling* PARALLEL SCIENTIFIC COMPUTATION, Oxford Univ. Press, 2004.

[2] A.T. OGIELSKI, W. AIELLO *Sparse matrix computations on parallel processor arrays* SIAM Journal on Scientific Computing,**14** (1993) 519–530.

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

[4] J. MELLOR-CRUMMEY, J. GARVIN *Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam* Intl. J. High Performance Comput. App.,**18** (2004) 225–236

[5] S. WILLIAMS, L. OLIKER, R. VUDUC, J. SHALF, K. YELICK, J. DEMMEL *Optimization of sparse matrix-vector multiplication on emerging multicore platforms* Parallel Computing,**35 (3)** (2009) 178–194

[6] C. LAWSON, R. HANSON, D. KINCAID, F. KROGH, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Trans. Mathematical Software, **5** (1979) 308–325.

[7] J. BALDESCHWIELER AND R. BLUMOFE AND E. BREWER *ATLAS: An Infrastructure for Global Computing* In Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, (1996).

[8] NVIDIA, CUDA CUBLAS Library. PG-00000-002.V2.1 September, 2008. `http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/CUBLAS_Library_2.1.pdf`

[9] J. KURZAK, W. ALVARO AND J. DONGARRA *Optimizing matrix multiplication for a short-vector SIMD architecture - CELL processor* Parallel Computing,**35 (3)** (2009) 138–150

[10] KRONOS GROUP, *OpenCL - The open standard for parallel programming of heterogeneous systems* `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`

[11] N. BELL, M. GARLAND *Efficient Sparse Matrix-Vector Multiplication on CUDA.* NVIDIA Technical Report NVR-2008-004, December 2008.

[12] L. Buatois, G. Caumon, B. Lvy, *Concurrent number cruncher - A GPU implementation of a general sparse linear solver*, International Journal of Parallel, Emergent and Distributed Systems, to appear.

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

[14] NVIDIA, *CUDA Programming guide. Version 1.1*, April, 2009 `http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf`

[15] D.R. Kincaid, T.C. Oppe, D.M. Young, *ITPACKV 2D User's Guide.* CNA-232 1989 `http://rene.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/`

[16] *The Matrix Market* `http://math.nist.gov/MatrixMarket`