Characterizing Dataset Dependence for Sparse Matrix-Vector Multiplication on GPUs

Naser Sedaghati Arash Ashari
Louis-Noël Pouchet Srinivasan Parthasarathy P. Sadayappan
Ohio State University
{sedaghat,ashari,pouchet,srini,saday}@cse.ohio-state.edu

Abstract

Sparse matrix-vector multiplication (SpMV) is a widely used kernel in scientific applications as well as data analytics. Many GPU implementations of SpMV have been proposed, proposing different sparse matrix representations.

However, no sparse matrix representation is consistently superior, and the best representation varies for sparse matrices with different sparsity patterns. In this paper we study four popular sparse representations implemented in the NVIDIA cuSPARSE library: CSR, ELL, COO and a hybrid ELL-COO scheme. We analyze statistical features of a dataset of 27 matrices, covering a wide spectrum of sparsity features, and attempt to correlate SpMV performance with each representation with simple aggregate metrics of the matrices. We present some insights on the correlation between matrix features and the best choice for sparse matrix representation.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling techniques

General Terms Performance

Keywords SpMV, GPU, Characterization

1. Introduction

Sparse matrix-vector multiplication is at the heart of numerous scientific methods, including iterative solvers on sparse linear systems [3]. A number of previous studies have focused on efficient implementation of the SpMV operation, on multi-core CPUs and GPUs [1, 2, 4, 6, 7, 13, 15, 18–21].

GPU implementations of SpMV is attractive because of the high internal GPU bandwidth and massive computation power available. Efforts to integrate sparse computations include NVIDIA cuSPARSE [9], which we focus on in this work. A particularly challenging aspect of optimizing SpMV is that the performance profile depends not only on the characteristics of the target platform, but also on the sparsity structure of the matrix, as we show in this paper. Despite the significant advances in performance of SpMV on GPUs, selecting the most efficient sparse representation for a given matrix remains a significant challenge.

We focus on four representations implemented in the NVIDIA cuSPARSE library [9]: CSR [5], ELLPACK (ELL) [16], COO, and a hybrid ELL-COO (HYB) [6], on an NVIDIA K40c GPU. We gathered a set of 27 sparse matrices, covering a spectrum of application domains and sparsity features. We evaluate the SpMV kernel performance of each of the four schemes implemented in cuSPARSE, for each matrix in our dataset, to determine which representation performs best. By analyzing and correlating sparsity features to the best performing representation, we seek insights on how to determine a priori the sparse representation delivering the highest SpMV kernel GFLOP/s. We make the following contributions:

- a performance evaluation of 4 SpMV representations implemented in the NVIDIA cuSPARSE library, on a high-end K40c GPU;
- a study of the correlation between sparse matrix features and the best kernel GFLOP/s for each representation;
- an empirical rule to select a priori the best representation on the studied dataset, based on the density and standard deviation of the number of non-zero elements per row.

The paper is organized as follows. Sec. 2 presents SpMV formats; Sec. 3 discusses the sparse matrix dataset we use; Sec. 4 characterizes the SpMV kernel performance for all covered cases; Sec. 5 discusses the selection of the best representation, and future work; related work is discussed in Sec. 6; we conclude in 7.

2. Background

2.1 Sparse Matrix-Vector Computations

SpMV is a Level-2 BLAS operations between sparse matrices and dense vectors, described by Equation 1.

$$y = \alpha \times op(A) \times x + \beta \times y \tag{1}$$

In our study, $\alpha = \beta = 1$, A is a two-dimensional sparse matrix, x and y are one-dimensional dense vectors, and $op()$ is the identity operation. Thus, the SpMV equation is simplified as $y = A \times x + y$. Looking at the element-wise operation, where $a_{i,j} \in A$, we have:

PPAA'15, February 8, 2015, San Francisco, CA, USA.
Copyright is held by the owner/author(s).
ACM /15/02.
http://dx.doi.org/10.1145/

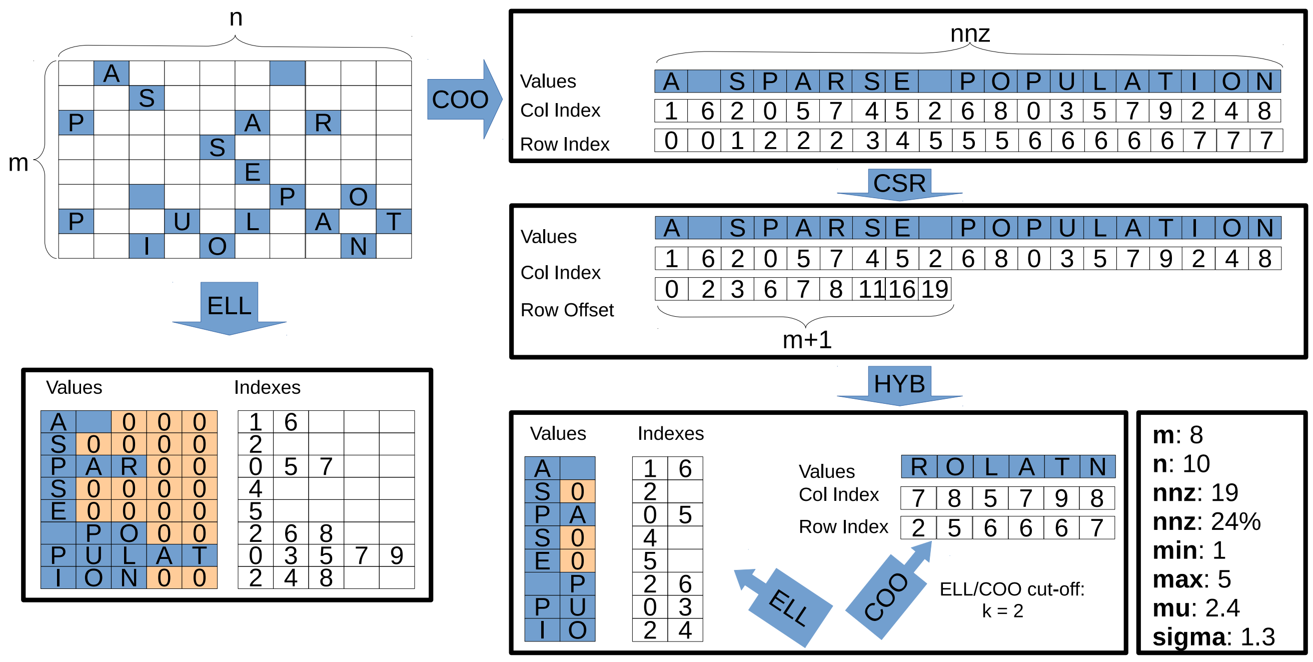


Figure 1. Sparse storage representations for matrix A in COO, CSR, ELL and HYB.

$$\forall a_{i,j} \neq 0 : y_i = a_{i,j} \times x_j + y_i \tag{2}$$

The SpMV algorithm by nature has low arithmetic intensity (i.e. AI which is defined as total flops per total DRAM bytes moved). It is typically bandwidth-bound, and due to the excellent GPU bandwidth, it has been a good candidate for GPGPU implementations [10]. For an $n \times n$ sparse matrix with total of nnz non-zero values, the algorithm performs $2 \times nnz$ flops for which a total of $nnz + 2 \times n$ words are moved. When $nnz >> n$ the AI moves towards $\frac{1}{2}$ as shown by Equation 3.

$$AI = \frac{flops}{words} = \frac{2}{4} \times \frac{nnz}{nnz + 2 \times n} \leq \frac{1}{2} \tag{3}$$

Sparsity in the matrix A leads to irregularity (and thus lack of locality) when accessing $a_{i,j}$ elements. Thus, even with optimal reuse for x and y elements, it is the accesses to matrix A (and thus the low arithmetic intensity) of SpMV that will significantly impact final execution time of a kernel.

2.2 Sparse Matrix Representations

To improve space utilization as well as temporal locality, many storage representations for the sparse matrix A have been proposed. The cost of transforming the input matrix into a more complex representation introduces a ”pre-processing” phase which is normally amortized for iterative SpMV operations (e.g. iterative linear solvers). In this work, we do not take into account this pre-processing time and focus only on the performance of the SpMV operation. However such pre-processing may become unaffordable if the input matrix changes frequently during the overall computation. Some of the well-established representations (for iterative and non-iterative SpMV) are implemented and maintained in the NVIDIA cuSPARSE library [9] which we explain in further details next.

2.2.1 Coordinate Representation (COO)

COO is most natural way of storing a sparse matrix by using three dense vectors: one to store the non-zero values, and two auxiliary vectors for storing column and row indexes of every non-zero elements. Figure 1 demonstrates how a given matrix A can be transformed into COO representation. For a given matrix with nnz non-zero values, the total memory space required for COO is $3 \times nnz$ (as listed in Table 1).

2.2.2 Compressed Sparse Row Representation (CSR)

Depending on the sparsity of the A matrix, it is possible to have very few rows with many data points (e.g. power-law matrices). For such cases, storing a row index for every element will be inefficient. Instead, one can only store the number of data points in each row. Inspired by this, CSR compresses the row index array such that non-zeros of row i as well as their column indexes can be found respectively at $values$ and col_index vectors, and in index r where $row\_offset[i] \leq r \leq row\_offset[i+1]$. As one of the simplest and most widely used representation, CSR only requires $2 \times nnz + m + 1$ of sparse to store values and row/col indexes (as shown in Figure 1). CSR-Vector [5, 6] is the currently available implementation of CSR in the cuSPARSE library [9].

2.2.3 ELLPACK Representation (ELL)

For the sparse matrices with similar nnz non-zeros per row, ELL [16] was proposed to convert the matrix into a rectangular shape by shifting the non-zero values in each row to

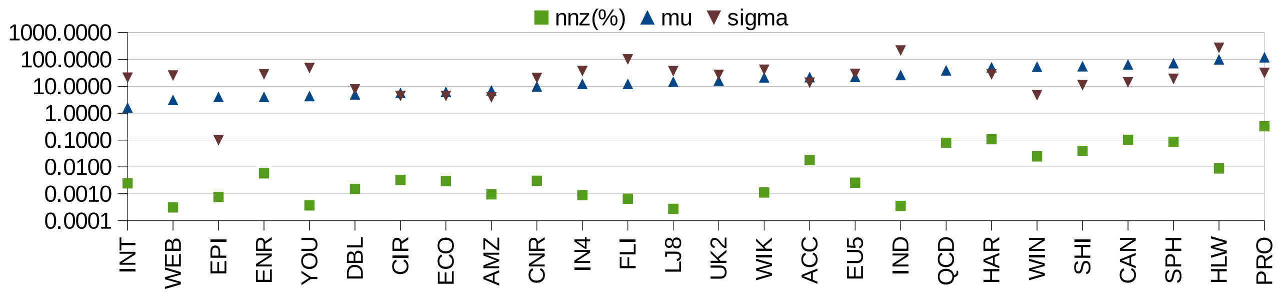


Figure 2. Matrix features: %nnz, μ and σ (sorted by increasing μ, logscale)

the left. Then, each row is zero-padded such that all the rows have the same width as the row with the largest number of non-zeros. In addition to the padded $values$ matrix, the representation requires an index matrix to store the corresponding row/col index for every non-zero element (as shown in Figure 1). As listed in Table 1, the ELL representation could be space inefficient, especially when there are very few wide rows and many very narrow ones.

2.2.4 Hybrid COO-ELL Representation (HYB)

HYB is a combination of ELL and COO representations. The idea is to partition the matrix into two parts: a dense part to be processed with ELL and a sparse part where COO could be most effective. Depending on the non-zero distribution of the matrix, a "cut-off" point is defined (i.e. k parameter) such that first k non-zero from each row are dedicated to the ELL part. If a row has less than k non-zeros, then it is zero-padded (shown in Figure 1). Note that the choice of the cut-off point is matrix-dependent and could impact the overall performance.

Representation	Space Required
COO	$3 \times nnz$
CSR	$2 \times nnz + m + 1$
ELL	$2 \times m \times max\_nnz$
HYB	$2 \times m \times k + 3 \times (nnz - X)$

Table 1. Memory space required for every storage representation, in single-precision (4-byte). Matrix A is $m \times n$ with nnz nonzero values. k is the cut-off point for ELL/COO partitioning in HYB where X nonzero values left for COO.

3. Sparse Matrices Evaluated

3.1 Dataset of Sparse Matrices

The 27 sparse matrices are selected from the Williams and LAW groups in the UFL repository [11] as well as some other groups. These matrices have been used in previous studies [1, 2, 6, 7, 20]. The matrices represent a vast range of sparsity (i.e. diagonal, blocked, banded-diagonal, etc) from different application domains. Table 2 shows key characteristics of the evaluated sparse matrices including name, size (number of rows and columns), non-zero distribution parameters (i.e. total number of non-zeros, min/max of non-zero per row, mean μ and standard deviation σ of non-zeros per row), and a snapshot that graphically demonstrates the spatial distribution of non-zero values in the sparse matrices.

3.2 Feature Analysis

We now dig into the various features for the dataset we consider. Figure 2 plots for 26 of the matrices the values of three features: μ, the arithmetic mean of non-zero entries in a row, σ the standard deviation of non-zero entries per row, and %nnz the total fraction of non-zero entries in the entire matrix. We have removed LP for easier plotting due to its extreme value for μ. The matrices are sorted by ascending value of μ, and features are shown using a log-scale.

First we observe the relatively even distribution of our dataset along the μ feature, covering well the range $3 - 100$. There are however clusters of matrices with similar μ values, as shown with matrices having almost the same μ value on this plot. On the other hand, the σ distribution is not even relative to μ: for large values of μ we have mostly $\sigma < \mu$ cases, and for small values of μ we have mostly $\sigma > \mu$ cases. It remains to be determined if these are fundamental properties of the applications modeled by those matrices, or if our dataset should be extended to cover more σ cases.

We also plot for each matrix its associated fraction of non-zero elements, %nnz. We observe a partial decorrelation between the value of μ and %nnz in our dataset: μ is not an accurate predictor of %nnz, as shown by the lack of ordering of the %nnz points. However, a trend is that for the highest values of μ (the 6 matrices on the right) a higher value of %nnz is observed: in these cases %nnz is above 0.01% that is at least one every 10,000 matrix element is non-zero, while this fraction is generally an order of magnitude smaller for smaller values of μ in our dataset.

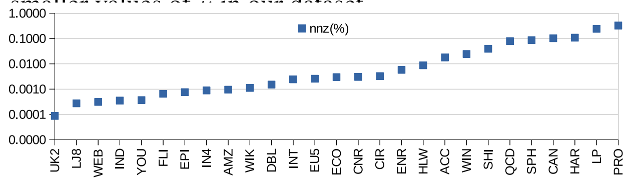


Figure 3. Matrices %nonzeros (nnz)

Figure 3 plots only the %nnz feature, sorting the matrices in increasing value of this feature. We observe also a relatively even distribution of our dataset along this feature. In later sections we will show that %nnz is actually an excellent discriminant to determine which of CSR, ELL, COO or HYB should be used to get best performance.

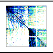
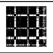
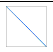
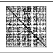
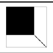
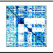
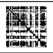
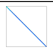
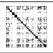
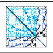
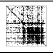
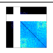
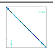
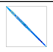
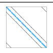
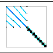
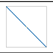
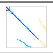
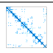
Matrix (**Abbrv.**)	Row×Col	Non-Zeros (NNZ) Distribution						Spyplot
		nnz	nnz(%)	Min	Max	μ	σ	
uk-2002 (**UK2**)	18.5M×18.5M	298.1M	0.0001	1	2450	16.1	26.7	
ljournal-2008 (**LJ8**)	5.4M×5.4M	79.0M	0.0003	1	2469	14.7	37	
webbase-1M (**WEB**)	1M×1M	3.1M	0.0003	1	4700	3.1	25.3	
indochina-2004 (**IND**)	7.4M×7.4M	194.1M	0.0004	1	6985	26.2	215.6	
youtube (**YOU**)	1.2M×1.2M	4.9M	0.0004	1	28644	4.3	48.3	N/A
flickr (**FLI**)	1.9M×1.9M	22.6M	0.0007	1	26185	12.2	101.1	
mc2depi (**EPI**)	526K×526K	2.1M	0.0008	2	4	4	0.1	
in-2004 (**IN4**)	1.4M×1.4M	16.9M	0.0009	1	7753	12.2	37.2	
amazon-2008 (**AMZ**)	735K×735K	5.2M	0.0010	1	10	7	3.9	
wiki (**WIK**)	1.9M×1.9M	40.0M	0.0011	1	6975	21.1	41.6	
dblp-2010 (**DBL**)	326K×326K	1.6M	0.0015	1	238	5	7.7	
internet (**INT**)	66K×66K	105K	0.0024	1	2639	1.6	21.1	
eu-2005 (**EU5**)	86K×86K	19.2M	0.0026	1	6985	22.3	29.3	
mac_econ_fwd500 (**ECO**)	207K×207K	1.2M	0.0030	1	44	6.2	4.4	
cnr-2000 (**CNR**)	326K×326K	3.2M	0.0030	1	2716	9.9	20.5	
scircuit (**CIR**)	171K×171K	959K	0.0033	1	353	5.6	4.4	
enron (**ENR**)	69K×69K	276K	0.0058	1	1392	4	28.3	
hollywood-2009 (**HLW**)	1.1M×1.1M	113.9M	0.0088	1	11468	99.9	271.7	
cop20k_A (**ACC**)	121K×121K	2.6M	0.0179	8	81	21.7	13.8	
pwtk (**WIN**)	218K×218K	11.6M	0.0245	2	180	53.4	4.7	
shipsec1 (**SHI**)	141K×141K	7.8M	0.0394	24	102	55.5	11.1	
qcd5_4 (**QCD**)	49K×49K	1.9M	0.0793	39	39	39	0	
consph (**SPH**)	83K×83K	6.0M	0.0865	1	81	72.1	19.1	
cant (**CAN**)	62K×62K	4.0M	0.1028	1	78	64.2	14.1	
rma10 (**HAR**)	46K×46K	2.3M	0.1082	4	145	50.7	27.8	
rail4284 (**LP**)	4K×1.1M	11.3M	0.2410	1	56181	2633	4209.3	
pdb1HYS (**PRO**)	36K×36K	4.3M	0.3276	18	204	119.3	31.9	

Table 2. Set of matrices used in this study (nnz: non-zero, μ mean, σ standard deviation)

4. Kernel Performance Characterization

4.1 Experimental Protocol

In this study, we use one of the recent NVIDIA Tesla GPUs from the Kepler generation, Tesla K40c, as described in Table 3. This GPU is characterized by its peak global memory bandwidth of 288 GB/s (which helps memory-bound operations such as SpMV) as well as its peak compute capability of 4.3 TFLOP/s (single-precision fused multiply-add).

The kernels are compiled by the NVIDIA C Compiler (*nvcc*) version 6.5, with the maximum compute capability supported (i.e. 3.5 for the Kepler GPUs). We have also enabled standard optimizations using $-O3$ switch.

GPU Model	Tesla K40c
Architecture	Kepler GK110B
Compute capability	3.5
#SMXs, cores per SMX	15 , 192
Warp size	32
Max threads / CTA,SMX	1024,2048
Sh-mem (KB) per CTA	48
L2 cache (KB)	1536
Total global memory (MB)	12288
Peak off-chip BW (GB/s)	288
Peak GFLOP/s (DP FMA)	1430

Table 3. GPU hosted the experiments.

We have selected four major sparse representations on GPUs available from the NVIDIA cuSPARSE library [9]. Each matrix is first read in Matrix Market representation [14] and stored in COO. We then convert COO to CSR from which every other representation used in this study has been generated. In our experiments we capture the time for transforming CSR into a target representation, and also the time consumed transferring data between CPU and GPU and the time spent executing the kernel. In this study, we focus only on the kernel performance (GFLOP/s), for each representation. The question we are eventually interested in answering is: given the sparsity features of the matrix, what is the most compute-efficient representation for SpMV usage. We focus on SpMV schemes where the sparse matrix fits in the GPU's global memory.

We report performance in terms of computation rate (as number of floating point operations per second in GFLOP/s). Each SpMV experiment was repeated 100 times and the average (arithmetic mean) is reported. Below, we first characterize the kernel performance, for each representation considered. We then compare the effectiveness of each representation on the same matrix later in Sec. 5.

4.2 Kernel Performance for CSR

Figure 4-A plots the GFLOP/s achieved by the SpMV computation for all matrices in our dataset, where matrices are sorted by increasing μ value. Figure 4-B plots the same data, but sorted by increasing σ value.

We observe that CSR achieves a performance above 10 GFLOP/s for more than 16 of the matrices, however a performance below 6 GFLOP/s is achieved for 7 of them. Although there are clear outliers, the GFLOP/s trend mainly follows the μ value: the higher the mean number of nonzeros per row, the higher the GFLOP/s achieved by CSR. This trend however is not verified for σ: there is no particular ordering of performance shown in Fig. 4-B.

A key concern about the implementation of CSR [5, 6] is the work distribution (i.e. how nonzeros are assigned to threads). In the CSR representation, a row is assigned to a warp of threads (i.e. group of 32). Thus, a balanced execution across threads would require matrix rows to have large-enough nonzeros, in order to avoid divergence between the threads in a warp. A high μ with a relatively low σ is expected to provide a balanced execution.

4.3 Kernel Performance for ELL

Figures 4-C:D plot similar data for the ELL representation. Numerous matrices are reported as 0 GFLOP/s as attempting to convert the input CSR matrices to this representation exhausted memory in our setup. This is typical for cases with at least one row with a very large number of non-zero (e.g., YOU), due to how ELL is generated.

We observe that for most of the matrices where ELL was applicable, ELL significantly outperforms CSR. This is particularly visible for high values of the μ spectrum (QCD to SPH), however not for the highest values. For lower values of μ, CSR generally outperforms ELL. Fig. 4-D provides interesting possible correlation between the ability to perform ELL conversion in our setup and/or get good performance, and the value of σ: matrices for which ELL provides good GFLOP/s are almost all for small values of σ. The ELL representation requires building rectangular matrices (from the input sparse matrix) by padding all the rows to be equivalent in size to the one with maximum nnz. This leads to waste of cycles for useless computation and and transfer (for padded zeros). It also significantly increases the storage space (i.e. when nonzero distribution in rows is non-uniform). Interestingly however, GFLOP/s do not correlate fully with the $max - \mu$ metric (which gives the average padding needed), as shown with the higher performance for WIN than for AMZ.

4.4 Kernel Performance for COO

Figures 4-E:F plot similar data for the stand-alone COO representation.

We observe a fairly consistent GFLOP/s achieved by COO, but the absolute performance remains limited: it is essentially between 6 and 12 GFLOP/s (except three lower performing matrices). There is a partial correlation between μ and the GFLOP/s, as for more than 2/3 of the matrices the higher the μ value the higher the GFLOP/s. However in our experiments COO never outperformed both CSR, ELL and HYB for any single matrix, as shown below with the HYB results.

The nonzero-to-thread mapping in COO is done such that every GPU thread works on a nonzero element. As a result, it is required to perform atomic operations in order to collect contributions of those threads working on elements from the same row. Additionally, an non-even distribution of nonzeros per row (i.e. high value of σ) may significantly decrease the performance because of unbalanced executions across threads. Solutions as segmented reduction [6, 17] have been proposed to decrease the atomic overhead, but the performance is still not completely invariant to the distribution of the nonzeros per row [6].

4.5 Kernel Performance for HYB

Figures 4-G:H plot similar data for the HYB representation, which automatically mixes ELL and COO.

We observe that the HYB representation can significantly outperform CSR, especially for the matrices where CSR was unable to achieve good performance, e.g. ENR and YOU. It is expected as HYB is a hybrid combining the best of ELL and COO. Cases where CSR fails to deliver good

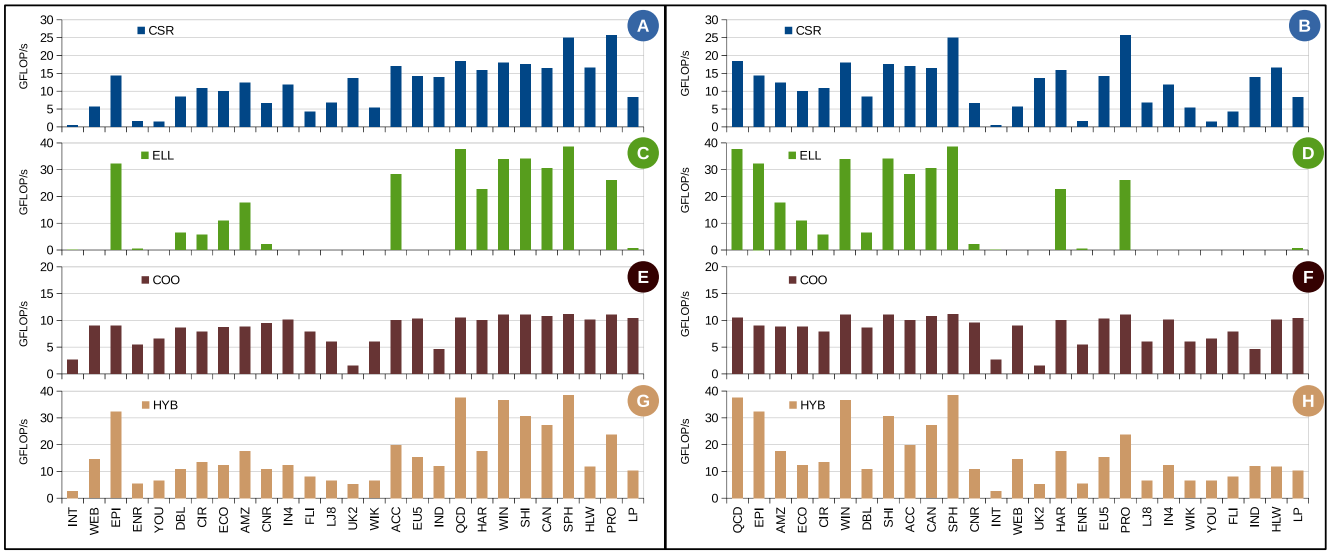


Figure 4. Performance (GFLOP/s): matrices are sorted by increasing μ (left) and by increasing $sigma$ (right).

performance are often very well handled by either ELL or COO. However, HYB is not systematically best either: it is (significantly) outperformed by CSR for 4 matrices, and outperformed by ELL in 5 other cases. While Fig. 4-G shows a somewhat general trend of higher GFLOP/s for higher value of μ, conversely Fig. 4-H shows a partial trend of lower GFLOP/s for higher values of σ.

The HYB representation automatically selects a cut-off point (i.e. k) for separation of dense (i.e. ELL) and sparse (i.e. COO) parts of a matrix. Such a parameter (which is usually found experimentally) is dependent to the target GPU as well as the input matrix. However, we have observed that for most of the matrices, using the μ value as the cut-off point achieves a better performance than what found by the library. Carefully selecting such a parameter is key to success of using the HYB for a given matrix.

4.6 Conclusions

This study allowed to conclude on several aspects. First, no representation consistently dominates the other ones. In other words, a selection of the best representation done on a per-matrix case can lead to significantly better performance. Second, there is a general (but not systematic) trend of higher GFLOP/s being achieved for higher values of μ, especially when discarding the low-end and high-end of the μ range. While this is an important finding, it however does not facilitate the actual selection of the best representation for a particular matrix, as all four representations studied have a similar trend. In the next section we discuss the various representations, and show that %nnz is a good first-order metric to determine the best representation to use.

5. Features vs. Performance

5.1 Selecting the Best Representation

It has been observed in numerous previous works that no single representation achieves the best performance on all matrices. Hence an open question is: how to select which representation performs best, given feature information about a matrix? The characterization we have performed in the previous section illustrates such differences in representation, focusing exclusively on the kernel performance (we ignore the pre-processing time needed to convert from one to another representation in this study). In the following, we correlate the %nnz feature with the best representation to choose for a majority of matrices.

5.2 The Role of the Fraction of Non-zero Entries

Figure 5 plots the kernel performance for the three useful schemes (CSR, ELL and HYB), for each matrix. We do not report COO as our previous study showed COO never outperforms all three other schemes on our dataset.

While μ had a loose correlation with the GFLOP/s achieved for all representations and σ showed no clear correlation, here we take a different approach and observe whether there is any correlation with the proportion of non-zero entries in the matrix.

A key observation is that, on our dataset, we can partition the %nnz range into three sets:

S1 where $\%nnz \in [0 - 0.00035]$: this set contains UK2 and LJ8 matrices.

S2 where $\%nnz \in [0.00036 - 0.009]$: this set contains WEB, IND, YOU, FLI, EPI, IN4, AMZ, WIK, DBL, INT, EU5, ECI, CNR, CIR, ENR and HLW matrices.

S3 where $\%nnz \in [0.009 - 0.35]$: this set contains ACC, WIN, SHI, QCD, SPH, CAN, HAR, LP and PRO.

From this partitioning into three sets, the following rules hold broadly: **use CSR for S1, use HYB for S2, use ELL for S3**. This extremely simple model successfully selects the best representation to use for all matrices except LP, WIN, HLW, and IND, i.e., it correctly selects the best representation for 23 out of the 27 evaluated matrices.

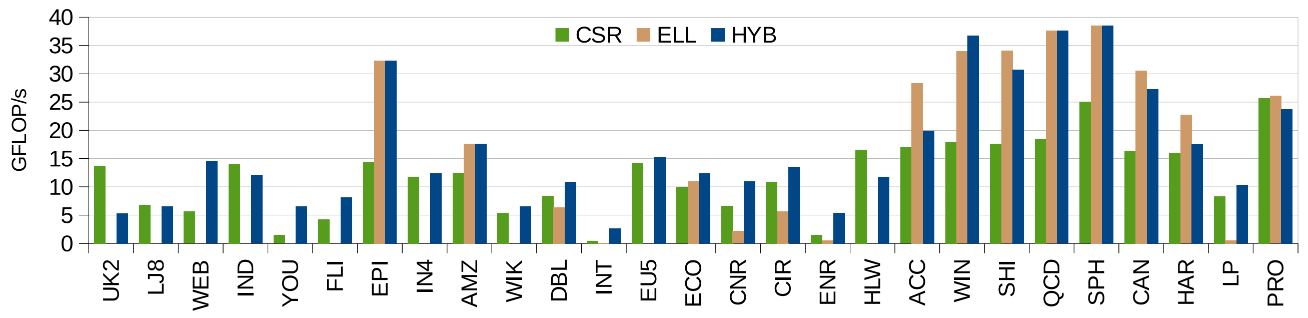


Figure 5. Comparison of performance (GFLOP/s), matrices are sorted by increasing %nnz

We can look further to the four incorrectly classified matrices. Table 4 recalls the specific features for these three matrices, which representation was predicted from the %nnz model, which representation performs best and how much performance improvement the best representation achieves over the predicted one.

Matrix	%nnz	μ	σ	Pred.	Best	Improv.
LP	0.241	2633	4209	ELL	HYB	22x
WIN	0.0245	53.4	4.7	ELL	HYB	1.05x
HLW	0.0088	99.9	271.7	HYB	CSR	1.45x
IND	0.0004	26.2	215.6	HYB	CSR	1.16x

Table 4. Three outliers to %nnz model

An interesting observation is that the three matrices with highest loss in performance have the highest σ values of the entire dataset (LP, HLW and IND), and LP has the highest μ value of the entire dataset, an order of magnitude higher to the next highest one. On the dataset we have, it appears that if $\sigma > 200$ then the %nnz rule does not work. We also remark that for IND, although CSR outperforms the predicted HYB, it is only by a marginal fraction. Actually IND is at the boundary of the range between CSR and HYB. The case of HLW is more interesting: it is at the boundary between HYB and ELL, however CSR is the better model, 1.45x faster than HYB. It shows that although CSR seems most appropriate for small values of %nnz, it may also be best for higher values of %nnz when σ is large. WIN is at the boundary between HYB and ELL, and only a marginal improvement is obtained by using HYB. However one may note that, apart from ACC, there is usually only at best a small advantage to ELL compared to HYB in our dataset.

LP on the other hand is a case of dramatic failure of our model: ELL gives the worst performance of the three, and HYB is 22x faster than ELL on this code. LP has the highest σ of all, and it is interesting to note that CSR is competitive on LP: it is only 1.3x slower than HYB.

5.3 Summary and Future Work

We have shown above how a very simple model, based on the fraction of non-zero entries in the matrix, can actually be used to determine which of the three representations (CSR, ELL and hybrid COO-ELL) should be used to achieve the best kernel GFLOP/s. This is a surprising result due to its simplicity, but we have found clear outliers. Clearly, we need to consider a more sophisticated model, which could take into account other information such as the mean and standard deviations of the number of non-zeros per row. From the study presented in this paper, several directions for future work are of interest:

1. The role of σ in the selection of the best representation needs to be further studied. In particular, we will consider increasing the matrix dataset considered with more matrices having higher values of σ, to study its correlation with the best representation.
2. The working ranges for the three representations has been manually derived, and is obviously very dependent on the dataset we have considered. As future work we will employ machine learning techniques (classifiers) to systematically compute a decision tree to select the best representation. It appears this decision will at least need to take into account %nnz and σ in an integrated way. But an open question is whether this decision tree is unique with regard to the features of the matrices, or if it also depends on the GPU architectural parameters.
3. There is also significant interest in incorporating the preprocessing cost for alternate representations in modeling the choice of best representation. For instance, assuming CSR is used as input, it may be profitable to switch to ELL only if the SpMV kernel is iteratively invoked. We will look into a model that determines how many iterations of SpMV are needed to make a change of representation worthwhile.

6. Related Work

Sparse storage representations have been studied extensively for CPUs [18, 19] as well as GPUs [6, 8, 9] and accelerators [13]. For the GPUs, the optimizations have been targeted at the features of the architecture. GPU implementation of sparse representations are significantly impacted by the work distribution strategy (i.e. the way nonzeros are mapped to the thread-blocks and threads).

In addition to the standard representations from NVIDIA cuSPARSE [9] studied in this paper (i.e. COO, ELL, CSR and HYB), other alternatives have been proposed. In general, and from the application perspective, sparse representations have moved towards more complex data structures for which a heavy pre-processing is required [2, 7, 12, 20, 21].

This is desirable when SpMV is run for several iterations (as opposed to where the input matrix changes frequently, e.g. dynamic graphs [1]). However, in addition to the data-structure and work distribution, GPU implementations may rely heavily on auto-tuning algorithms to find the best kernel configuration as well as runtime parameters [15, 20].

Our objective in this paper is not to introduce any new representation but to seek to understand how performance is related to sparsity features of the input dataset. Although this study was limited to the standard representations available in NVIDIA cuSPARSE [9], such analysis and observations can be expanded for other available representations (as future work).

7. Conclusion

Libraries implementing efficient SpMV operations on GPUs have been developed with a focus on efficient representation and exploitation of matrix sparsity. NVIDIA cuSPARSE implements several such representations, such as CSR, ELL-PACK, COO and a hybrid scheme ELL-COO.

In this work we have evaluated 27 different sparse matrices on each of the four cuSPARSE schemes, characterizing the SpMV kernel performance in each case. By reasoning on matrix features such as the mean and standard deviation of the number of non-zeros per row, and the fraction of non-zero elements in the matrix, we have observed a good correlation between %non-zeros and the best performing representation on our dataset, enabling a priori choice of the best sparse representation for iterative SpMV schemes in many of our test cases. However we found clear outliers using this simple approach, motivating the need for future work to develop more sophisticated models taking into account several matrix features, as well as to cover a wider range of matrices to gain statistical confidence in the classification heuristics.

Acknowledgments This work was supported in part by National Science Foundation awards CCF-0926127, CCF-1217353, CCF-1321147, ACI-1440749, and DMS-1418265.

References

[1] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus for graph applications. In *SC'14*, pages 781–792, 2014.

[2] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan. An efficient two-dimensional blocking mechanism for sparse matrix-vector multiplication on gpus. In *ICS'14*, 2014.

[3] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2013. http://www.mcs.anl.gov/petsc.

[4] M. M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on gpus. In *Technical report, IBM Research Report RC24704 (W0812-047)*, 2008.

[5] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[6] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[7] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, January 2010.

[8] CUSP. The nvidia library of generic parallel algorithms for sparse linear algebra and graph computations on cuda architecture gpus. https://developer.nvidia.com/cusp. URL `https://developer.nvidia.com/cusp`.

[9] cuSPARSE. The nvidia cuda sparse matrix library. https://developer.nvidia.com/cusparse. URL `https://developer.nvidia.com/cusparse`.

[10] J. Davis and E. Chung. Spmv: A memory-bound application on the gpu stuck between a rock and a hard place. Microsoft Research Technical Report MSR-TR-2012-95, Microsoft Research, 2012.

[11] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[12] J. Godwin, J. Holewinski, and P. Sadayappan. High-performance sparse matrix-vector multiplication on gpus for structured grid computations. GPGPU-5, 2012.

[13] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. *International conference on supercomputing*, pages 273–282, 2013.

[14] N. I. of Standards and Technology. The matrix market format. URL `http://math.nist.gov`.

[15] I. Reguly and M. Giles. Efficient sparse matrix-vector multiplication on cache-based gpus. In *Innovative Parallel Computing (InPar)*, pages 1–12, 2012.

[16] D. M. Y. Roger G. Grimes, David Ronald Kincaid. *ITPACK 2.0: User's Guide*. 1980. URL `http://books.google.com/books?id=h8RcNAAACAAJ`.

[17] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Graphics Hardware*, pages 97–106, 2007.

[18] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, January 2004.

[19] S. Williams, L. Oliker, R. W. Vuduc, J. Shalf, K. A. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[20] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaspmv: Yet another spmv framework on gpus. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118. ACM, 2014.

[21] X. Yang, S. Parthasarathy, and P. Sadayappan. Fast sparse matrix-vector multiplication on gpus: implications for graph mining. *Proc. VLDB Endow.*, 4(4):231–242, January 2011.