

Selecting optimal SpMV realizations for GPUs via machine learning

Ernesto Dufrechou¹ , Pablo Ezzatti¹
and Enrique S Quintana-Ortí²

The International Journal of High Performance Computing Applications 2021, Vol. 35(3) 254–267
© The Author(s) 2021
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342021990738
journals.sagepub.com/home/hpc



Abstract

More than 10 years of research related to the development of efficient GPU routines for the sparse matrix-vector product (SpMV) have led to several realizations, each with its own strengths and weaknesses. In this work, we review some of the most relevant efforts on the subject, evaluate a few prominent routines that are publicly available using more than 3000 matrices from different applications, and apply machine learning techniques to anticipate which SpMV realization will perform best for each sparse matrix on a given parallel platform. Our numerical experiments confirm the methods offer such varied behaviors depending on the matrix structure that the identification of general rules to select the optimal method for a given matrix becomes extremely difficult, though some useful strategies (heuristics) can be defined. Using a machine learning approach, we show that it is possible to obtain unexpensive classifiers that predict the best method for a given sparse matrix with over 80% accuracy, demonstrating that this approach can deliver important reductions in both execution time and energy consumption.

Keywords

Sparse numerical linear algebra, sparse matrix-vector product (SpMV), automatic method selection, machine learning, parallel architectures, graphics processing units (GPUs)

1. Introduction

The sparse matrix-vector product (SpMV) is one of the most distinctive examples of sparse linear algebra operations. Among other reasons, its parallel performance is considered to be representative for that of almost every relevant sparse kernel, and the challenges that its parallelization presents are common to many other sparse matrix computations. Moreover, the SpMV is the most important building block in several sparse numerical methods, such as Krylov subspace methods for the solution of systems of linear equations (Davis, 2006).

With the disruptive evolution of the computational power offered by graphics processing units (GPUs), and their adoption by the high performance computing (HPC) community, it is crucial to design algorithms that make the most of these devices. This has motivated considerable research on developing efficient GPU routines for the SpMV in the last decade. Nowadays, there exist several routines that follow different strategies. Unfortunately, many of these realizations render extremely high performance for some particular sparse matrices but considerably lower for others. The main reason is that the specific pattern of nonzeros in the sparse matrix has a strong impact on the sequence of memory accesses, and therefore the

relation between the matrix characteristics and the performance of sparse computational methods is not as straightforward as in the dense case. In consequence, anticipating which routine will work best for a certain sparse matrix is a research topic on its own, especially in massively parallel devices, where different nonzero patterns can imply radically different speeds and attainable parallelism.

This yet increasing plethora of SpMV routines and sparse storage formats motivates the periodic revision of the state-of-the-art, and the experimental evaluation of the available methods. However, this type of evaluation in general occurs when a new algorithm and/or sparse storage format is introduced, and it is often performed using a small collection of sparse matrices, against a small set of third-party routines. Furthermore, the experimental evaluation is

¹ Instituto de Computación (INCO), Facultad de Ingeniería (FING), Universidad de la República (UDELAR), Uruguay

² Depto. de Informática de Sistemas y Computadores, Universitat Politècnica de València, Spain

Corresponding author:

Ernesto Dufrechou, Instituto de Computación (INCO), Facultad de Ingeniería (FING), Universidad de la República (UDELAR), Uruguay.
Email: edufrechou@fing.edu.uy

usually conducted from the point of view of the execution time exclusively, without considering other costs such as the energy consumption. In this sense, the current concern about the energy consumption of data centers makes it important to take into account the energy efficiency of the software. In Anzt et al. (2017), the authors make a relevant effort in this line, though that work is not dedicated exclusively to the SpMV.

In addition to providing a comprehensive evaluation for the characterization of the performance of each SpMV variant, it is also interesting to explore mechanisms that allow to automatically select the best SpMV realization for the execution of each matrix on a given platform. Some efforts in this line include Yeom et al. (2016), where the authors use a machine learning model to select the best preconditioner–solver pair in the context of iterative methods for the solution of sparse linear systems. Also, in Dufrechou et al. (2019) we evaluate black-box machine learning classifiers to select the best GPU triangular linear system solver.

Considering these reasons, in this paper we review the state-of-the-art in SpMV implementations for GPUs, and experimentally evaluate the most important routines publicly available using a very large collection of matrices from different source applications. In addition, we study the application of machine learning techniques to select the best method according to the matrix characteristics. Particularly, we study and compare different methods taking into account the runtime or the energy consumption as selection criteria. The empirical evaluation of our proposal provides strong evidence that it is possible to obtain classifications with almost 90% of accuracy. These results constitute a significant step toward the automatic selection of the best (massively-)parallel routine for any given sparse matrix, which can lead to important runtime and energy consumption savings for many scientific applications.

The rest of the paper is structured as follows. In Section 2 we review the main aspects of the GPU implementations of the SpMV kernel, offering a brief review of related work in this field. Section 3 presents the methodology followed in our study of the automatic method selection, detailing the matrix features selected for the machine learning techniques. Later, in Section 4, we describe our experimental evaluation and highlight the most important results. Finally, Section 5 offers several concluding remarks and summarizes a few lines of future work.

2. Efficient Realizations of SpMV for GPUs

Given a sparse $m \times n$ matrix A and a dense vector x with n components, the SpMV computes the result vector y , of size m , so that $y_i = \sum_{j=1}^n a_{ij}x_j$, $i = 1, 2, \dots, m$, where x_j , y_i are the j -th and i -th components of x, y , respectively, and a_{ij} denotes the (i, j) entry of the matrix A . These products can be computed in any order (or in parallel), taking proper care of race conditions that can occur when accumulating partial results to the solution y . A high level realization of this

operation, using the Compressed Sparse Rows (CSR) storage format (Barrett et al., 1994), is presented in Algorithm 1.

Algorithm 1. Serial computation of sparse matrix-vector multiplication (SpMV) with the sparse matrix A stored in the CSR format. The vector val stores the nonzero values of A by rows; row_ptr contains the indexes that specify the first element of each row in vector val ; and col_idx contains the column index of each element in the original matrix. The nonzero elements within each row are ordered by column index.

```

1 Input:  $row\_ptr, col\_idx, val, x$ 
2 Output:  $y$ 
    $y = 0$ 
   for  $i = 1$  to  $m$  do
     for  $j = row\_ptr[i]$  to  $row\_ptr[i + 1] - 1$  do
        $y[i] = y[i] + val[j] \cdot x[col\_idx[j]]$ 
     end for
   end for

```

A relevant challenge on GPUs, from the performance point of view, is that the memory-bound nature of the SpMV causes the computational units to spend most of the time waiting for data. Additionally, the distribution of the work among the computational units is a major issue in this sort of massively parallel architectures. In this sense, two strategies have been explored: The first one, and perhaps the most straightforward, is to assign different rows of the sparse matrix (or columns depending on the chosen sparse storage format) to different computational units. This strategy has the advantage of being rather simple while effectively avoiding the race conditions that can originate when several computational units accumulate their partial results to the same entry of the output vector. However, as the number of nonzeros of distinct rows can vary significantly, this first option can also lead to major load imbalance problems. This problem is addressed by an alternative family of methods that distribute the workload (tied to the nonzero entries of the matrix) more evenly between computational units. This is done at the expense of introducing more complex synchronization and communication mechanisms between computational units.

2.1. State-of-the-art

In Anzt et al. (2017), the authors made a comprehensive analysis of the performance of sparse kernels on GPUs, discussing the challenges and strategies to reduce communications in the context of sparse computations. The authors of that work state that, even if a 100% efficient memory access can be achieved, a theoretical SpMV kernel will only be able to reach about 2.4% of the peak performance of a K40 GPU. Despite this fact, the relevance of SpMV as a building block for a myriad of numerical methods, and the unstoppable spread of GPUs as components of HPC platforms, have motivated a tremendous amount of work on the subject.

In the early years of GPUs as general-purpose processors, the works by Bell and Garland (2008, 2009) were two of the most relevant contributions regarding the use of this type of accelerators to tackle sparse operations. They introduced two variants of the SpMV for the plain CSR format (*scalar* and *vector*) as well as implementations for the COO, DIA and ELL formats (Chow et al., 1995), and a hybrid format (HYB) that splits the sparse matrix in two partitions, storing the most regular one (in the sense of number of nonzeros per row) in ELL format and the other in COO.

The distribution of the workload among computational units, in both the *scalar* and *vector* implementations, is static and row-based. In the *scalar* routine, each thread of the GPU deals with one row, sequentially computing the products that correspond to that row. This distribution of the workload has a negative impact on the coalescing of memory accesses of the GPU and turns the sequential computation of the longest row into a lower bound for the execution time. The *vector* variant addresses these two problems by assigning a warp of threads to the computation of each row. We offer more details about this routine later on.

In their experimental analysis, the HYB sparse storage format was the fastest implementation for the majority of the tested matrices. The CSR-*vector* variant follows closely, showing an excellent performance on matrices with a large average number of nonzeros per row. In contrast, the *scalar* routine was not competitive in general with the previous realizations, and only delivers acceptable performance for matrices with few nonzero coefficients per row.

A number of works followed these efforts, introducing domain-specific sparse formats and expanding the ecosystem of routines for the implementation of SpMV in GPUs. Many of these efforts propose strategies to store the sparse matrix A that aim to reduce the transfers between the cores and the main memory, leverage the caches and low latency memories of the different devices, and take advantage of the structure for some sparse matrices. The results of these works have led to many successful sparse storage formats, such as BCSR, HYB, ELLPACK, ELLPACK-R, BELLPACK, SELL-P, CSR5, Cocktail, BCOO, BRO, AMB, or Jad. However, the CSR storage format is ubiquitous for representing sparse matrix, and the use of more efficient formats in specific domains frequently implies the conversion from CSR to the new representation. As the overhead related to this transformation sometimes undermines the performance gain in the SpMV routine, considerable research has been devoted to improve the performance of SpMV for the CSR format. The previous survey does not nearly include all the methods that have been proposed over the years. Therefore, in the remainder of this section we highlight some fairly recent works whose source-code or executable routines are publicly available on the web.

2.1.1. CSR-vector. In the original version of this method, presented in Bell and Garland (2009), each matrix row is

processed by a CUDA *warp* of 32 threads. The routine was released later as part of CUSP library and, on more recent versions, a *vector* is defined as a group of 2, 4, 8, 16, or 32 adjacent threads that are in charge of processing a row. Using modular arithmetic, each thread obtains its *vector-id* and *vector-lane-id*, which determines which row and element of that row the thread will process. Once those indexes are obtained, each thread fetches the coefficients and indexes from the sparse matrix, multiplies the coefficients by the appropriate entries of the input vector, and accumulates the result into a register. If the row has more elements than the vector size, the threads that correspond to that vector advance *vector_size* positions on the row and repeat the process. Afterward, the accumulated products are offloaded to shared memory and reduced. Offloading the results to shared memory could be avoided by using warp-shuffle operations, but the latest implementation of the routine found on the library's GitHub page¹ (v 0.5.1) employs the shared memory to perform this reduction.

The routine is developed as a template, which allows it to define the size of the *vector* dynamically when calling the routine. It is important to note that the performance of this routine can be sensitive to the vector size and the variations in the row length. Although a vector size close to 32 exploits parallelism and coalesced access for long rows, it leaves many threads idle for short ones, while vector sizes close to 2 are efficient if many small rows lie together in the matrix, but imply considerable serial work on long rows. A logical choice of vector size is the average row length (number of nonzeros divided by row matrix size, or n_{nz}/m), and we follow this criteria later on in this work.

2.1.2. bhSparse. Liu and Vinter (2015) developed a SpMV routine that performs speculative segmented sums on the GPU, delivering possibly incorrect results. The partial sums are later re-arranged on the CPU to produce the correct output vector.

In this method, the distribution of the workload is nonzero-based, decomposing the nonzero entries of the input matrix into equally sized tiles that will be managed by a certain number of threads. In a first stage, a binary-search is performed to obtain the range of rows spanned by the tile, and a descriptor of the tile is computed using the information in the row-pointers vector. The tile is marked as *dirty* if it contains empty rows. After this process is completed, the nonzero coefficients are multiplied by the input vector, storing the result in scratch-pad memory. A segmented sum is then performed and the values that correspond to the sum of a row are stored in consecutive positions of the output vector. As there can be empty rows (a *dirty* tile), the positions in which these partial sums are stored may be incorrect. In this case, a CPU procedure is then launched to correct the output vector layout in the positions that correspond to dirty tiles. To communicate the output values and the information to correct the wrong speculative results between the CPU and the GPU, the

authors use the GPU Shared Virtual Memory space (or Unified Memory, as it is known in CUDA).

2.1.3. CSR-Adaptive. This procedure, proposed by Great-house and Daga (2014), handles sets of rows with few nonzeros per row by a special algorithm called *CSR-Stream*, while rows with many nonzeros per row are processed by the *CSR-Vector* algorithm. In the *CSR-Stream* stream algorithm, each thread block processes one of the sets. The threads operating on one block first make a coalesced load of the nonzeros of the corresponding set to the shared memory, with each thread loading one nonzero multiplied by the corresponding entry of the input vector. In a second stage, each thread reduces one of the rows in the set sequentially (as in *CSR-Scalar* algorithm).

CSR-adaptive implies a CPU pre-processing stage that aggregates contiguous rows until the set exceeds a certain *block size*. The sets are delimited by array of pointers that point to the start and end of each set of rows in the CSR structure.

2.1.4. Light-SpMV. This method aims to mitigate the load-unbalancing that *CSR-Vector* suffers by providing a dynamic mechanism to distribute the rows among the vectors. As in the current implementation of *CSR-Vector*, the *Light-SpMV* method in Liu and Schmidt (2015, 2018) divides the warp into sections of a fixed number of lanes. Each of these sections or *vectors* is in charge of computing one row. The vector size is adjusted at launch-time depending on the average *nnz* per row. The procedure is as follows:

1. Lane 0 of each warp requests a row index to a global data structure using an atomicAdd operation.
2. The retrieved row index is broadcast to other lanes using warp-shuffle operations. Each vector computes its corresponding row by adding the vector index to the row index obtained by lane 0.
3. While there are still entries to be processed by the warp, each vector loads, multiplies and reduces the corresponding entries, similar to the previously described *CSR-Vector* variant.

2.1.5. Perfect-CSR. PCSR (Gao et al., 2016) consists of two main stages. The first stage launches as many blocks as the number of nonzero entries divided by the block dimensions (i.e., one thread per nonzero entry). Each thread will load the corresponding nonzero, multiply it by the entry of the input vector and store it in a temporary array in global memory.

The second stage launches as many blocks as the number of rows divided by the block dimensions (i.e., one thread per row). The first step of this second stage is to assemble the section of the CSR row-pointers array that corresponds to that block into the shared memory. Then, each thread loads a number of entries of the temporary array computed in the first stage into the shared memory of each thread block.

Using both arrays, each thread reduces one row sequentially writing the output to global memory.

The same authors later proposed an improved variant of this procedure in Gao et al. (2017), which merges the process in a unique kernel that avoids the unnecessary storage and subsequent load of the temporary vector.

2.1.6. Merge-based SpMV. The main idea behind (Dalton et al., 2015) *merge-path*-based SpMV is to divide the workload evenly among all threads, regardless of the characteristics of the sparse matrix. This means that the performance of this SpMV variant is intended to scale with *nnz*, independently of the presence of long and short rows and other particularities of the sparsity pattern.

Applied to SpMV, the *merge-path* is the set of products between an entry of the sparse matrix and an entry of the input vector. These products are theoretically ordered row-wise, and can be regarded as the result of merging the row indexes with the natural numbers from 1 to *nnz*. As there are no data dependencies between these products, any section of the merge path can be computed concurrently with any other.

Given the start and end of its section of the merge path, which is dictated by the *thread id*, each thread first calculates the starting and ending row and nonzero that it has to process. This is done using local binary searches, which are restricted by the size of the section of the merge path that the thread has to process. Once these coordinates are obtained, the thread advances accumulating the multiplication of the nonzeros of its corresponding rows by the elements of the input vector. Every time a complete row is traversed, the thread stores the accumulated value in the output vector and resets the accumulated value to 0. If a row spans more than one section of the merge path, the threads assigned to those sections store a partial value, which is then reduced by a *reduce-by-key* procedure.

2.1.7. MAGMA-Sparse SpMV. MAGMA (Anzt et al., 2014a; Yamazaki et al., 2014) is an open-source project developed by Innovative Computing Laboratory (ICL), at the University of Tennessee, Knoxville, USA, that includes a collection of state-of-the-art GPU accelerated routines for linear algebra, designed to exploit heterogeneous CPU-GPU architectures, providing an interface similar to that of standard libraries like LAPACK and BLAS.²

In addition to the extensive collection of dense linear algebra kernels, MAGMA includes a package to handle sparse computations. The package supports sparse matrix formats such as CSR and ELLPACK, SELL-P and CSR5, and provides a comprehensive set of iterative linear solvers, eigensolvers, and preconditioners. MAGMA uses *cuSparse* for the SpMV in CSR format (version 2.5.2 uses the row-based implementation) while it implements its own GPU kernels for ELLPACK, SELL-P and CSR5.

2.2. Automatic tuning and performance models for the SpMV in GPUs

Several authors have worked on modeling the performance of the SpMV on hardware accelerators considering different sparse formats, often producing automatically tuned routines that can adjust to the characteristics of the sparse matrix and the target GPU. For example Monakov et al. (2010) proposed a new storage format, Sliced-ELLPACK, and provided a mechanism to automatically select the slice size, the thread block, and a parameter related to the reordering of rows. In Choi et al. (2010), the authors derived a performance model to automatically tune parameters such as the block size in the blocked ELLPACK format. Later, Yoshizawa and Takahashi (2012) enhanced Bell and Garland's *vector* variant by automatically selecting the vector size, claiming a 26% overall performance improvement. Xu et al. (2013) also approached the time modeling of the ELLPACK SpMV, but leveraged this to analyze the performance of the kernel, proposing two optimizations, one that consisted on disabling the caching of the coefficient matrix to leave more cache for the input vector, and the second aimed to reduce the memory bandwidth consumption using the Reverse Cuthill-McKee heuristic (Cuthill and McKee, 1969), diminishing also the range of integers required to store the column indexes of the matrix and enabling the use of a smaller integer representation.

An additional field of interest is the automatic selection between sparse matrix representation and SpMV routine. In Guo et al. (2014), the authors tackle this problem by developing a model of the execution time of the CSR, COO, ELL, and HYB kernels. Using this model, they later apply dynamic programming to determine whether, for a certain matrix, higher performance can be attained by partitioning the matrix into several blocks of rows, each stored with a different sparse storage layout. A similar effort is found in Li et al. (2015), where the Probability Mass Function is used to estimate the performance of the SpMV kernel with the CSR, COO, ELL, and HYB matrix formats. The developed model takes the hardware characteristics into account. However, the experimental evaluation is performed using only one hardware platform and the experimental results are expressed as relative values (i.e. speedups) which offsets the hardware effects. Later, Guo and Lee (2016) present a similar approach in their framework for SpMV performance on GPUs.

Elafrou et al. (2015) present a radically different strategy. They start by identifying and defining several categories of problems according to their performance bottlenecks. Later, the work offers both an heuristic method and a machine learning classifier. Finally, specific optimizations are applied.

Other interesting efforts related with this topic are the article by Williams et al. (2007) where the authors study some optimization techniques for the SpMV kernel over several hardware platforms; the proposal by Erguiz et al. (2017) with advances over the automatic selection of

different sparse triangular linear solvers on GPU; and the work by Barreda et al. (2020) which offers a performance modeling of the SpMV kernel via convolutional neural networks with ARM as the target hardware platform.

In general, the reviewed works (especially Guo and Lee, 2016; Guo et al., 2014, Li et al., 2015;) focus on predicting the performance of SpMV with different storage formats, such as COO, CSR, ELL and HYB formats, to then select the best storage layout for each matrix or adjust some algorithmic parameters. In contrast, our work studies several fine-tuned implementations for CSR (and others formats), and the performance improvements that can be obtained by selecting the right implementation for each matrix. A machine learning classifier is then used to make such decision automatically, considering both execution time and energy consumption. Additionally, our proposal is transparent to the target hardware platform.

3. Automatic method selection

The purpose of this work is twofold. On the one hand, we are interested in analyzing the performance of some of the most relevant GPU routines for SpMV (that employ the CSR storage format), and study which method is the best according to the characteristics of the sparse matrices. On the other hand, we intend to advance in the development of a tool that can a priori estimate which SpMV realization will deliver the best performance.

In this work, we use a supervised machine learning technique for the task. Unlike previous efforts reviewed in Section 2.1, we do not attempt to estimate neither the execution time nor the energy consumption of the routines. Instead, we train the machine learning model with a number of selected features of the sparse matrix as predictors, and the routine that obtains the best execution time/energy consumption as the response.

In Dufrechou et al. (2019), we evaluated different classifiers from the *Classification Learner App* of Matlab[®] when dealing with routines for the solution of sparse triangular systems. The results of that work favored the Bagged Trees Classifier (Breiman, 1996), which offered accurate predictions at an acceptable speed. We can expect similar results regarding the SpMV kernel, and therefore we evaluate our proposal using the same classifier.

To select the features of the sparse matrices that will serve as predictors, we consider their relation with different parameters of the target methods, such as the number of computational units, the size of the workload, data locality, etc. As a starting point, we analyzed the following nine features related with the parallel performance of the SpMV:

- **Number of rows (m):** In an important number of SpMV methods, the number of rows is deeply linked with the distribution of the workload. For example, *cusp_vect* assigns one warp to each row.
- **Number of columns (n):** As the storage format is CSR, the number of columns is not directly related

with the behavior of the different variants. However, it can be related indirectly with the consumption of auxiliary memory or the length of rows and the data locality.

- **Aspect ratio (m/n):** Short and wide matrices can present a strong variation in row lengths which can cause load imbalance issues among threads or warps. Works such as Bell and Garland (2008) and Dalton et al. (2015) have observed that the performance of certain kernels is strongly sensitive to this factor.
- **Number of nonzeros (nnz):** The number of nonzeros is a fair estimation of the amount of work to be performed, since at least a multiplication and an addition are necessary for each nonzero.
- **Maximum number of nonzeros per row (nnz_{max}):** The presence of rows much longer than the average can be an important performance problem for kernels that organize the workload row-wise.
- **Minimum number of nonzeros per row (nnz_{min}):** This feature does not seem determinant, but we include it for completeness.
- **Average number of nonzeros per row (nnz_{avg}):** In kernels that organize the workload row-wise, this feature estimates the average workload per computational unit.
- **Standard deviation of nonzeros per row (nnz_{std}):** Together with nnz_{max} and nnz_{avg} , this feature aims to estimate the load imbalance between computational units during the execution of the method.
- **Average bandwidth (bw_{avg}):** The bandwidth of each row is the region of the input vector that will be processed by the rows. Therefore, the average bandwidth can be related to data locality.

It is important to remark that some of these features are more difficult to compute than others. While m , n , m/n , nnz and nnz_{avg} can be obtained in $\mathcal{O}(1)$ from the matrix data structure, computing $nnz_{min/max}$, nnz_{std} and bw_{avg} have a cost $\mathcal{O}(n)$. Therefore, it is interesting to evaluate the performance of the models trained only with the features that are inexpensive to extract.

4. Experimental evaluation

This section describes the numerical evaluation of the selected SpMV routines for the CSR format, and the experiments conducted to assess the performance of the supervised-learning classifiers on the automatic method selection.

Additionally we include a preliminary study to evaluate the application of our procedure when considering other sparse matrix formats.

4.1. Hardware platform

The results were obtained in two different hardware platforms:

- A server with an Intel(R) Core(TM) i7-4770 CPU at 3.40 GHz and 160 GiB of RAM connected to an NVIDIA GeForce GTX 1080Ti GPU with 11 GiB of RAM. The operating system is CentOS Linux 7 (Core) and the CUDA Toolkit for the GPU is version 9.2.
- A server with an Intel(R) Core(TM) i7-7700K CPU at 4.20 GHz and 64 GiB of RAM connected to an NVIDIA GeForce RTX 2080Ti GPU with 11 GiB of RAM. The operating system is Ubuntu 16.04 and the CUDA Toolkit for the GPU is version 10.1.

It is necessary to remark that the GPUs employed in the experiments have relatively low double-precision performance. However, as the SpMV is a memory-bound procedure, the performance restriction is diluted in this computing context. We estimate that the relative performance between the different methods will not change drastically on a server GPU with full support for double-precision arithmetic.

For the classification experiments, we used MATLAB[®] 2018a.

4.2. Test set

The experimental evaluation was carried out using a dataset of 3,005 matrices, where 925 are square, 1,060 have more rows than columns, and 1,020 have more columns than rows. All the square matrices, and 230 of the non-square cases were taken from the subset of SuiteSparse Matrix Collection³ matrices with at least 100,000 nonzero entries. The remaining 1,850 matrices were generated to expand our dataset and obtain instances with diverse aspect ratios (data augmentation). Concretely, for each square matrix in the original set, we generated two additional matrices: one removing a random number of columns so the result is short and wide, and the transpose of this matrix (which is tall and thin). The aspect ratio (number of rows divided by the number of columns) of the generated matrices is distributed uniformly between 0.0003 and 1.

For the energy measurements, we chose to measure only the GPU energy consumption, as most of the evaluated methods do their computations entirely on the GPU. We used the CUDA NVML library to query the GPU for power measurements. The energy results were taken as the average power obtained after 1 second of repeatedly executing each kernel, multiplied by the runtime of the kernel measured independently. The reason is that the polling thread that queries the power readings may impact the runtime of the kernel and, consequently, the total energy consumption.

The runtimes of our experiments were taken as the average of 100 independent executions, recording also the standard deviation.

Additionally, we employed IEEE double-precision floating point representation and arithmetic for all the experiments.

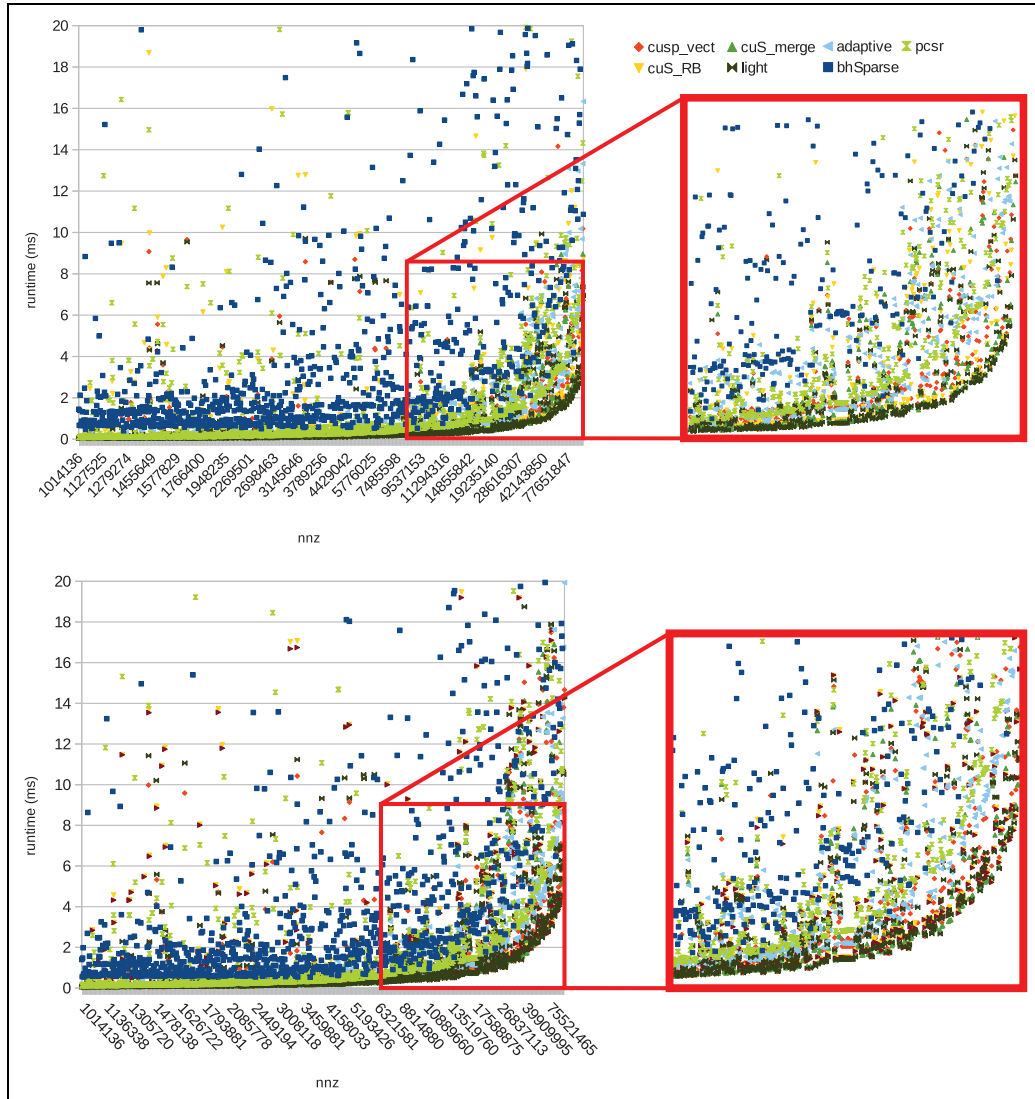


Figure 1. Scatter plots of the runtimes obtained for the execution of all the selected variants of the SpMV on the RTX2080Ti (top) and the GTX1080Ti (bottom). The plots only show the results for matrices with $nnz > 1,000,000$.

4.3. Experimental results

The first step in the empirical evaluation of our proposal is the execution of all the selected variants of SpMV and a preliminary assessment of results. Our initial experiment compares the runtimes of the *cuSparse* CsrMV routine with the row-based and merge-path algorithms (which we refer to as *cuS_RB* and *cuS_merge*, respectively); the CUSP implementation of *CSR-Vector* (*cusp_vect*); an implementation of Liu and Vinter (2015) (*bhSparse*) published by the authors in their GitHub repository⁴; and our own implementations of Liu and Schmidt (2018) (*light*), Gao et al. (2016) (*pcsr*), and Greathouse and Daga (2014) (*adaptive*), based on codes found online.⁵

Figure 1 plots the runtime of the different variants against the nnz of the matrix in the two hardware platforms. Although it can be observed that the runtimes obtained in the RTX2080Ti GPU are generally lower than those in the

GTX1080Ti, the results show similar tendencies. In general, nnz appears to be a good estimator of the total workload and hence the runtime of the SpMV regardless of the variant and platform of choice. Considering the individual variants, it can be observed that *bhSparse* performs significantly worse than the rest of methods in the majority of cases. Even though it is not the worst variant in all cases, it is never the best one. For this reason we drop *bhSparse* from the subsequent evaluation. Something similar occurs with *pcsr*, but its runtimes are not clearly worse than those of other variants, especially when nnz is not too large. In addition to the general trend, the figure also shows the presence of outliers that in many occasions take considerably more runtime than the average for a certain case, especially for *light*, *pcsr* and *cuS_RB* (leaving out the *bhSparse* routine).

Table 1 shows the number of times each method outperforms the rest from the points of view of runtime, power

Table 1. Number of times that each method is the best form the points of view of runtime, power, and energy consumption.

		<i>cusp_vect</i>	<i>cuS_RB</i>	<i>cuS_merge</i>	<i>adaptive</i>	<i>light</i>	<i>pcsr</i>
GTX1080Ti	Time	1448	389	517	262	384	5
	Power	1	574	331	661	271	1167
	Energy	850	583	394	616	529	32
RTX2080Ti	Time	1178	551	671	106	497	2
	Power	66	266	5	1338	309	1021
	Energy	959	604	641	294	501	6

Table 2. Time and energy spent by each variant to perform the SpMV across the entire dataset. The last column is the total time and energy spent when executing always the best variant for each matrix.

Total	<i>cusp_vect</i>	<i>cuS_RB</i>	<i>cuS_merge</i>	<i>adaptive</i>	<i>light</i>	<i>pcsr</i>	Optimal
GTX1080Ti							
Time (sec.)	1.76	3.55	1.45	1.80	1.93	3.92	1.30
Energy (kj)	312.92	466.46	273.72	296.21	305.21	520.67	240.03
RTX2080Ti							
Time (sec.)	1.15	2.44	0.76	1.32	1.14	3.30	0.69
Energy (kj)	236.24	389.76	179.96	252.20	220.98	517.35	158.76

dissipation and energy consumption, in both computing platforms. In GTX1080Ti, *cusp_vect* was the fastest variant for 48% of the tested matrices, and the most energy efficient for 28%. The power consumption of *cusp_vect*, however, is the highest for almost all the evaluated matrices. These results are followed by *cuS_merge*, *cuS_RB* and *light*, which together are the fastest for 42% of the matrices (57% on the RTX2080Ti), and consume the least energy for 50% (58%) of the matrices. In turn, *adaptive* was the most energy-efficient variant for 20% of the matrices, though it is the fastest in only 8% of the cases. Unlike *cusp_vect*, its energy efficiency originates in a generally lower power consumption. This behavior is intensified on the RTX2080Ti, where it is the fastest variant only in 4% of the cases and the least power consuming in 45% of the cases. However, that is not enough to outperform the rest in energy consumption, being the most energy efficient for about 10% of the matrices only.

Although *cusp_vect* outperforms the other variants most of the times, it is neither the variant that took least total time nor the one that consumed least energy. In fact, as shown in Table 2, *cuS_merge* has a total runtime (to compute the SpMV operation for all evaluated matrices) of 1.45 s while *cusp_vect* took 1.76 s, and the total energy consumption for *cusp_vect*, 312.92 kJ, is only exceeded by *cuS_RB* and *pcsr*.

Table 3 takes into account the optimal total time and energy (which is the time or energy required to compute the entire dataset if the fastest or most energy-efficient variant is executed for each matrix) and shows the ratio between aggregated time/energy spent by each method, and the aggregated time/energy spent by the best one, computed as

$$R_T = \frac{\sum T_{method}}{\sum T_{best}}$$

Table 3. R_T and R_E for each of the methods.

	<i>cusp_vect</i>	<i>cuS_RB</i>	<i>cuS_merge</i>	<i>adaptive</i>	<i>light</i>	<i>pcsr</i>
GTX1080						
R_T	1.35	2.72	1.11	1.38	1.47	3.00
R_E	1.29	1.92	1.19	1.22	1.26	2.15
RTX2080						
R_T	1.67	3.54	1.10	1.92	1.66	4.79
R_E	1.49	2.46	1.13	1.59	1.39	3.26

and

$$R_E = \frac{\sum E_{method}}{\sum E_{best}}$$

This shows that choosing *cuS_merge* by default (i.e., always) results in only 11% (10% on the RTX2080Ti) more time and 19% (13%) more energy than choosing the individual (matrix-specific) best method.

The remarkable performance showed *cuS_merge* in Tables 2 and 3 seems unexpected, considering that *cusp_vect* is the best variant for the majority of cases. The principal explanation of this behavior is that *cuS_merge* outperforms *cusp_vect* for the largest matrices. The upper part of Table 4 shows the percentage of times each method is the fastest for each quintile of matrices according to the *nnz* value, while the lower part shows the value of R_T for the matrices in each quintile. As the general behavior of the variants on each quintile is similar in the two platforms, we use the results obtained in the GTX1080Ti GPU for the following analysis.

According to Table 4, the number of times *cuS_merge* is the fastest variant steadily scales with *nnz*, taking *cusp_vect* place. As a consequence, if the first quintile is

Table 4. Percentage of times that each version is the fastest (top) and value of R_T (bottom) for the matrices on each quintile computed according to nnz . The runtime values correspond to the GTX1080Ti.

quintile	$nnz >$	cuS_vect	cuS_RB	cuS_merge	$adaptive$	$light$	$pcsr$
% of best							
1st	3497020	21.17	6.00	42.33	11.83	18.83	0.00
2nd	878922	33.06	9.14	25.08	4.15	28.24	0.17
3rd	329472	49.17	14.50	11.33	9.33	15.83	0.00
4th	139806	58.24	25.62	4.99	9.98	1.00	0.00
5th	0	79.37	8.99	2.33	8.32	0.00	0.67
R_T							
1st	3497020	1.22	2.11	1.09	1.34	1.30	2.50
2nd	878922	2.44	7.49	1.11	1.55	2.92	8.21
3rd	329472	3.70	13.99	1.23	1.37	4.49	9.02
4th	139806	1.58	5.63	1.44	1.37	2.24	5.14
5th	0	1.62	3.94	1.76	2.65	2.41	4.21

considered, cuS_merge is the fastest variant 42% of the times, while cuS_vect drops to 21%. Moreover, the bottom half of the table shows that, although cuS_vect is the fastest variant on most matrices in the smaller quintiles, cuS_merge is the variant that is closer to the optimal runtime. This suggests that, on each quintile, cuS_merge performs better than cuS_vect for the largest and most irregular matrices.

These results confirm the remarkable scalability of the $merge-path$ algorithm, and suggests that the row-wise distribution of the workload of cuS_vect is not so efficient for large matrices.

4.3.1. Automatic selection of the best variant. Although the results presented so far show a clear dominance of the cuS_merge implementation, they also show that, even in scenarios where one method excels, there are matrices for which other methods can achieve significantly higher performance (and/or lower power/energy dissipation). Therefore, we now evaluate if the previous results can be improved by automatically executing the SpMV method that is predicted to be the best by a Bagged Trees ensemble classifier.

The results that follow are the product of performing five-fold cross validation of our classifier on the set of 3,005 matrices described previously. This means that the dataset is randomly divided into five equally sized partitions, which are iteratively used as the test set (1/5) for the model trained with the rest of the partitions (4/5). The final accuracy result is the average of the accuracy values obtained for each of the five test sets.

The Bagged Trees classifier consists on an ensemble of 30 decision trees, with a maximum of 2,417 non-leaf nodes. Each tree is trained with a bootstrapped copy of the dataset, and each split of the random forest algorithm is made considering a random subset of the predictors.

The left part of Table 5 shows the performance of the classifier predicting the best SpMV implementation in terms of execution time. The first row, which reports the result of training the model with all the features, shows that

Table 5. Performance of the Bagged Trees Classifier trained with the fastest method as response (left) and the most energy efficient (right), and different sets of features, on the GTX1080Ti (top) and the RTX2080Ti (bottom) GPUs.

Features	Time as response			Energy as response		
	Acc.	$\frac{\sum T_{pred}}{\sum T_{best}}$	$\frac{\sum E_{pred}}{\sum E_{best}}$	Acc.	$\frac{\sum E_{pred}}{\sum E_{best}}$	$\frac{\sum T_{pred}}{\sum T_{best}}$
GTX1080Ti						
all features	82%	1.03	1.05	79%	1.03	1.06
all cheap feat.	69%	1.19	1.14	65%	1.15	1.28
add min nnz	71%	1.14	1.11	67%	1.10	1.19
add max nnz	82%	1.05	1.06	78%	1.03	1.06
add std nnz	79%	1.05	1.06	76%	1.04	1.07
add avg bw	73%	1.15	1.11	68%	1.11	1.21
RTX2080Ti						
all features	89%	1.01	1.02	84%	1.03	1.04
all cheap feat.	71%	1.24	1.14	68%	1.14	1.25
add min nnz	74%	1.28	1.17	72%	1.17	1.29
add max nnz	89%	1.02	1.03	83%	1.03	1.03
add std nnz	86%	1.03	1.03	81%	1.03	1.03
add avg bw	73%	1.27	1.16	71%	1.13	1.23

the classifier correctly chose the best method for 82% of the matrices. Moreover, the total runtime was only 3% (1% running the models with the RTX2080Ti dataset) higher than optimal, and the energy overhead was of 5% (2%). The second row shows the result of training only with the m , n , nnz , m/n and (nnz_{avg}) parameters. Here, both the accuracy of the model and the resulting overhead in time and energy are worse than a default solution that always chooses cuS_merge . The following rows show the effect of adding one more feature to the set in the second row. From these results, it seems clear that the row with most nonzeros and the variation in the length of the rows are key predictors of the performance of the methods, and training with these features improves the quality of the classifier.

Finally, the right part of Table 5 shows the performance of the model to predict the best SpMV implementation in

terms of energy consumption. The results suggest that training the model with the most energy-efficient method as the response leads to similar results. Nonetheless, for the RTX2080Ti GPU, training the models with the fastest method as response reaches even better energy consumption results than training with the most energy efficient as response.

4.4. Other sparse formats

Although our study is focused on delivering a comparison between varied realizations of SpMV for the CSR format, the same approach can be applied to assess implementations for different sparse representations. As matrices are often stored in CSR,⁶ utilizing other formats implies a conversion overhead, which hopefully is offset by the gains obtained in the application of the SpMV kernel. Unfortunately, this is not always the case, and some of these formats are known to render low performance in certain situations. In this sense, our approach can help to predict when a change of format can be beneficial.

As mentioned in Section 2.1, the number of sparse matrix representations proposed in the literature is considerable, and for some formats there is more than one software library providing an SpMV implementation. For this reason, in this experiment we selected two formats that are fairly well-known and are implemented in the MAGMA library: ELLPACK (Bell and Garland, 2009) and SELL-P (Anzt et al., 2014b).

The ELLPACK format (*Ell*) stores a sparse matrix of n rows and a maximum of k nonzeros per row in an $n \times k$ dense matrix, where each row i contains the nz_i nonzero elements of the corresponding row of the sparse matrix padded with $k - nz_i$ zeros at the end. This dense matrix is stored in column-major format so that it can be processed more efficiently by vector processors. Although this format has proved to outperform CSR in several cases (Bell and Garland, 2009), it has the disadvantage of presenting a large memory overhead on sparse matrices with an irregular pattern of nonzeros per row.

The SELL-P (*SellP*) is a variant of the SELL- c format (Kreutzer et al., 2013), which aims to reduce the memory overhead of ELLPACK by slicing the matrix into blocks of rows of size c , which are stored independently in ELLPACK. The memory overhead of SELL- c is thus determined by the longest row of each block instead of the longest row of the matrix. SELL-P introduces, as the key enhancement, a padding of the blocks so that the length of the rows becomes a multiple of the computational units that will later compute the SpMV for each block. A more detailed explanation of both the format and the SpMV kernel is given in Anzt et al. (2014b).

In the following experiment we executed the double-precision *Ell* and *SellP* kernels of MAGMA v2.5.2 on the GTX1080Ti GPU. The runtimes and power measurements correspond to those of the SpMV kernel, and do not include neither the conversion overhead from CSR to ELL or

SELL-P, nor the memory transfers of matrices and vectors. The dataset for this experiment contains 2,004 matrices, as some kernels were unable to terminate successfully for some matrices.

Table 6 shows the percentage of matrices for which each implementation is the best in terms of execution time and energy consumption, as well as the corresponding values of R_T and R_E , for each quintile of the dataset divided according to nnz . The table shows a remarkably low performance of the *Ell* relative to the other kernels, which seems to improve with the number of nonzero of the matrices. For this dataset, this kernel was the fastest for only 5% of the matrices, and the most energy efficient in 3%. This is reasonable since, for small matrices, it is likely that the memory padding does not result in a significant performance gain.

Conversely, *SellP* was the fastest kernel for 33% of the matrices, and the most energy efficient for 50%. However, it still presents higher R_T than most of the other implementations, especially in the smallest matrices.

Regarding the application of the Bagged Trees classifier to select between implementations, Table 7 shows that the machine learning model delivers high relative performance, for both time and energy, using only the cheap predictors plus any of the predictors related to the row length.

5. Final remarks and future work

We have reviewed the state-of-the-art for the SpMV implementation in GPUs, and studied the application of machine learning to select the best performing method regarding runtime and energy consumption.

Our experimental evaluation of the selected variants, performed using more than 3,000 matrices (1,000 square matrices from the SuiteSparse Matrix Collection and 2,000 rectangular ones generated from the former), on two modern GPUs (a GTX1080Ti and a RTX2080Ti), shows a clear advantage of the merge-path SpMV of Dalton et al. (2015), both from the standpoint of execution time and energy consumption.

Despite this clear advantage, we prove that leveraging a machine learning model to select the best method for each matrix can be considerably better than always choosing the same variant, especially in scenarios where the SpMV kernel has to be computed a large number of times with the same sparse matrix (as is the case, e.g. in an iterative solver).

We have applied the Bagged Trees classifier with cross validation to our matrix dataset. To select the matrix features employed as predictors in the classifier, we not only considered their ability to separate the dataset into different groups, but also the cost of extracting the properties from the sparse matrix.

The empirical evaluation shows that our selection procedure is able to reach accuracy values close to 90% for both runtime and energy consumption. Furthermore, the computation of accuracy metrics specific to our problem allows observing that using the predicted methods to

Table 6. R_T (top) and R_E (bottom) for the matrices on each quintile computed according to nnz. The runtime values correspond to the GTX1080Ti.

nnz >	<i>cusv_vect</i>	<i>cuS_RB</i>	<i>cuS_merge</i>	<i>Ell</i>	<i>SellP</i>	<i>adaptive</i>	<i>light</i>	<i>pcsr</i>
% best in execution time								
1.79e+06	8.48	2.99	18.45	0.5	49.13	1	19.45	0
5.52e+05	19.7	2.99	6.48	3.99	48.63	2.74	15.46	0
2.36e+05	32.5	9.5	7.5	10.75	32.25	4.75	2.75	0
1.20e+05	36.66	18.95	3.49	8.73	25.69	6.48	0	0
	68.33	5.49	2.24	2	5.49	14.96	0	1.5
Total	33.13	7.98	7.63	5.19	32.23	5.99	7.53	0.29
R_T								
1.79e+06	1.5	1.47	1.27	7.93	1.52	2	1.31	3.79
5.52e+05	1.45	2.36	1.41	22.34	4.02	1.97	1.59	7.62
2.36e+05	1.44	3.49	1.5	38.05	4.92	1.65	1.99	7.24
1.20e+05	1.77	4.67	1.8	46.87	6.36	1.59	2.51	6.34
	1.99	4.54	2.06	52.77	5.2	1.56	3.04	5.33
Total	1.50	1.80	1.32	12.76	2.13	1.96	1.43	4.44
% best in energy consumption								
1.79e+06	2.24	2.49	6.48	1.75	70.82	1.75	14.46	0
5.52e+05	6.98	2.74	5.24	1.5	62.59	7.73	13.22	0.25
2.36e+05	14.5	6.75	10	3	52.5	6.5	7	0
1.20e+05	22.19	20.2	3.74	6.48	39.65	8.23	0.5	0
	42.64	6.23	2	4.99	26.93	18.95	0	2
Total	17.71	7.58	5.49	3.49	50.05	8.28	6.98	0.40
R_E								
1.79e+06	1.82	1.57	1.54	7.23	1.27	1.89	1.44	3.05
5.52e+05	1.7	2.08	1.56	20.95	2.59	1.87	1.6	5.09
2.36e+05	1.61	2.79	1.56	40.38	3.36	1.62	1.84	5.18
1.20e+05	1.82	3.81	1.77	58.16	4.62	1.57	2.25	5.05
	1.92	4.02	1.96	74.15	4.13	1.49	2.72	4.49
Total	1.80	1.75	1.55	11.94	1.59	1.86	1.50	3.40

compute SpMV on the entire dataset, yields similar runtime and energy consumption to that obtained by using the best method for each matrix. Specifically, the difference with respect to the optimal time and energy consumption can be as low as 1% and 2%, respectively. Additionally, we show that several of the most expensive features (from a computational point of view) can be discarded without a significant impact on the quality of the results.

Complementing our study of CSR SpMV realizations, we have explored the extension of our approach to other sparse matrix formats. The experimental results showed that GPU-friendly formats, such as ELL or SELL-P, can be outperformed by CSR in many cases, and that our machine learning procedure can be useful to predict whether it is convenient to use these formats for a given problem.

As part of future work we identified several lines, such as advancing in the identification of a cheap heuristic to estimate the maximum row length, including different

Table 7. Performance of the Bagged Trees Classifier trained with the fastest method as response (left) and the most energy efficient (right), and different sets of features, on the GTX1080Ti GPU.

Features	Time as response			Energy as response		
	Acc.	$\frac{\sum T_{pred}}{\sum T_{best}}$	$\frac{\sum E_{pred}}{\sum E_{best}}$	Acc.	$\frac{\sum E_{pred}}{\sum E_{best}}$	$\frac{\sum T_{pred}}{\sum T_{best}}$
GTX1080Ti						
all features	78%	1.02	1.07	78%	1.07	1.02
all cheap feat.	62%	1.29	1.26	64%	1.27	1.31
add min nnz	65%	1.18	1.16	65%	1.27	1.30
add max nnz	76%	1.06	1.11	77%	1.09	1.04
add std nnz	74%	1.07	1.09	74%	1.10	1.07
add avg bw	66%	1.36	1.33	66%	1.37	1.42

hardware platforms in the experimental evaluation to study the relation between the hardware characteristics and the machine learning procedure, performing a more complete

analysis of SpMV implementations for other sparse matrix formats, and implementing a computational tool that computes the best method and calls the respective routine without human intervention.

Acknowledgments

The researchers from Universidad de la República were supported by the UDELAR and PEDECIBA. We acknowledge the ANII – MPG Independent Research Groups: “Efficient Heterogeneous Computing” with the CSC group.


Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: ES Quintana-Ort was supported by project TIN2017-82972-R of the MINECO and FEDER.

ORCID iD

Ernesto Dufrechou  <https://orcid.org/0000-0003-4971-340X>

Notes

1. <https://cusplibrary.github.io>.
2. We note that MAGMA-sparse is being gradually replaced by the more modern library Ginkgo, available at <https://ginkgo-project.github.io>.
3. <http://faculty.cse.tamu.edu/davis/suitesparse.html> (accessed in February 2020).
4. https://github.com/bhSPARSE/Benchmark_SpMV_using_CSR.
5. <https://github.com/poojahira/spmv-cuda>.
6. We note that SuiteSparse also uses COO format, and in some applications the matrix can be constructed in the appropriate format. Moreover, for iterative solvers the cost of the format conversion can be easily amortized by the cost of several applications of the SpMV kernel.

References

- Anzt H, Sawyer W, Tomov S, et al. (2014a) Optimizing Krylov subspace solvers on graphics processing units. In: *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*. Phoenix, AZ, 19–23 May 2014. Phoenix, AZ: IEEE.
- Anzt H, Tomov S and Dongarra JJ (2014b) Implementing a sparse matrix vector product for the SELL-C / SELL-C- σ formats on Nvidia GPUS. Available at: <https://www.icl.utk.edu/publications/implementing-sparse-matrix-vector-product-sell-csell-c-%CF%83-formats-nvidia-gpus>
- Anzt H, Tomov S and Dongarra JJ (2017) On the performance and energy efficiency of sparse linear algebra on GPUS. *The International Journal of High Performance Computing Applications* 31: 375–390.
- Barreda M, Dolz MF, Castaño MA, et al. (2020) Performance modeling of the sparse matrix-vector product via convolutional neural networks. *The Journal of Supercomputing* 76(11): 8883–8900.
- Barrett R, Berry M, Chan TF, et al. (1994) *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PE: Society for Industrial and Applied Mathematics.
- Bell N and Garland M (2008) Efficient sparse matrix-vector multiplication on CUDA. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.
- Bell N and Garland M (2009) Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Pinfold W (ed) *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09. New York, NY, USA: Association for Computing Machinery. ISBN 9781605587448.
- Breiman L (1996) Bagging predictors. *Machine Learning* 24(2): 123–140.
- Choi JW, Singh A and Vuduc RW (2010) Model-driven autotuning of sparse matrix-vector multiply on GPUS. *SIGPLAN Notices* 45(5): 115–126.
- Chow E, Chow E and Saad Y (1995) Tools and libraries for parallel sparse matrix computations.
- Cuthill E and McKee J (1969) Reducing the bandwidth of sparse symmetric matrices. In: *Proceedings of the 1969 24th National Conference*, ACM '69. New York, NY: Association for Computing Machinery. ISBN 9781450374934, pp. 157–172.
- Dalton S, Baxter S, Merrill D, et al. (2015) Optimizing sparse matrix operations on GPUS using merge path. In: *2015 IEEE International Parallel and Distributed Processing Symposium* Hyderabad, India, 25–29 May 2015, pp. 407–416. DOI: 10.1109/IPDPS.2015.98.
- Davis TA (2006) *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA: Society for Industrial and Applied Mathematics. ISBN 0898716136.
- Dufrechou E, Ezzatti P and Quintana-Ort ES (2019) Automatic selection of sparse triangular linear system solvers on GPUS through machine learning techniques. In: *31st International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2019*, Campo Grande, Brazil, 15–18 October 2019, pp. 41–47.
- Elafrou A, Goumas GI and Koziris N (2015) A lightweight optimization selection method for sparse matrix-vector multiplication. *CoRR* abs/1511.02494. URL <http://arxiv.org/abs/1511.02494>.
- Erguiz D, Dufrechou E and Ezzatti P (2017) Assessing sparse triangular linear system solvers on GPUS. In: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops, SBAC-PAD Workshops*, Campinas, Brazil, 17–20 October 2017. IEEE Computer Society, pp. 37–42.
- Gao J, Qi P and He G (2016) Efficient CSR-based sparse matrix-vector multiplication on GPU. *Mathematical Problems in*

- Engineering* 2016. DOI: <https://doi.org/10.1155/2016/4596943>.
- Gao J, Wang Y and Wang J (2017) A novel multi-graphics processing unit parallel optimization framework for the sparse matrix-vector multiplication. *Concurrency and Computation: Practice and Experience* 29(5): e3936.
- Greathouse JL and Daga M (2014) Efficient sparse matrix-vector multiplication on GPUS using the CSR storage format. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. New York, NY: IEEE Press. ISBN 9781479955008, pp. 769–780.
- Guo P and Lee CW (2016) A performance prediction and analysis integrated framework for SPMV on GPUS. *Procedia Computer Science* 80(C): 178–189.
- Guo P, Wang L and Chen P (2014) A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on GPUS. *IEEE Transactions on Parallel and Distributed Systems* 25(5): 1112–1123.
- Kreutzer M, Hager G, Wellein G, et al. (2013) A unified sparse matrix data format for modern processors with wide SIMD units. *ArXiv abs/1307.6209*.
- Li K, Yang W and Li K (2015) Performance analysis and optimization for SPMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems* 26(1): 196–205.
- Liu W and Vinter B (2015) Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing* 49(C): 179–193.
- Liu Y and Schmidt B (2015) LightSpMV: faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUS. In: *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Toronto, ON, Canada, 27–29 July 2015, pp. 82–89. DOI:10.1109/ASAP.2015.7245713.
- Liu Y and Schmidt B (2018) LightSpMV: faster CUDA-compatible sparse matrix-vector multiplication using compressed sparse rows. *Journal of Signal Processing Systems* 90(1): 69–86.
- Monakov A, Lokhmotov A and Avetisyan A (2010) Automatically tuning sparse matrix-vector multiplication for GPU architectures. In: Patt YN, Foglia P, Duesterwald E, et al. (eds) *High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 111–125. ISBN 978-3-642-11515-8.
- Williams S, Oliker L, Vuduc R, et al. (2007) Optimization of sparse matrix-vector multiplication on emerging multi-core platforms. In: *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. Reno, NV, USA, 10–16 November 2007, pp. 1–12. DOI:10.1145/1362622.1362674 .
- Xu S, Xue W and Lin HX (2013) Performance modeling and optimization of sparse matrix-vector multiplication on Nvidia CUDA platform. *The Journal of Supercomputing* 63(3): 710–721.
- Yamazaki I, Anzt H, Tomov S, et al. (2014) Improving the performance of CA-GMRES on multicores with multiple GPUS. In: *IPDPS 2014*, Phoenix, AZ, 19–23 May 2014. Phoenix, AZ: IEEE.
- Yeom J, Thiagarajan JJ, Bhatele A, et al. (2016) Data-driven performance modeling of linear solvers for sparse matrices. In: *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, Salt Lake City, UT, USA, 14 November 2016, pp. 32–42. DOI:10.1109/PMBS.2016.009 .
- Yoshizawa H and Takahashi D (2012) Automatic tuning of sparse matrix-vector multiplication for CRS format on GPUS. In: *2012 IEEE 15th International Conference on Computational Science and Engineering*, Nicosia, Cyprus, 5–7 December 2012, pp. 130–136. DOI:10.1109/ICCSE.2012.28.

Author biographies

Ernesto Dufrechou earned his PhD degree from the Universidad de la República in 2018, elaborating his theses under the tuition of Dr Pablo Ezzatti and Dr Enrique Quintana-Ort. Currently, he teaches Numerical Linear Algebra and General-Purpose GPU programming courses at the university. His current research interests include the application of parallel computing techniques to accelerate numerical methods and matrix computations, sparse numerical linear algebra and energy-aware computing. In particular, most of his research is devoted to the use of accelerators and heterogeneous platforms to boost the performance of NLA kernels.

Pablo Ezzatti received his PhD in Computer Science in 2011, from Universidad de la República (UDELAR), Uruguay. He is also a professor and researcher of that university, where he leads the Heterogeneous Computing Laboratory (HCL) research group. His principal research topics involve the use of heterogeneous hardware platforms to accelerate scientific computing and numerical linear algebra problems, energy-aware computing, transprecision computing and bioinformatics, among others.

Enrique S Quintana-Ort received the bachelor's and PhD degrees in computer sciences from the Universidad Politécnica de Valencia, Spain, in 1992 and 1996, respectively. He is currently a Professor in computer architecture at Universitat Politècnica de València, Spain. He has published more than 200 papers in international conferences and journals, and has contributed to software libraries like PLiC/SLICOT, MAGMA, FLARE, BLIS, and libflame for control theory and parallel linear algebra. Recently, he has participated/participates in EU projects on parallel programming, such as TEXT, INTERTWinE, and energy efficiency such as EXA2GREEN and OPRECOMP. His current research interests include parallel programming,

linear algebra, energy consumption, transprecision computing and bioinformatics, and advanced architectures and hardware accelerators. He has also been a member of the program committee for around 100 international conferences. In 2008, he was a recipient of an NVIDIA Professor

Partnership Award for his contributions to the acceleration of dense linear algebra kernels on graphics processors, and was also a recipient of two awards from NASA for his contributions to fault-tolerant dense linear algebra libraries for space vehicles.