The Computer Journal Advance Access published May 1, 2013

doi:10.1093/comjnl/bxt038

FastSpMM: An Efficient Library for Sparse Matrix Matrix Product on GPUs

GLORIA ORTEGA[1,*], FRANCISCO VÁZQUEZ[1], INMACULADA GARCÍA[2] AND ESTER M. GARZÓN[1]

[1]*Department of Informatics, University of Almería, Agrifood Campus of International Excellence (CeiA3) Ctra. Sacramento s/n, 04120 Almería, Spain*
[2]*Department of Computer Architecture, University of Málaga, 29071 Málaga, Spain*
[*]*Corresponding author: gloriaortega@ual.es*

Sparse matrix matrix (SpMM) multiplication is involved in a wide range of scientific and technical applications. The computational requirements for this kind of operation are enormous, especially for large matrices. This paper analyzes and evaluates a method to efficiently compute the SpMM product in a computing environment that includes graphics processing units (GPUs). Some libraries to compute this matricial operation can be found in the literature. However, our strategy (FastSpMM) outperforms the existing approaches because it combines the use of the ELLPACK-R storage format with the exploitation of the high ratio computation/memory access of the SpMM operation and the overlapping of CPU–GPU communications/computations by Compute Unified Device Architecture streaming computation. In this work, FastSpMM is described and its performance evaluated with regard to the CUSPARSE library (supplied by NVIDIA), which also includes routines to compute SpMM on GPUs. Experimental evaluations based on a representative set of test matrices show that, in terms of performance, FastSpMM outperforms the CUSPARSE routine as well as the implementation of the SpMM as a set of sparse matrix vector products.

Keywords: SpMM; streaming computation; GPU computing; matricial operations

Received 25 September 2012; revised 5 April 2013
Handling editor: Fionn Murtagh

1. INTRODUCTION

Sparse matrices are present in many applications from different science and engineering fields, such as fluid dynamics [1], biology [2, 3], structural engineering, Optical Diffraction Tomography [4] and so on. Many problems require the resolution of large partial differential equations and large linear systems of equations where operations involving sparse matrices have large computational requirements [5–8]. In this context, high performance computing systems are required for implementing an efficient approach of these matricial operations. It is noteworthy that the optimization of this kind of computation is a challenge because of the strong irregularity of the sparse operations from the computational point of view. The relevance of this kind of operations in computational science is supported by the constant effort devoted to optimizing the computation of sparse matrices on the existing most advanced processors. An important part of this effort is focused on the design of appropriate data formats to store the sparse matrices [5, 9].

Currently, graphics processing units (GPUs) offer massive parallelism for scientific computations. In the last few years, owing to the availability of application programming interfaces, such as Compute Unified Device Architecture (CUDA) [10] and OpenCL [11], the use of GPUs for general purpose applications has grown enormously. CUDA and OpenCL actually facilitate the development of applications targeted at GPUs. This is why GPUs have emerged as new computing platforms that offer massive parallelism and provide high performance/cost ratio for scientific computations.

An approach to facilitate GPU programming is based on the use of basic routines or libraries for computing the most used operations in scientific and practical applications. These

operations are optimally accelerated by GPUs. In this line, NVIDIA supplies a wide set of routines related to matrix computation such as CuBLAS for dense vectors and matrices [12] and CUSPARSE for sparse matrices [13].

The sparse matrix vector product (SpMV) is the most studied example of operation with sparse matrices. Recently, several implementations of SpMV developed with CUDA have been included in libraries to facilitate the GPU programming [14–17]. Devising GPU-friendly matrix storage formats has also been a key issue for these implementations [18]. An example of this is the ELLR-T algorithm, which relies on the ELLPACK-R storage format for sparse matrices [9, 19]. This format is a GPU-friendly variant of one previously designed for vector architectures, ELLPACK [20]. In [19], an extensive performance evaluation of ELLR-T using a representative set of test matrices has been reported. The comparative study has drawn the conclusion that ELLR-T outperforms the most common routines for SpMV on GPUs used so far. An additional advantage of ELLR-T is related to the model that allows one to optimally configure it according to the particular combination of input sparse matrix/GPU architecture [14].

In this work, our interest is focused on a different and more complex matrix operation which includes sparse and dense matrices, the sparse matrix matrix (SpMM) product. Our goal is the optimization of the SpMM code and its implementation on GPUs. This kind of matricial operation is involved in very relevant applications. For example, in tomography the reconstruction methods weighted back projection (WBP) and simultaneous iterative reconstruction technique can be expressed in terms of this kind of matrix product [2, 3]. SpMM computes the matrix $C = A \cdot B$, where A is a sparse matrix, B is a dense matrix and consequently C is also dense. This operation is classified as a level 3 routine for sparse matrix computation [13]. This kind of routines are able to efficiently exploit the instruction level parallelism (ILP) on modern architectures [21]. The computational advantages of this kind of routines in terms of performance are well known [22]. The level 3 BLAS has been implemented on different target architectures for dense matrices [12, 23]. However, few references of SpMM implementations on GPUs can be found. For example, CUSPARSE library supplies one sparse level 3 function which computes $C = \alpha \cdot A \cdot B + \beta \cdot C$, where α, β are scalars; it is available for real or complex matrices [13] but its performance on GPUs is poor.

The goal of this work is to analyze strategies intended to improve the performance of SpMM on GPUs. Our proposal supplies an approach, FastSpMM, which efficiently combines the computational advantages of a level 3 matrix computation and the exploitation of several GPU computing resources: (1) the thread level parallelism [24] and (2) the streaming computation that allows the overlap of CPU–GPU communication/computation. Section 3 shows that FastSpMM clearly outperforms the corresponding CUSPARSE routine in terms of performance on GPUs.

The remainder of the paper is structured as follows. Section 2 starts with a review of the ELLPACK-R format to compress sparse matrices on GPUs and follows up with a detailed analysis of the FastSpMM implementation. In Section 3, a comparative performance evaluation of FastSpMM and other routines for computing SpMM are carried out. Finally, Section 4 summarizes the main conclusions and future works.

2. SPMM PRODUCT ON GPUS

Let us assume that in the matrix matrix product ($A = B \cdot C$), A is sparse with N rows and M columns, while B and C are dense matrices with sizes $M \times L$ and $N \times L$, respectively. Special attention should be paid to the storage of sparse matrix A. To optimize the computation of operations that involve sparse matrices, several storage formats have been devised for specific architectures. These formats define the locality or the coalescence of memory access of the sparse matrix, which are essential to optimize the performance on CPU or GPU architectures.

Compressed row storage (CRS) is the most extended format for storing sparse matrices on superscalar processors [5]. Several GPU routines to compute SpMV based on the CRS format have been proposed [13, 15, 25]. ELLPACK was introduced to exploit vector architecture with sparse operations [20], and recently, variations of ELLPACK have been the key to developing sparse routines capable of achieving high performance on GPUs [9, 16, 17, 26].

Our interest is focused on the format ELLPACK-R [20]. It allows us to store the sparse matrix in a regular manner. It consists of two arrays, $E[\ ]$ (real or complex), which stores the entries and $J[\ ]$ (integer), which stores their column index. The size of the 2D arrays, $E[\ ]$ and $J[\ ]$, is $N \times Max\_nzr$, where Max_nzr is the maximum number of non-zero elements of the rows. Moreover, an additional 1D integer array called $rl[\ ]$ of size N, stores the number of non-zero elements of every row (see Fig. 1a).

The kernel ELLR-T to compute SpMV on GPU, based on ELLPACK-R, outperforms other approaches in terms of performance [14, 18]. In ELLR-T every set of T threads computes the inner product of one row of the sparse matrix and the vector. The global memory access to the matrix is coalesced and aligned. According to the mapping of the threads involved in the computation related to every row, several configurations of ELLR-T are possible. To optimize the performance of the method, the values of two parameters, threads number (T) and threads block size (BS), have to be tuned for each sparse matrix. These can be automatically determined by a specific configuration model associated to ELLR-T according to the particular combination of the input sparse matrix/GPU architecture [14, 19].

In this work, our efforts are oriented to optimize the performance of SpMM operations on GPUs. For this goal,

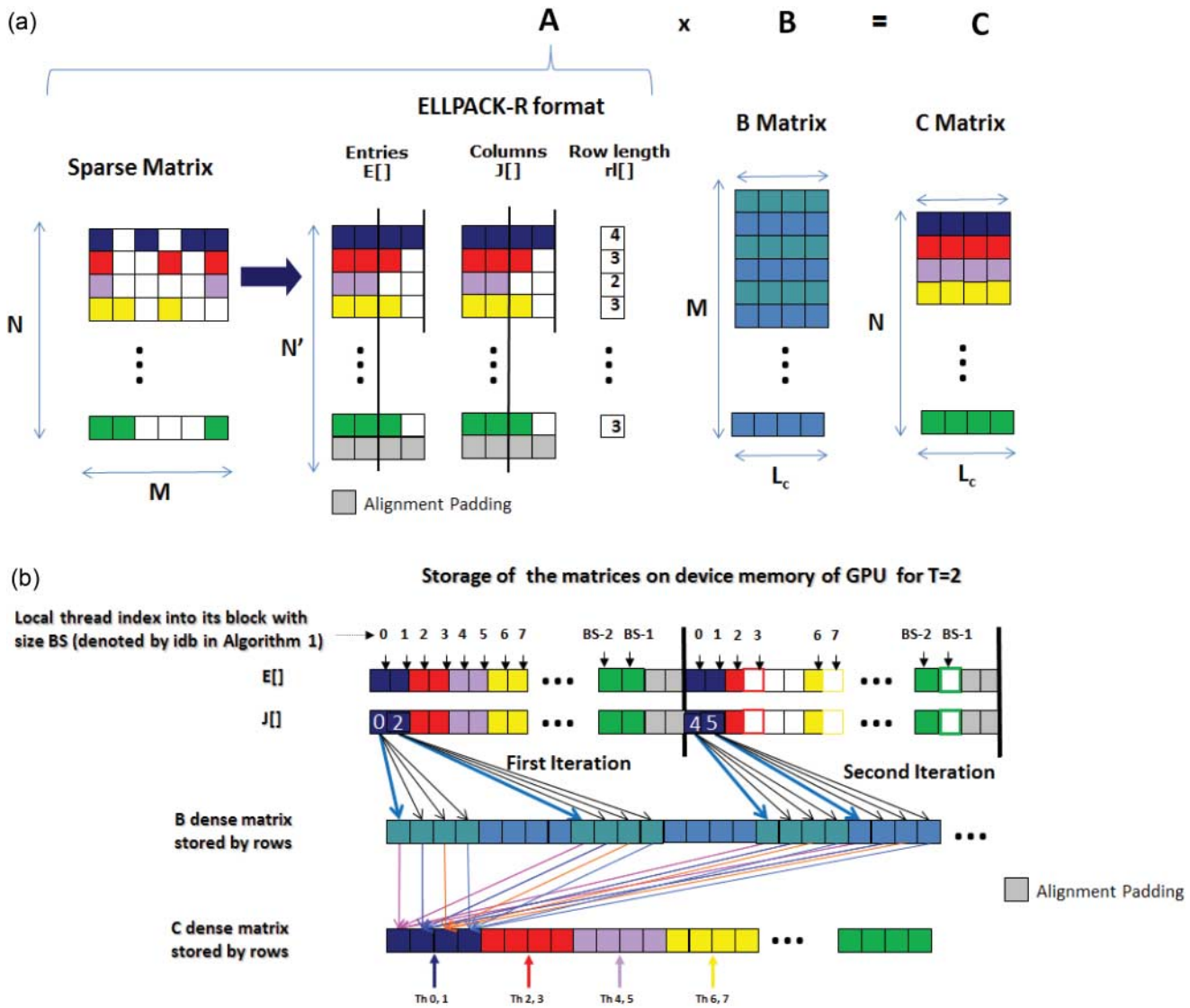

FIGURE 1. Storage format of the sparse and dense matrices and threads mapping of SSFastSpMM (Algorithm 1) for the SpMM operation on GPUs. (**a**) Format of matrices involving an SSFastSpMM. (**b**) Matrices storage scheme for $T = 2$ on device memory and threads mapping of SSFastSpMM on the GPU.

we have created an SpMM routine based on an extension of ELLR-T, called FastSpMM. In our description of the GPU computation of FastSpMM, we have assumed that matrices B and C have $L_{\text{GPU}} \leq L$ columns, where the value of L_{GPU} is chosen such that the memory requirements to store the matrices (A, B and C) are available on the device memory of the GPU architecture. Later on we will explain how to deal with larger matrices. In this way, every set of T threads carries out L_{GPU} reduction operations, those involved in the computation of every row of the output matrix, C. To accelerate them, every thread stores its partial results in the shared memory of the GPU [10]. Bearing in mind, the small size of the shared memory of GPU architectures, every set of T threads can only compute a very limited number of reductions. Therefore, a new parameter is introduced, L_c ($L_c \leq L_{\text{GPU}} \leq L$), as the number of columns of C which are computed by a set of T threads, as described by Algorithm 1 (single step of FastSpMM, SSFastSpMM). FastSpMM will iteratively compute $C_l^{L_c} = AB_l^{L_c}$ ($0 \leq l \leq \lceil L_{\text{GPU}}/L_c \rceil$), where $C_l^{L_c}$ and $B_l^{L_c}$ denote submatrices of C and B with L_c columns, respectively. Note that in Algorithm 1 line 9, the parameter N' ($N' \geq N$) is used to redefine the dimension of E and J such that they fulfill the memory alignment requirements. So, rows of zeroes are padded up to the number of elements of T columns of E (and J) is a multiple of 16 [10]. Consequently, accesses to the matrix A from the device memory satisfy coalescence and alignment

Algorithm 1 Pseudocode of SSFastSpMM to compute SpMM on GPUs. It computes the product $C^{L_c} = AB^{L_c}$ where A is sparse matrix of dimensions $N \times M$, B^{L_c} and C^{L_c} are dense matrices of dimensions $M \times L_c$ and $N \times L_c$, respectively.

```
1: idx=global thread index;
2: idb= local thread index into its block;
3: i = ⌊idx/T⌋;                                          /*Row index of matrix A*/
4: idp = idb mod T; /*Thread index into the set of T threads related to the i-th row of A*/
5: if i < N then
6:    sv_0 = sv_1 = ... = sv_{L_c-1} = 0;
7:    max = ⌈rl[i]/T⌉;
8:    for j = 0 → j < max do
9:       index=T · (j · N'+i)+idp;
10:      v=E[index];
11:      col=J[index];
12:      sv_0+=v · B[col, 0];
13:      sv_1+=v · B[col, 1];
14:      sv_2+=v · B[col, 2];
15:      ...
16:      sv_{L_c-1}+=v · B[col, L_c − 1];
17:   end for
18:   Reduction of sv_0, sv_1, sv_2, ... sv_{L_c-1} on shared memory to compute C[i, 0]C[i, 1]C[i, 2]...C[i, L_c − 1] for the specific
      value of T
19: end if
```

requirements. More details about this approach to optimize the reading of matrix A from the device memory of GPU can be found in [14]. Figure 1 illustrates the organization of threads for SSFastSpMM with $T = 2$ and $L_c = 4$, where a row has been added for memory alignment in the data structure of matrix A.

Algorithm 1 describes the pseudocode for an SSFastSpMM, where several sets of T threads are defined and every set computes L_c elements of the ith output row (i.e. $C[i, 0], \ldots, C[i, L_c - 1]$). So, the ith row of A is split in sets of T elements. Each thread computes $\lceil rl[i]/T \rceil$ iterations of the inner loop of SSFastSpMM (lines 8–17); they compute their partial reductions which are stored in shared memory (lines 12–16). Finally, to generate the values of $C[i, 0], \ldots, C[i, L_c - 1]$, L_c reductions of the T values computed and stored in shared memory by every thread are carried out.

It is remarkable that the data structure to store the sparse matrix and the mapping of threads of SSFastSpMM provides the following advantages: (i) It optimally exploits the memory bandwidth of GPU for the reading of the sparse matrix (E, J and rl) since the storage format allows the coalesced and aligned global memory access and the reduction of load unbalance among threads [14]; (ii) SSFastSpMM is able to take advantage of the high ratio computation/memory access of the SpMM operation compared with the SpMV operation. Note that the SpMM operation ($C = A \cdot B$) could be computed by a set of SpMV operations ($c_i = A \cdot b_i$ with $0 \leq i \leq L_{GPU}$). However, in this case, the sparse matrix is read L_{GPU} times from the device memory to compute C. In contrast, FastSpMM reads the sparse matrix once and computes L_c columns of C, so the ratio computation/memory access for FastSpMM is L_c times higher than for SpMM computed as a set of SpMV. Furthermore, FastSpMM has another computational advantage since the temporal locality for reading B is increased and the indirect addressing to read the elements of B is evaluated just once to compute L_c elements of C; thus, the ILP is better exploited by FastSpMM. Consequently, FastSpMM combines the computational advantages described above, with the advantages of the ELLR-T format to exploit the GPU architecture.

It is relevant to emphasize that high performance values of FastSpMM on GPUs are only obtained for very large sparse matrices [19]. So, the size of the corresponding dense matrices is also very large, with huge memory requirements not provided by the device memory of the GPU. Consequently, the number of columns of B involved in one step of FastSpMM cannot be increased excessively.

Bearing in mind these GPU limitations, FastSpMM has finally been designed as a routine that computes $C^{L_{GPU}} = A \cdot B^{L_{GPU}}$ (the value of L_{GPU} is chosen such that the memory requirements to store the matrices, A, $B^{L_{GPU}}$ and $C^{L_{GPU}}$, are available on the device memory of the GPU architecture). Thus, to compute $C = A \cdot B$ on the GPU, it is necessary to include $\lceil L/L_{GPU} \rceil$ successive communications between CPU–GPU with sets of L_{GPU} columns of B and C. It is noteworthy that the above-mentioned limitations to compute SpMM on GPUs are associated to all the implementations of the SpMM on GPUs. Consequently, for the CUSPARSE routine it is also necessary

Algorithm 2 FastSpMM*, general scheme to compute $C = A \cdot B$ based on SSFastSpMM.

1: Copy the sparse matrix A to the GPU memory
2: **for** $i = 0 \rightarrow \lceil L/L_{GPU} \rceil - 1$ **do**
3: Copy L_{GPU} columns of B from CPU to GPU memory
4: **for** $j = 0 \rightarrow \lceil L_{GPU}/L_c \rceil - 1$ **do**
5: SSFastSpMM ($C_j^{L_c} = AB_j^{L_c}$ on GPU)
6: **end for**
7: Copy L_{GPU} columns of C from GPU to CPU memory
8: **end for**
9: output C matrix is stored on CPU memory

Algorithm 3 FastSpMM: code with streaming computation CPU–GPU to compute the SpMM operation.

1: Streams creations 0 and 1
2: **Stream 1:** Asynchronous copy of B_0 (the column chunk of B) to the GPU memory
3: **for** $i = 0 \rightarrow \lceil L/L_{GPU} \rceil - 1$ **do**
4: $p = i \mod 2; q = (p + 1) \mod 2$
5: **Stream p:** $\lceil L_{GPU}/L_c \rceil$ executions of SSFastSpMM to compute $C_i = A \cdot B_i$
6: **Stream q:** Asynchronous copy of B_i to the GPU memory
7: **Stream p:** Asynchronous copy of C_i to the CPU memory
8: **end for**
9: Streams delete

to include the same communications between the GPU and the CPU. This library is taken as a reference to evaluate our proposal. A basic algorithm called FastSpMM* is considered as the starting point to compute $C = A \cdot B$ with a basic CPU–GPU communication scheme. The CPU–GPU communication scheme to compute $C = A \cdot B$ on the GPU is described in Algorithm 2 (FastSpMM*), where the parameter L_{GPU} is an upper bound of L_{c} for FastSpMM* and a multiple of L_{c}. This scheme is also valid for the approach based on the SpMV operation ($L_{\text{c}} = 1$).

From the description of Algorithm 2, it is clear that the cost of the CPU–GPU communications is very high because every iteration requires the exchange of L_{GPU} columns of B and C between the CPU and the GPU memory. Therefore, the streaming computation for overlapping CPU–GPU communication and computation is also considered in the optimized version of the SpMM operation as described in the next subsection.

2.1. Streaming computation for FastSpMM

CPU–GPU communications play an important role in FastSpMM* (Algorithm 2) and have a strong impact on the performance of this routine. However, the scheme of FastSpMM* includes iterative steps of communications and computations which could be overlapped at runtime. The idea of the overlapping communication/computation consists in doing the computational work while the communication infrastructure simultaneously performs data transfers, with the goal of hiding the latency and transfer costs of the inter-process communication [27].

In this work, the strategy for overlapping communication/computation between the host and the GPU platform is based on the stream processing supplied by CUDA [27, 28]. In this context, a stream represents a queue of GPU operations that are executed in a specific order. Operations in different streams can be interleaved and in some cases overlapped, a property that can be used to hide data transfers between the host and the device. Therefore, the scheme described in Algorithm 2 can be expressed by two streams which interleave their execution as shown in Algorithm 3.

Our approach relies on two points: the 'chunked' computation (for overcoming the GPU memory limitation and executing SpMM with high dimensions) and overlapping the memory read/write operations (memcpy) with the kernel executions (for decreasing the communication penalization CPU–GPU and vice versa). Assuming that our memory read/write operations and kernel executions take roughly the same amount of time, Fig. 2 shows the execution timeline of FastSpMM, where: stream 0 sends its input buffers to the GPU; later, stream 1 executes the same operation while stream 0 is executing its kernel; then, stream 1 will execute its kernel while stream 0 copies its results to the host; and, finally, a new input buffer is copied to the GPU and the obtained results are sent to the host memory. This process will be repeated for the next chunks of data.

From the interleaved execution of both streams (0 and 1) in Fig. 2, we can identify an iterative sequence of four time steps with six operations (enclosed in brackets). These six operations would consume only four time steps instead of six. Therefore,

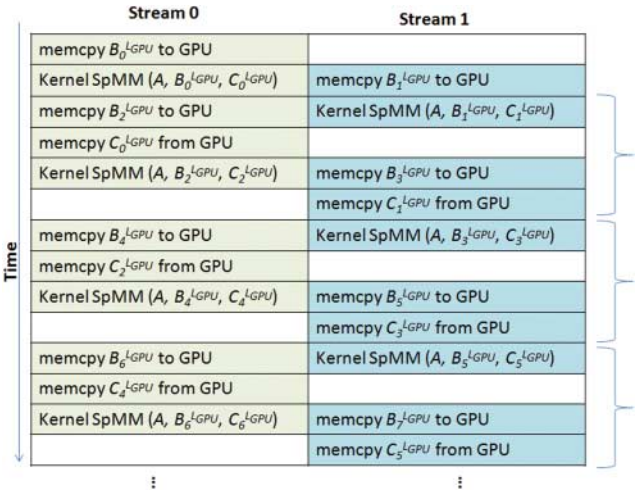

FIGURE 2. Timeline of FastSpMM execution using two independent streams (Stream 0 and 1).

if we define the acceleration factor (AF) as the ratio between sequential and overlapped runtimes, then AF can achieve the maximum value of 1.5.

It is necessary to underline that the previous analysis has been simplified, since it assumes that the SpMM kernel (A, B_i, C_i) in Fig. 2 spends the same time as every communication GPU–CPU and CPU–GPU (memcpy in the same figure), i.e. the computation steps waste the same time than the communication steps. However, on the current GPU platforms the bandwidth is higher for the CPU–GPU communication than in the GPU–CPU direction. Moreover, the time of the SpMM kernel (A, B_i, C_i) increases as $nz \cdot L_{\text{GPU}}$ increases and the communication time (CT) increases as N and/or M increase. Therefore, the impact of the streaming included in FastSpMM in terms of performance depends on the characteristics of the matrices involved in the SpMM.

3. EVALUATION

This section is devoted to evaluating several approaches for computing the SpMM operation using a wide set of test sparse matrices. These matrices come from a broad spectrum of disciplines of science and engineering and exhibit different characteristics, from those that are well structured and regular to highly irregular matrices with large unbalances in the distribution of non-zeros per matrix row. Table 1 shows the set of test matrices and the characteristic parameters related to their specific patterns: number of rows (N), total number of non-zero elements (nz), average (Av) and maximum (Max_nzr) number of entries per row and the estimated giga floating point operations for the SpMM computation ($NFLOP = 2N \cdot nz/2^{30}$). In this table, some special matrices are considered: three dense matrices (dense2, dense5000 and dense10000) and two tridiagonal matrices (tridiagonal1 and tridiagonal2). It is relevant to underline that these special matrices do not represent the prototype of unstructured sparse matrices, and so FastSpMM has not been specifically designed to accelerate the SpMM for these types of matrices. However, they are included in our evaluation because they help to analyze computational characteristics of our library for matrices with extreme values of Av. Note that all matrices are ordered according to the value of the Av parameter and all of them verify $N = M$.

The evaluation has been carried out on two GPU platforms with the Fermi architecture of NVIDIA [29]: a Tesla C2050 and a GeForce GTX480. The main characteristics of both GPUs are shown in Table 2.

TABLE 1. Characteristics of test sparse matrices (denoted as A in our definition of SpMM), where N is the number of rows, nz is the total number of nonzero elements, Av and Max_nzr are the average and the maximum number of entries per row, respectively, and $NFLOP = 2N \cdot nz/2^{30}$ is the estimated giga floating point operations.

Matrix	$N[\times 10^3]$	$nz[\times 10^6]$	Av	Max_nzr	$NFLOP$
tridiagonal1	200	0.6	3	3	223.5
tridiagonal2	500	1.5	3	3	1396.9
cop20k_A	121	2.6	22	81	592.4
qcd5_4	49	1.9	39	40	175.5
Si87H76	240	10.6	45	361	4773.4
msdoor	416	19.1	47	77	14 851.6
pwtk	218	11.6	53	181	4722.5
shipsec1	141	7.8	55	102	2050.2
cant	62	4.0	64	78	466.2
Ga41As41H72	268	18.4	69	702	9232.5
consph	83	6.0	72	81	933.0
x104	108	8.7	81	324	1759.1
RM07R	382	37.4	99	295	26 635.8
pdb1HYS	36	4.3	119	204	294.7
wbp128	16	3.9	240	256	120.0
nd3k	9	3.2	365	515	55.0
wbp256	66	31.4	480	512	3834.7
dense2	2	4.0	2000	2000	14.9
dense5000	5	25.0	5000	5000	232.8
dense10000	10	100.0	10 000	10 000	1862.6

TABLE 2. Characteristics of the GPU platforms considered for the evaluation.

	Tesla C2050	GTX480
Peak GFlops (single precision)	1030	1350
Peak GFlops (double precision)	515	168
Bandwidth (GB/s)	144	177.4
Clock (GHz)	1.2	1.4
Device memory (GB)	2.6	1.5
Cores	448	480

Four routines to compute the SpMM on GPUs have been evaluated:

(1) **SpMM CUSPARSE**, based on the level 3 function of this library, computes $C = \alpha \cdot A \cdot B + \beta \cdot C$ with $\alpha = 1$ and $\beta = 0$. This routine prevents the useless computation of βC when $\beta = 0$.
(2) **SetSpMVs** (ELLR-T) computes the SpMM as a set of N SpMV operations based on the kernel ELLR-T [14].
(3) **FastSpMM*** is based on the ELLR-T format without overlapping communication/computation (Algorithm 2).
(4) **FastSpMM** is based on the ELLR-T format and takes advantage of overlapping communication/computation (Algorithm 3).

The CUSPARSE library provides a set of basic linear algebra subroutines used for handling sparse matrices based on the CRS format to compress the sparse matrix [5, 13]. The paradigm of CUSPARSE is to define multiple blocks where each block is in charge of processing a group of rows. Each row is assigned to a set of threads. Moreover, a few additional approaches for improving performance are considered: (1) adjust the number of threads per row to minimize the unbalance among threads; (2) align threads per row for coalescing and (3) use shared and texture memory [13]. However, CUSPARSE does not allow the user to select values of configuration parameters such as threads BS or the number of threads (T) to achieve the optimal performance.

A comparative evaluation of the performance achieved by the four versions of the SpMM operation has been carried out. To fairly compare the performance values of both FastSpMM versions with the CUSPARSE routine, all of them compute the operation $C = \alpha \cdot A \cdot B$ with $\alpha = 1$. It is relevant to note that SpMM CUSPARSE, SetSpMVs and FastSpMM* versions include the same communications scheme (as described in Algorithm 2). In our evaluation, the size of B and C is $N \times N$ ($L = M = N$).

Let us remark that the values of L_c (number of columns of C computed in the execution of SSFastSpMM) and L_{GPU} (number of columns of B and C involved in a single CPU–GPU transfer) are strongly related to the performance of the SpMM for all the approaches. So, $L_c = 1$ for SetSpMVs and L_c can be considered as a variable parameter for FastSpMM* and FastSpMM, with $L_c \leq L_{GPU}$. Experiments to explore the best results in terms of performance according to the values of L_c have been carried out and we have concluded that the highest performance is achieved when $L_c = L_{GPU}$.

A preliminary evaluation of the performance of FastSpMM for several values of L^* ($L^* = L_{GPU} = L_c$) has been carried out. The result of this evaluation will help us to determine the value of L^* (for each platform, Tesla C2050 and GTX480), which will be later used in our comparative evaluation of the performance of FastSpMM and FastSpMM* with respect to CUSPARSE and SetSpMVs. Figure 3 shows the experimental results of our evaluation (performance of FastSpMM as a function of L^*). The values of the performance have been optimized with respect to the configuration parameters BS and T, taking also into account the memory limitations of both platforms. It is remarkable that, for the *Ga41As41H72* matrix and the GTX480 platform, only the configurations with $L^* \leq 4$ have been possible because of the high device memory requirements of this matrix. The complete set of test matrices has been evaluated; however, for the sake of clarity dense and tridiagonal matrices have not been included in Fig. 3. For tridiagonal matrices the performance of FastSpMM is very low and decreases as L^* increases. For dense matrices, the performance is very high and the larger the value of L^*, the better is the performance. From this study, we can conclude that, for most of the test matrices, appropriate values of L^* for the Tesla C2050 and GTX480 platforms are 8 and 4, respectively. For larger values of this parameter, the performance does not improve significantly. Note that the less the value of L^*, the less memory resources are required, facilitating the portability of FastSpMM to GPU platforms with limited resources or the allocation of additional data structures required when FastSpMM is assembled with other kernels on the GPU. Bearing in mind all the previous considerations, in the rest of our experiments the value of the parameter L^* has been set to 4.

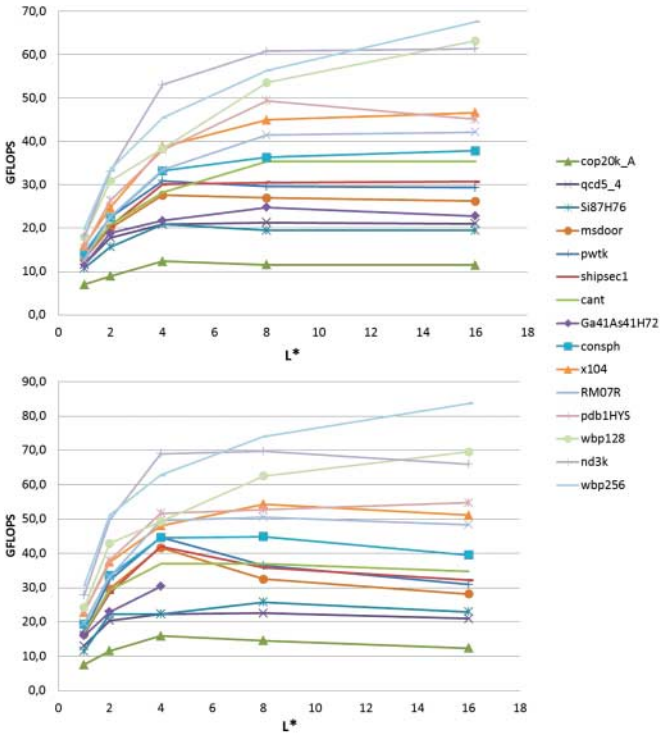

FIGURE 3. Performance of FastSpMM as a function of L^* ($L^* = L_c = L_{GPU}$) for test sparse matrices on Tesla C2050 (top) and GTX480 (bottom).

An additional evaluation of the impact of the configurable cache on our SpMM approach has been carried out. The purpose of this evaluation is to optimize our SpMM kernel in terms of performance. Note that, for CUDA devices of compute capability over 2.0, like the two platforms used in this paper (Fermi architecture), there is 64 kB of memory for each multiprocessor. This per-multiprocessor on-chip memory is split and used for both shared memory and L1 cache [29]. Thus, we have tuned the scratchpad-cache for making better use of on-chip memory. The tested configurations have been: (1) 48 kB of shared memory with 16 kB of L1 cache (default configuration); and (2) 16 kB of shared memory with 48 kB of L1 cache.

Note that both configurations applied to FastSpMM can achieve different processing times (PTs) but the GPU–CPU CTs do not depend on the cache configuration. So, the analysis of the impact of both configurations in the performance of FastSpMM has been done in terms of the values of the PT. Table 3 shows the PTs of FastSpMM for the set of test matrices and both configurations. As can be observed in this table, the configuration using a higher capacity of the L1 obtains the best results. Bearing in mind that FastSpMM requires a maximum of 16 kB for shared memory for all test matrices (for $L_c = 4$ and float data type), both configurations provide these shared memory requirements. FastSpMM is dominated by device memory accesses to read and write the matrices involved in the operation $A \cdot B = C$. The access pattern to read the matrix B is related to the location of the entries of the sparse matrix A; therefore, the management of the device memory can be improved with a larger cache memory [30]. According to these considerations, results in Table 3 show that the configuration with a larger L1 achieves better performance. This is the configuration used in the remaining evaluations.

We have also carried out a comparative analysis of FastSpMM* and FastSpMM to assess the improvement of the performance due to the overlapping communication/computation in the SpMM. Table 4 shows the values for the CT, PT, total Runtime (Runt), percentage of the CT in the Runt (% Com) and AF of the FastSpMM versus FastSpMM* (AF) for every test matrix on both GPUs. It is necessary to highlight that the values of runtime related to the FastSpMM* algorithm include the computation and CT. Communications represent a relevant percentage of runtime, according to the scheme of Algorithm 2.

Experimental results from Table 4 show that the values of CT are very relevant in this kind of operation and depend on the matrix dimension (N). From Table 4, we can conclude that FastSpMM outperforms FastSpMM* in terms of runtime for all test matrices due to the exploitation of overlapping communication/computation. This advantage is less relevant if the CT with respect to the Runt (see % Com column of Table 4) achieves extreme values. So, for tridiagonal matrices (with %Com > 84) and for dense matrices (with %Com < 13) the AF is nearly 1. Therefore, these experimental results confirm that the approach to overlap communication/computation considerably improves the performance of SpMM on GPU.

TABLE 3. PT (seconds) of the SpMM operation using two configurations of the shared and L1 memory over the FastSpMM approach.

	Tesla C2050			GTX480		
Matrix	PT_s	PT_{L1}	AF_{L1}	PT_s	PT_{L1}	AF_{L1}
tridiagonal1	19.3	19.3	1.0	14.0	13.7	1.0
tridiagonal2	76.9	76.2	1.0	46.3	45.4	1.0
cop20k_A	37.1	29.0	1.3	28.3	20.2	1.4
qcd5_4	5.6	5.0	1.1	4.2	3.7	1.1
Si87H76	173.4	157.8	1.1	123.5	113.2	1.1
msdoor	361.4	325.1	1.1	251.2	210.0	1.2
pwtk	87.4	81.9	1.1	69.0	56.5	1.2
shipsec1	42.0	40.9	1.0	32.8	28.5	1.2
cant	12.2	10.9	1.1	9.3	8.3	1.1
Ga41As41H72	336.4	320.0	1.1	243.4	240.8	1.0
consph	20.9	19.0	1.1	16.2	13.6	1.2
x104	40.8	39.9	1.0	36.0	27.0	1.3
RM07R	678.5	623.2	1.1	557.3	413.1	1.3
pdb1HYS	6.8	6.0	1.1	5.1	4.5	1.1
wbp128	3.3	3.1	1.1	2.5	2.2	1.1
nd3k	1.3	0.9	1.4	1.0	0.7	1.4
wbp256	108.0	82.0	1.3	80.2	65.5	1.2
dense2	0.3	0.2	1.3	0.2	0.1	1.3
dense5000	3.7	3.0	1.2	2.6	2.0	1.3
dense10000	24.2	22.9	1.1	16.6	15.2	1.1

Columns PT_s and PT_{L1} refer to the configuration with 48 kB of shared memory and with 48 kB of L1 cache, respectively. Column AF_{L1} identifies the AF ($AF_{L1} = TP_s/TP_{L1}$). Special matrices (dense and tridiagonal matrices) are in italics.

A comparative analysis of the performance achieved by the CUSPARSE, SetSpMVs (ELLR-T), FastSpMM* and FastSpMM versions of SpMM has been carried out. Experimental results for all the sparse matrices and both the Tesla C2050 and GTX480 platforms have been depicted in Fig. 4. These results clearly show that the performance strongly depends on the following:

(1) *The characteristics of the sparse matrices*. For instance, *wbp256*, *nd3k* and *wbp128* matrices achieve the best performance for all approaches due to their high average number of elements per row (see Av in Table 1). However, the product of the *Ga41As41H72* matrix achieves a poor performance due to the irregular filling of its rows, which produces a high value of Max_nzr (specific parameter of the ELLPACK-R format described in Section 2). So, the memory requirements to store the matrix A are enlarged with ELLPACK-R and the memory management is also

TABLE 4. Profiling of SpMM based on FastSpMM* and FastSpMM on Tesla C2050 and GTX 480.

					FastSpMM*			FastSpMM	
GPU	Matrix	Av	Max_nzr	PT	CT	% Com	Runt	Runt	AF
Tesla C2050	*tridiagonal1*	3	3	19.3	108.8	84.9	128.1	109.9	1.17
	tridiagonal2	3	3	76.2	619.8	89.1	695.9	623.5	1.12
	cop20k_A	22	81	29.0	41.9	59.1	70.9	51.5	1.38
	qcd5_4	39	40	5.0	8.1	62.0	13.0	9.0	1.44
	Si87H76	45	361	157.8	158.6	50.1	316.4	244.9	1.29
	msdoor	47	77	325.1	431.5	57.0	756.6	610.5	1.24
	pwtk	53	181	81.9	125.2	60.4	207.1	164.2	1.26
	shipsec1	55	102	40.9	56.8	58.1	97.7	73.2	1.33
	cant	64	78	10.9	13.1	54.5	23.9	17.7	1.35
	Ga41As41H72	69	702	320.0	188.9	37.1	508.9	457.1	1.11
	consph	72	81	19.0	21.2	52.8	40.2	30.2	1.33
	x104	81	324	39.9	32.5	44.9	72.3	56.8	1.27
	RM07R	99	295	623.2	379.1	37.8	1002.3	855.1	1.17
	pdb1HYS	119	204	6.0	4.6	43.4	10.5	8.3	1.27
	wbp128	240	256	3.1	1.0	24.0	4.1	3.4	1.20
	nd3k	365	515	0.9	0.3	26.5	1.3	1.1	1.15
	wbp256	480	512	82.0	14.5	15.0	96.5	90.5	1.07
	dense2	2000	2000	0.2	0.0	10.3	0.2	0.2	1.06
	dense5000	5000	5000	3.0	0.1	3.5	3.2	3.0	1.05
	dense10000	10 000	10 000	22.9	0.4	1.9	23.3	23.0	1.01
GTX480	*tridiagonal1*	3	3	13.7	89.4	86.7	103.0	99.2	1.04
	tridiagonal2	3	3	45.4	519.6	92.0	565.0	533.5	1.06
	cop20k_A	22	81	20.2	36.0	64.0	56.2	40.0	1.40
	qcd5_4	39	40	3.7	7.0	65.3	10.7	8.4	1.27
	Si87H76	45	361	113.2	145.1	56.2	258.3	210.7	1.23
	msdoor	47	77	210.0	356.6	62.9	566.5	404.2	1.40
	pwtk	53	181	56.5	103.2	64.6	159.6	113.5	1.41
	shipsec1	55	102	28.5	47.1	62.3	75.6	52.5	1.44
	cant	64	78	8.3	11.1	57.1	19.4	13.6	1.43
	Ga41As41H72	69	702	240.8	152.4	38.8	393.2	326.2	1.21
	consph	72	81	13.6	18.9	58.2	32.5	22.5	1.45
	x104	81	324	27.0	28.7	51.5	55.7	45.9	1.21
	RM07R	99	295	413.1	357.5	46.4	770.6	577.2	1.34
	pdb1HYS	119	204	4.5	3.9	46.5	8.4	6.1	1.37
	wbp128	240	256	2.2	0.8	26.8	3.0	2.6	1.16
	nd3k	365	515	0.7	0.3	27.8	1.0	0.9	1.15
	wbp256	480	512	65.5	12.5	16.0	78.0	65.6	1.19
	dense2	2000	2000	0.1	0.0	12.2	0.2	0.2	1.05
	dense5000	5000	5000	2.0	0.1	4.5	2.0	2.0	1.04
	dense10000	10 000	10 000	15.2	0.4	2.5	15.6	15.2	1.03

The following notation is used, Av and Max_nzr, average and the maximum number of entries per row, respectively; PT, processing time; CT, communication time; Runt, total runtime (all in seconds); % Com, percentage of CT with respect to the Runt for the FastSpMM* approach; AF, acceleration factor of the FastSpMM with respect to FastSpMM*. Special matrices (dense and tridiagonal matrices) are in italics.

penalized. The regular pattern of the *pwtk* matrix allows one to achieve a better performance despite its low value of Av.

(2) *The approach for computing the SpMM on GPU*. We can observe that, for most of the test matrices, the CUSPARSE library achieves the poorest performance

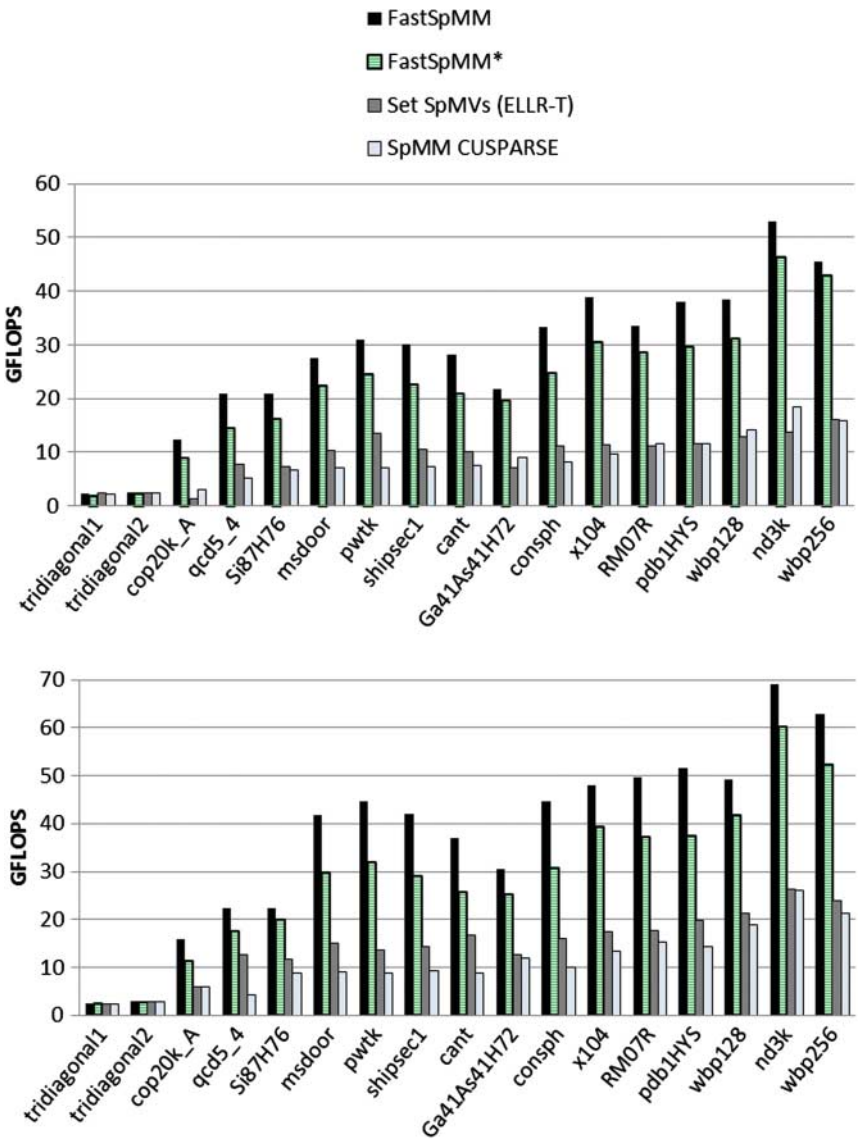

FIGURE 4. Performance evaluation of the sparse matrices using the approaches: CUSPARSE, SetSpMVs, FastSpMM* and FastSpMM to compute SpMM on Tesla C2050 (top) and GTX480 (bottom).

on both GPUs. This is mainly due to the fact that this library does not allow the user to set up the parameters according to the pattern of the matrix. The SetSpMVs version achieves slightly better performance since suitable parameters to obtain the best performance of SetSpMVs have been selected according to the GPU platform and the characteristics of the sparse matrices. However, its performance is very poor compared with FastSpMM* and FastSpMM. Focusing our attention on the performance improvements of FastSpMM* versus CUSPARSE and SetSpMVs, we can remark that they are related to the suitable exploitation of the high ratio computation/memory access in combination with the advantages of the ELLPACK-R format to exploit the GPU architecture. Finally, we can conclude that FastSpMM achieves the best performance owing to the use of the schema for overlapping communication/computation based on CUDA streaming and the advantages of FastSpMM*.

To estimate the net gain provided by GPUs over modern processors in the SpMM, we have chosen the optimized versions of SpMM routines for both kinds of architectures. In our experiments, a computer based on a state-of-the-art multicore processor, Intel Xeon E5640 with 8 cores, has been used to compute the SpMM operation based on the MKL library [23]. It is remarkable that, for very large matrices, the memory requirements of SpMM are not supplied by the CPU

TABLE 5. Runtime executions in seconds of the SpMM multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C) and AFs of FastSpMM on Tesla and GTX480 (*AF Tesla* and *AF GTX*480, respectively) over the MKL version with 8 cores.

	1C	2C	4C	8C	*AF Tesla*	*AF GTX*480
cop20k_A	721.8	444.0	211.6	112.0	2.2	2.8
qcd5_4	148.8	72.2	38.0	19.6	2.2	2.3
Si87H76	4112.9	2191.3	1219.8	639.6	2.6	3.0
msdoor	14 049.4	7191.6	3936.9	2282.7	3.7	5.6
pwtk	3781.4	1975.1	1043.7	566.3	3.4	5.0
shipsec1	1651.1	865.5	469.3	251.0	3.4	4.8
cant	381.0	244.6	103.8	55.8	3.1	4.1
Ga41As41H72	7629.0	4022.5	2298.3	1211.7	2.7	3.7
consph	734.6	385.1	218.6	109.6	3.6	4.9
x104	1563.6	866.8	447.1	258.3	4.5	5.6
RM07R	21 263.3	11 166.8	5900.3	3390.4	4.0	5.9
pdb1HYS	242.6	121.2	68.4	36.8	4.4	6.0
wbp128	106.4	52.4	27.6	14.4	4.3	5.5
nd3k	42.9	22.0	12.8	7.3	6.6	8.5
wbp256	4145.6	2208.1	1033.9	518.5	5.7	7.9

architecture. In these cases, the 'chunked' computation has been also applied for overcoming the CPU memory limitation. Table 5 shows the experimental results in terms of runtime (in seconds) of the SpMM multicore version using MKL with 1, 2, 4 and 8 cores (1C, 2C, 4C and 8C) and the AFs of FastSpMM on Tesla and GTX480 (*AF Tesla* and *AF GTX*480, respectively) with respect to the MKL version with 8 cores. As can be observed, the increase of the number of cores means a considerable reduction in the Runt of the MKL version. The results obtained using MKL and 8 cores have been compared with the runtimes of the FastSpMM algorithm, showing the AFs for every matrix and GPU considered. These AFs range from 2.2× (2.3×) to 6.6× (8.5×) on Tesla C2050 (GTX480) for the set of test matrices. So, we can conclude that the GPU turns out to be an excellent accelerator of SpMM.

4. CONCLUSION AND FUTURE WORKS

In this paper, the FastSpMM approach to accelerate the SpMM operation on GPUs is analyzed. FastSpMM combines two considerations: (1) a high ratio computation/memory access with the advantages of the ELLPACK-R format to exploit the GPU architecture and (2) streaming computation for overlapping communication/computation. The comparative evaluation with other proposals (CUSPARSE, SpMM as a set of SpMV operations based on ELLR-T and FastSpMM*) has proved that FastSpMM outperforms them in terms of performance. A comparison of FastSpMM on two GPUs (Tesla C2050 and GeForce GTX480) has revealed that AFs of up to 6.6× and 8.5× can be achieved in comparison with an optimized implementation of SpMM, which exploits 8 cores of a state-of-the-art multicore processor. However, for very large and extremely sparse matrices, the SpMM achieves poor performance on the GPU, mainly due to the relevance of the CPU–GPU communications of the dense matrices (B and C).

According to the previous results, several aspects of the FastSpMM can be improved and so our future work will be focused on them. Concretely, our next work will consist of extending FastSpMM to matrices with complex elements instead of real ones and broadening the kinds of platforms that can be exploited by FastSpMM. Moreover, we are particularly interested in the reduction of the PT through improving the memory management. To achieve this goal, FastSpMM will be rewritten according to the GPU programming tool CudaDMA [31], which allows one to efficiently manage data transfers between the on-chip and off-chip memories of GPU platforms. Finally, due to the importance of the L^* parameter in the performance of the SpMM product, it will be interesting to develop a model to optimize the value of this parameter according to the specific matrix characteristics and the GPU platform.

FastSpMM for computing SpMM on GPU platforms is freely available through the following web site: https://sites.google.com/site/mcfastsparse/.

FUNDING

G.O. is a fellow of the Spanish FPU programme. This work supported by the Spanish Ministry of Science (TIN2008-01117, TIN2012-37483-C03-03, TIN2012-37483-C03-01) and J. Andalucía (P10-TIC-6002), partially financed by the European Reg. Dev. Fund (ERDF).

REFERENCES

[1] Geveler, M., Ribbrock, D., Göddeke, D., Zajac, P. and Turek, S. (2011) Towards a complete FEM-based simulation toolkit on GPUs: unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. *Comput. Fluids*, doi: 10.1016/j.compfluid.2012.01.025.

[2] Vázquez, F., Garzón, E.M. and Fernández, J.J. (2010) A matrix approach to tomographic reconstruction and its implementation on GPUs. *J. Struct. Biol.*, **170**, 146–151.

[3] Vázquez, F., Garzón, E.M. and Fernández, J.J. (2011) Matrix implementation of simultaneous iterative reconstruction technique (SIRT) on GPUs. *Comput. J.*, **54**, 1861–1868.

[4] Ortega, G., Lobera, J., Arroyo, M., García, I. and Garzón, E.M. (2012) High Performance Computing for Optical Diffraction Tomography. *Proc. 2012 Int. Conf. High Performance Computing & Simulation (HPCS 2012)*, Madrid, Spain, July 2–6, pp. 195–201. IEEE, New York, USA.

[5] Bisseling, R.H. (2004) *Parallel Scientific Computation*. Oxford University Press, Oxford.

[6] Quateroni, A., Sacco, R. and Saleri, F. (2007) *Numerical Mathematics*. Springer, New York.

[7] Ortega, G., Garzón, E.M., Vázquez, F. and García, I. (2013) The BiConjugate gradient method on GPUs. *J. Supercomput.*, **64**, 49–58.

[8] Tabik, S., Romero, L.F., Garzón, E.M. and Ramos, J.I. (2008) On a model of three-dimensional bursting and its parallel implementation. *Comput. Phys. Commun.*, **178**, 471–485.

[9] Vázquez, F., Fernández, J.J. and Garzón, E.M. (2011) A new approach for sparse matrix vector product on NVIDIA GPUs. *Concurrency Comput. Pract. Exp.*, **23**, 815–826.

[10] DG-05603-001_v4.0 (2011) *CUDA C Best Practices Guide 4.0 edition*. NVIDIA Corporation, USA.

[11] Version:1.2 (2010) *The OpenCL Specification*. Khronos OpenCL Working Group, Beaverton, OR, USA.

[12] PG-05326-032_V02 (2010) *CuBLAS Library*. NVIDIA Corporation, Santa Clara, USA.

[13] PG-05329-032_V02 (2010) *CUDA CUSPARSE Library*. NVIDIA Corporation, Santa Clara, USA.

[14] Vázquez, F., Fernández, J.J. and Garzón, E.M. (2012) Automatic tuning of the sparse matrix vector product on GPUs based on the ELLR-T approach. *Parallel Comput.*, **38**, 408–420.

[15] Bell, N. and Garland, M. (2009) Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. *Proc. Conf. High Performance Computing Networking, Storage and Analysis (SC '09)*, Portland, Oregon, November 14–20, pp. 18:1–18:11. ACM, New York, NY, USA.

[16] Choi, J.W., Singh, A. and Vuduc, R. (2010) Model-driven autotuning of sparse matrix-vector multiply on GPUs. *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP '10)*, Bangalore, India, January 9–14, pp. 115–126. ACM, New York, NY, USA.

[17] Monakov, A., Lokhmotov, A. and Avetisyan, A. (2010) Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. *Proc. HiPEAC 2010*, Pisa, Italy, Lecture Notes in Computer Science 5952, January 25–27, pp. 111–125. Springer, Berlin.

[18] Dziekonski, A., Lamecki, A. and Mrozowski, M. (2011) A memory efficient and fast sparse matrix vector product on a GPU. *Prog. Electromag. Res.*, **116**, 49–63.

[19] Vázquez, F., Ortega, G., Fernández, J.J. and Garzón, E.M. (2010) Improving the Performance of the Sparse Matrix Vector Product with GPUs. *Proc. IEEE Int. Conf. Computer and Information Technology (CIT 2010)*, Bradford, UK, June 29–July 1, pp. 1146–1151. IEEE, New York, USA.

[20] CNA-232 (1989) *ITPACKV 2D User's Guide*. Center for Numerical Analysis, University of Texas at Austin.

[21] Anderson, E. *et al.* (1999) *LAPACK Users' Guide* (3rd edn). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[22] Golub, G.H. and van Loan, C.F. (1996) *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)* (3rd edn). The Johns Hopkins University Press, Baltimore, USA.

[23] 630813-054US (2009) *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA.

[24] Vázquez, F., Ortega, G., Fernández, J.J., Garcia, I. and Garzón, E.M. (2012) Fast Sparse Matrix Matrix Product based on ELLR-T and GPU Computing. *Proc. IEEE 10th Int. Symp. Parallel and Distributed Processing with Applications (ISPA '12)*, Madrid, Spain, July 10–13, pp. 669–674. IEEE Computer Society, Washington, DC, USA.

[25] RC24704 (W0812–047) (2009) *Optimizing Sparse Matrix-Vector Multiplication on GPUs*. IBM T. J. Watson Research Center, New York, USA.

[26] Su, B.-Y. and Keutzer, K. (2012) clSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs. *Proc. 26th ACM Int. Conf. Supercomputing (ICS '12)*, San Servolo Island, Venice, Italy, June 25–29, pp. 353–364. ACM, New York, NY, USA.

[27] White, J.B. and Dongarra, J.J. (2011) Overlapping Computation and Communication for Advection on Hybrid Parallel Computers. *IPDPS*, Alaska, USA, May 16–20, pp. 59–67. IEEE, New York, USA.

[28] Sanders, J. and Kandrot, E. (2010) *CUDA by Example: An Introduction to General-Purpose GPU Programming* (1st edn). Addison-Wesley, Massachusetts, USA.

[29] Whitepaper V1.1 (2010) *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. NVIDIA Corporation, Santa Clara, USA.

[30] Torres, Y., Gonzalez-Escribano, A. and Llanos, D.R. (2011) Understanding the Impact of CUDA Tuning Techniques for Fermi. *Proc. 2011 Int. Conf. High Performance Computing & Simulation (HPCS 2011)*, Istanbul, Turkey, July 4–8, pp. 631–639. IEEE, New York, USA.

[31] Bauer, M., Cook, H. and Khailany, B. (2011) CudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC '11)*, Seatle, WA, November 12–18. IEEE, New York, USA.