Automatic Selection of Sparse Matrix Representation on GPUs

Naser Sedaghati, Te Mu, Louis-Noël Pouchet, Srinivasan Parthasarathy, P. Sadayappan
The Ohio State University
Columbus, OH, USA
{sedaghat,mut,pouchet,srini,saday}@cse.ohio-state.edu

ABSTRACT

Sparse matrix-vector multiplication (SpMV) is a core kernel in numerous applications, ranging from physics simulation and large-scale solvers to data analytics. Many GPU implementations of SpMV have been proposed, targeting several sparse representations and aiming at maximizing overall performance. No single sparse matrix representation is uniformly superior, and the best performing representation varies for sparse matrices with different sparsity patterns.

In this paper, we study the inter-relation between GPU architecture, sparse matrix representation and the sparse dataset. We perform extensive characterization of pertinent sparsity features of around 700 sparse matrices, and their SpMV performance with a number of sparse representations implemented in the NVIDIA CUSP and cuSPARSE libraries. We then build a decision model using machine learning to automatically select the best representation to use for a given sparse matrix on a given target platform, based on the sparse matrix features. Experimental results on three GPUs demonstrate that the approach is very effective in selecting the best representation.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Efficiency

Keywords

GPU, SpMV, machine learning models

1. INTRODUCTION

Sparse Matrix-Vector Multiplication (SpMV) is a key computation at the core of many data analytics algorithms, as well as scientific computing applications [3]. Hence there has been tremendous interest in developing efficient implementations of the SpMV operation, optimized for different architectural platforms [6, 25]. 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. There is a growing interest in characterizing many graph analytics applications in terms of a dataset-algorithm-platform triangle. This is a complex relationship that is not yet well characterized even for much studied core kernels like SpMV. In this paper, we make a first attempt at understanding the dataset-algorithm dependences for SpMV on GPUs.

The optimization of SpMV for GPUs has been a much researched topic over the last few years, including auto-tuning approaches to deliver very high performance [22, 11] by tuning the block size to the sparsity features of the computation. For CPUs, the compressed sparse row (CSR) – or the dual compressed sparse column – is the dominant representation, effective across domains from which the sparse matrices arise. In contrast, for GPUs no single representation has been found to be effective across a range of sparse matrices. The multiple objectives of maximizing coalesced global memory access, minimizing thread divergence, and maximizing warp occupancy often conflict with each other. Different sparse matrix representations, such as COO, ELLPACK, HYB, CSR (these representations are elaborated later in the paper), are provided by optimized libraries like SPARSKIT [22], CUSP and cuSPARSE [10] from NVIDIA because no single representation is superior in performance across different sparse matrices. But despite all the significant advances in performance of SpMV on GPUs, it remains unclear how these results apply to different kinds of sparse matrices. The question we seek to answer in this paper is the following: *Is it feasible to develop a characterization of SpMV performance corresponding to different matrix representations, as a function of simply computable metrics of sparse matrices, and further use such a characterization to effectively predict the best representation for a given sparse matrix?*

We focus on four sparse matrix representations implemented in the cuSPARSE and CUSP libraries [10] from NVIDIA: CSR [5], ELLPACK (ELL) [21], COO, and a hybrid scheme ELL-COO (HYB) [6], as well as a variant of the HYB scheme. To study the impact of sparsity features on SpMV performance, we gathered a set of 682 sparse matrices from the UFL repository [12] covering nearly all available matrices that fit in GPU global memory. We perform an extensive analysis of the performance distribution of each considered representation on three high-end NVIDIA GPUs: Tesla K20c and K40c, and a Fermi GTX 580, demonstrating the need to choose different representations based on the matrix features, library and GPU used.

We then address the problem of automatically selecting, a priori, the best performing representation using only features of the input matrix such as the average number of nonzero entries per row (the average degree of graphs). This is achieved by developing a machine learning approach using decision tree classifiers. We perform extensive characterization of the performance of our automatically generated models, achieving on average a slowdown of no more than

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS'15, June 8–11, 2015, Newport Beach, CA, USA.
Copyright © 2015 ACM 978-1-4503-3559-1/15/06 ...$15.00.
http://dx.doi.org/10.1145/2751205.2751244.

1.05x compared to an ideal oracle approach systematically selecting the best representation for every matrix. We make the following contributions:

- an extensive performance evaluation of popular SpMV schemes implemented in NVIDIA cuSPARSE and CUSP, on three high-end NVIDIA GPUs;
- the development of a fully automated approach to select the best performing sparse representation, using only simple features of the input sparse matrices;
- an extensive evaluation of numerous machine learning models to select sparse representations, leading to portable and simple-to-translate decision trees of 30 leaf nodes achieving 95% of the maximal possible performance.

The paper is organized as follows. Sec. 2 motivates our work and presents the SpMV representations considered. Sec. 3 presents and analyzes the sparse matrix dataset we use. Sec. 4 characterizes the SpMV kernel performance for all covered cases, and Sec. 5-6 presents and evaluates our machine learning method. Related work is discussed in Sec. 7.

2. BACKGROUND AND OVERVIEW

Sparse Matrix-Vector Multiplication (SpMV) is a Level-2 BLAS operation between a sparse matrix and a dense vector ($y = A \times x + y$), described (element-wise) by Equation 1.

$$\forall a_{i,j} \neq 0 : y_i = a_{i,j} \times x_j + y_i \quad (1)$$

As a low arithmetic-intensity algorithm, SpMV is typically bandwidth-bound, and due to the excellent GPU memory bandwidth relative to CPUs, it has become a good candidate for GPU implementations [11]. Sparsity in the matrix A leads to irregularity (and thus lack of locality) when accessing the matrix elements. Thus, even with optimal reuse for the elements of vectors x and y, it is the accesses to matrix A that significantly impacts the execution time of a kernel. In particular, accessing the matrix A leads to irregular and non-coalesced memory accesses and divergence among threads in a warp. A number of efforts [5, 6, 28, 17, 27, 2] have sought to address these challenges. The NVIDIA libraries, cuSPARSE [10] and CUSP [9, 5, 6], are two of the most widely-used CUDA libraries that support different sparse representations (e.g., COO, ELL, CSR, HYB). The libraries also provide a "conversion" API to convert one representation to another.

2.1 Problem Statement

From a performance perspective, for a given matrix A with a certain sparsity structure, the representation is selected based on required space (nonzero + meta-data), the conversion time, and the kernel execution time. As shown in Table 1, the required space is a factor of the number of nonzeros (nnz), matrix dimensions ($m \times n$), and other internal (matrix-specific) parameters (e.g., cut-off point k for HYB).

The representation of the sparse matrix affects SpMV performance on GPUs and none of the representations is consistently superior as shown later in Sec. 4. Table 2 shows four sparse matrices and performance of SpMV in GFLOP/s on a single Tesla K40c GPU. It can be seen that each of the four representations is the best for one of the four matrices.

The choice of "best representation" for a given matrix depends on the sparsity structure (i.e. matrix features) as well as the GPU architecture (e.g. number of SMs, size/bandwidth of global memory, etc.). However, using representative parameters from both matrix and architecture, we can develop a model to predict the best performing representation for a given input matrix, as demonstrated in this paper.

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

Table 1: Memory space required for every representation. (matrix A is $m \times n$ with total nnz nonzeros and nnz_max maximum nonzeros per row). k is the cut-off point for ELL/COO partitioning in HYB where X non-zero values handled by ELL.

Group	Matrix	COO	ELL	CSR	HYB
Meszaros	dbir1	**9.9**	0.9	4.4	9.4
Boeing	bcsstk39	9.9	**34.3**	18.5	25.0
GHS_indef	exdata_1	9.5	4.3	**31.5**	11.3
GHS_psdef	bmwcra_1	11.5	26.7	24.8	**35.4**

Table 2: Performance difference for sparse matrices from UFL matrix repository [12] (single-precision GFLOP/s on Tesla K40c GPU)

2.2 Sparse Matrix Representations

In this work, we do not take into account pre-processing (i.e. conversion) time and focus only on the performance of the SpMV operation but the developed approach can be extended to also account for the pre-processing time, if information about the expected number of invocations of SpMV for a given matrix is known. Some of the well-established representations (for iterative and non-iterative SpMV) are implemented and maintained in the NVIDIA cuSPARSE and CUSP libraries [10, 9].

2.2.1 *Coordinate Format (COO)*

COO stores a sparse matrix using three dense vectors: the nonzero values, column indices and row indices. Figure 1 demonstrates how a given matrix A can be represented using the COO.

2.2.2 *Compressed Sparse Row Format (CSR)*

CSR [5, 6] compresses the row index array (into row_offset) such that the non-zeros of row i, as well as their column indexes, can be found respectively in the $values$ and col_index vectors, at index r, where $row_offset[i] \leq r \leq row_offset[i+1]$.

2.2.3 *ELLPACK Format (ELL)*

ELL [21] packs the matrix into a rectangular shape by shifting the nonzeros in each row to the left and zero-padding the rows so that they all occupy the same width as the one with the largest number of nonzeros. In addition to the padded $values$ matrix, the representation requires an index matrix to store the corresponding column index for every non-zero element (as shown in Figure 1).

2.2.4 *Hybrid COO-ELL Format (HYB)*

HYB uses a mix of ELL and COO representations. It partitions the input matrix into a dense part and a sparse part. The "dense" part is represented and processed using the ELL representation, while the "sparse" part uses COO.

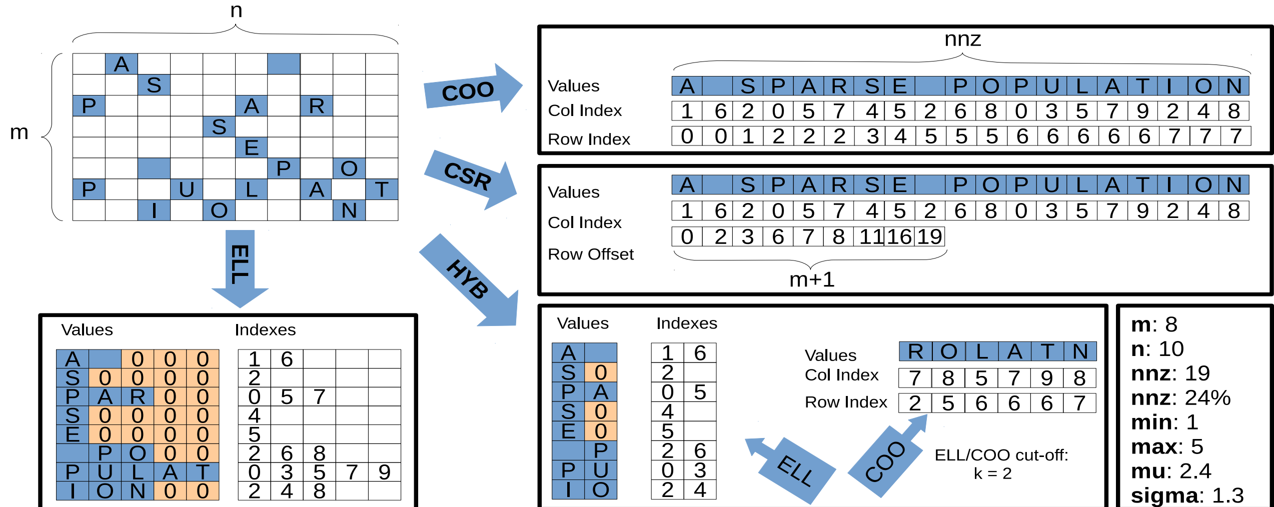

Figure 1: Sparse representations for matrix A in COO, CSR, ELL and HYB.

A parameter, k, is used to partition the first k non-zeros from each row for storiage in the ELL part, and the rest are stored in COO format. If a row has less than k non-zeros, then it is padded with zeros as shown in Figure 1. The choice of the cut-off point depends on the input matrix, and can significantly affect performance.

2.3 GPUs and Compilation

Two NVIDIA Tesla GPUs (K20c and K40c) and one Fermi GPU (GTX 580) were used for our experiments. Table 3 shows the details for each GPU. When selecting the GPUs, we have considered variety in global memory bandwidth (i.e. 192 vs 288 GB/s) as well as the compute capacity (i.e. total number of CUDA cores). The kernels were compiled using the NVIDIA C Compiler (*nvcc*) version 6.5, with the maximum compute capability supported by each device. We enabled standard optimizations using the $-O3$ switch.

GPU Model	Fermi	Tesla K20c	Tesla K40c
Chip	GTX580	GK110	GK110B
Compute Capability	2.0	3.5	3.5
Num. of SMs	16	13	15
ALUs per SM	32	192	192
Warp size	32	32	32
Threads per CTA	1024	1024	1024
Threads per SM(X)	1536	2048	2048
Sh-mem /CTA(KB)	48	48	48
L2 cache(KB)	768	1536	1536
Glob-mem (GB)	1.5	5	12
Glob-mem (GB/s)	192	208	288
Tera-FMA/s (SP/DP)	1.5/-	3.5/1.2	4.3/1.4

Table 3: GPUs hosted the experiments.

3. DATASET FEATURE ANALYSIS

3.1 Dataset Selection

We first needed a dataset of sparse matrices representative of most sparsity feature cases. For this purpose, we analyzed sparse matrices from the UFL repository [12] in Matrix Market format [18]. From more than 2650 matrices (from 177 groups of applications) in the repository, we selected 682 matrices from 114 groups. These matrices included all those used in numerous previous GPU studies [6, 8, 2, 1, 27]. While previous work using sparse matrices has usually tried to group matrices based on the domain where they arise, in this work we instead look to obtain a statistically relevant coverage among several key quantitative features describing the sparsity structure of the matrix.

Our selection mechanism applied the following constraints to set bounds on two metrics: nnz_tot, the total number of non-zero entries in the matrix, and n_rows, the number of rows:

- **C1**: the sparse matrix does not fit in the CPU LLC (8 MB for our machines). This is to focus on matrices where GPU execution is most beneficial.
- **C2**: the sparse matrix fits in the "effective" space on the global memory of the device (i.e. single-GPU execution).
- **C3**: the number of rows is large enough to guarantee minimum GPU concurrency. This is achieved by assuming that a warp works on a row; thus the minimum number of rows equals the maximum warp-level concurrency on a given device.

The size of the matrix (for **C1** and **C2**) is conservatively computed as $S = 16 \times nnz_tot$, where a nonzero in COO, double-precision, needs 16 bytes (8 bytes for data plus two 4-byte indexes). It is also assumed that up to 80% of the global memory on the device is available to hold the sparse matrix data. In **C3**, the maximum warp-level concurrency is computed by dividing the maximum number of threads per GPU (i.e. $Threads/SMX \times \#SMXs$) by the warp size.

3.2 Feature Set

Our objective is to gather relevant information about the sparsity structure of the matrix without utilizing any domain knowledge, i.e., without considering the domain where the matrix originates. For this work, we computed a set of features as shown in Table 4.

These features were (re-)computed by doing an initial traversal of the input matrix to gather these statistics. We show later in Sec. 6 that only a small subset of these features are actually needed for effective prediction of the best

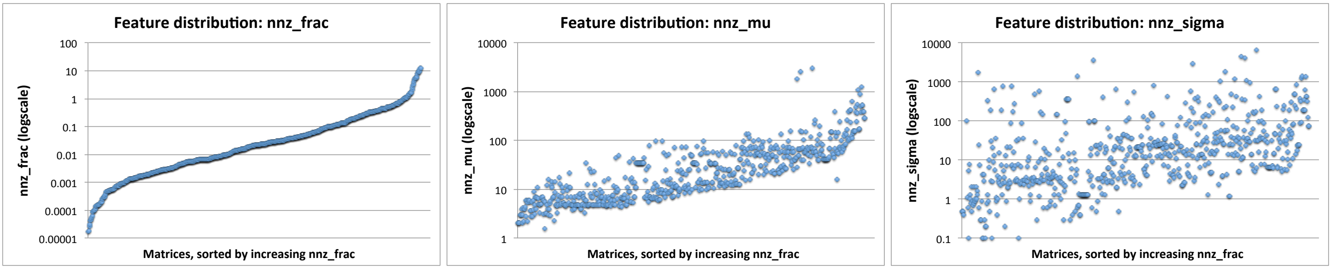

Figure 2: nnz_frac (682 matrices) **Figure 3: nnz_mu (682 matrices)** **Figure 4: nnz_sigma (682 matrices)**

Feature	Description
n_rows	Number of rows
n_cols	Number of columns
nnz_tot	Number of non-zeros
nnz_frac	Percentage of non-zeros
$nnz_\{min, max, mu, sig\}$	Min, max, average (μ) and std. deviation (σ) of non-zero per row
$nnzb_tot$	Number of non-zero blocks
$nnzb_\{min, max, mu, sig\}$	Min, max, average (μ) and std. deviation (σ) of number of non-zero blocks per row
$snzb_\{min, max, mu, sig\}$	Min, max, average (μ) and std. deviation (σ) of the size of non-zero blocks per row

Table 4: Matrix features

representation. Nevertheless, the use of more features could possibly improve the quality of the prediction. The nnz_frac feature is simply derived from the sparse data structure size and the number of matrix rows and columns, similarly for nnz_mu.

	Min	Max	Avg.	Std. Dev.
n_rows	1301	12M	377k	118k
n_cols	2339	12M	399k	126k
nnz_tot	350k	53M	4.5M	1.7M
nnz_frac	0.000018	12.48	0.31	0.92
nnz_min	1	130	8.90	11.63
nnz_max	3	2.3M	13k	8k
nnz_mu	1.60	3098	60.46	108.73
nnz_sigma	0	6505	107.38	213.77
nnzb_tot	8256	27M	2.1M	696k
nnzb_min	1	49	2.17	2.43
nnzb_max	2	462k	2203	2580
nnzb_mu	1.5	1333	21.73	28.95
nnzb_sigma	0	2720	29.21	71.69
snzb_min	1	18	1.77	0.82
snzb_max	1	309k	1101	365.8
snzb_mu	1	126.3	5.27	6.02
snzb_sigma	0	737.4	8.45	11.09

Table 5: Statistics on main features

3.3 Feature Distribution

The selected matrices represent a wide range of structure (i.e. diagonal, blocked, banded-diagonal, etc.) from different application domains. We now focus on three critical features that are group/domain-independent, and discuss their distribution and correlation below.

Feature statistics. Table 5 summarizes, across the 682 matrices, the minimum, maximum, average and standard deviation of the main features considered.

We observe that the dataset covers a wide spectrum of matrix sizes, with an average of 4.5 million non-zero entries. The average block size represents the average number of contiguous columns with non-zero entries in the rows of a matrix. As shown by the snzb_mu metric of 5.27, the average block size is quite low. These statistics provide sensible information about the UFL database concerning GPU-friendly matrices, and confirm via the standard deviation values of these features (last column) that we do cover wide ranges for the various features and that our selection of matrices is effective.

Figures 2-4 show the distribution for the three critical features considered.

Fraction of non-zeros. This feature represents the fractional sparsity of the matrix, in a manner independent of the matrix size. Figure 2 plots its value for each matrix, sorted in ascending order. We observe a near perfect coverage of this feature, but with a lower density at the extremes of the range. It may be observed that this feature is an excellent discriminant between matrices in our database, from the absence of horizontal segments. This motivates its inclusion in all feature sets for prediction of the best sparse representation.

Average number of non-zeros per row. This feature is plotted in Figure 3, using the same order along the x axis as in Figure 2, i.e., it plots nnz_mu in increasing order of nnz_frac. We also observe an excellent coverage of the feature range. We observe only a mild correlation between the two features, as seen by the overall increasing trend of nnz_mu. Nevertheless there are significant outliers to this trend, indicating that nnz_mu carries complementary information to nnz_frac.

Standard deviation of number of non-zeros per row. This feature is plotted in Figure 4 in increasing order of nnz_frac. While we also observe an excellent coverage of the feature range, there is a clear decorrelation between the two features. This is indicative of different information being carried by nnz_sigma than nnz_frac, and motivates its inclusion in all feature sets.

4. PERFORMANCE ANALYSIS

We now perform a characterization of the performance of each representation tested, on all matrices in the dataset. We conducted experiments by measuring the performance of the SpMV kernel in GFLOP/s, on three GPUs (K40c, K20c and Fermi, as described in Sec. 2). The figure reports the total kernel execution time; data transfer time between

CPU and GPU is not included. This is appropriate since we particularly target iterative SpMV schemes, where the matrix is expected to be loaded once and the kernel repeated numerous times.

Best performing representations. Table 6 reports how many matrices, out of the 682 considered, are best executed (i.e., highest GFLOP/s achieved), with a given representation.[1] For cuSPARSE, we also evaluated HYB-MU, a variation of the hybrid scheme, where the cut-off point between ELL and COO is not chosen by the library, but instead set to the nnz_mu value of the matrix. Such a user-specified control is not available with CUSP.

	COO	ELL	CSR	HYB	HYB-MU
K40c-cusparse	10	**278**	104	103	185
K40c-cusp	0	168	172	**339**	N/A
K20c-cusparse	10	**269**	95	134	174
K20c-cusp	1	194	168	**318**	N/A
Fermi-cusparse	5	200	**302**	86	87

Table 6: Fastest representation statistics

First, we observe that the hybrid scheme is the best performing one in the highest number of matrices for CUSP, while for cuSPARSE ELL dominates on the Teslas. We also observe an apparent consistency between Tesla GPUs: similar trends in the distribution of the best appears for both K20c and K40c. On the older Fermi architecture, CSR actually performs consistently best, but as shown below, the true performance differences with HYB are minimal. However, care must be taken with these numbers: we conducted a more precise analysis showing that the matrices for which CSR is best for K40c are not strictly the same as the ones for K20c, and similarly between CUSP and cuSPARSE. While ELL, COO and HYB have correlation, since HYB combines the first two, we observe that CSR remains an important representation for getting the best performance for around 20% of the matrices.

Performance differences. The previous table only shows a fraction of the information. In practice, what matters is not only which representation performs best, but also the performance difference between them. Table 7 shows the average slowdown in always using the same representation for the entire dataset, compared to using the best representation found for each matrix individually. For instance, 1.97x for COO / K40c-cusparse means that we lose about 50% of the performance by always using COO for the 682 matrices, compared to using the best representation for each of them.

	COO	ELL	CSR	HYB	HYB-MU
K40c-cusparse	1.97x	1.60x	1.54x	**1.18x**	1.18x
K40c-cusp	2.21x	1.87x	1.54x	**1.14x**	N/A
K20c-cusparse	1.88x	1.60x	1.55x	**1.17x**	1.17x
K20c-cusp	2.16x	1.90x	1.52x	**1.14x**	N/A
Fermi-cusparse	2.02x	1.71x	**1.20x**	**1.20x**	**1.20x**

Table 7: Average slowdown over the 682 matrices when a fixed representation is used instead of the individual best

The HYB representation performs best overall, and there is overall consistency across libraries and GPUs. However, it also clearly demonstrates that using the popular CSR can lead to a significant average slowdown on Tesla GPUs.

Large slowdowns. While it seems from the above that HYB is a good overall representation in terms of performance, these average numbers mask very significant slowdowns on some matrices, due to the large number of matrices in the dataset. Table 8 reports the number of matrices (out of 682) for which use of a fixed representation across the entire dataset results in a slowdown greater than 2x over the best representation.

	COO	ELL	CSR	HYB	HYB-MU
K40c-cusparse	360	243	166	47	**37**
K40c-cusp	422	322	248	**50**	N/A
K20c-cusparse	339	245	134	45	**33**
K20c-cusp	410	326	231	**42**	N/A
Fermi-cusparse	404	268	68	**20**	23

Table 8: Number of > 2x slowdown cases

This table carries critical information and dismisses the one-size-fits-all strategy that might be suggested from the previous data. If HYB is always used, between 20 and 50 matrices would achieve less than 50% of the possible performance. Slowdowns of up to 10x can actually arise by always using HYB. The numbers are even more staggering for the other representations, where such loss of performance can occur for half of the matrices. *This data clearly motivates the need for an effective approach to select the best representation for each matrix* in order to achieve consistently good performance. This metric of the number of cases with 2x slowdown is a particular point of focus in the quality of our machine learning approach described in Sec. 5, where we succeed in reducing this number to no more than 6 out of 682 matrices.

Impact of ELL and CSR. Table 9 focuses on the K40c, considering the 2x-slowdown cases when using HYB, and reports which representation actually performing best, and the actual average slowdown for each of these. For example, out of the 47 cases where HYB has a 2x slowdown, for 18 of them CSR was the best representation, and a speedup of 2.36x on average can be achieved by using CSR instead of HYB for these 18 cases.

	COO	ELL	CSR	HYB-MU
(CS) > 2x slowdown	0	29	18	0
(CS) Avg. slowdown	N/A	2.36x	2.54x	N/A
(CU) > 2x slowdown	0	22	28	N/A
(CU) Avg. slowdown	N/A	2.28x	2.75x	N/A

Table 9: K40c using HYB

This shows that while the HYB-MU case does not lead to any improvement greater than 2x over HYB, both ELL and CSR are equally critical to address the cases with large slowdowns. In other words, to ensure good overall performance and avoid cases with significant loss of performance, a model must select at least between HYB, CSR and ELL.

SpMV performance breakdown. Finally Figures 5-9 plots the matrix distribution of performance (GFLOPS/s) achieved by the SpMV kernel, for all configurations. The range of GFLOP/s achieved for a configuration is split into 10 buckets, and the number of matrices achieving no more than the GFLOP/s label on the x axis (and no less than the value for

[1] The latest version of CUSP did not work on the older Fermi GPU; we only report cuSPARSE results for this GPU.

the previous bucket) is reported. For example, 394 matrices achieve between 8.3 GFLOP/s and 12.4 GFLOP/s using COO with K40c/cuSPARSE in Fig. 5.

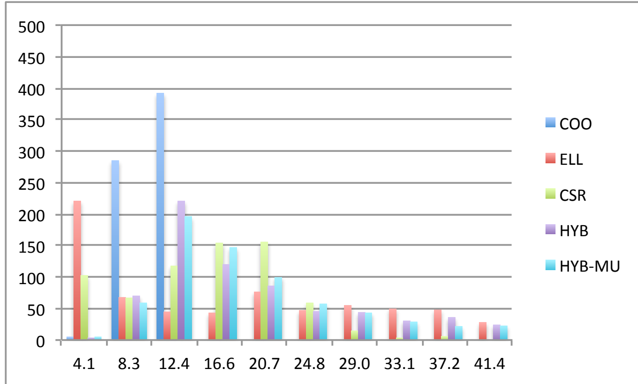

Figure 5: K40c, cuSPARSE

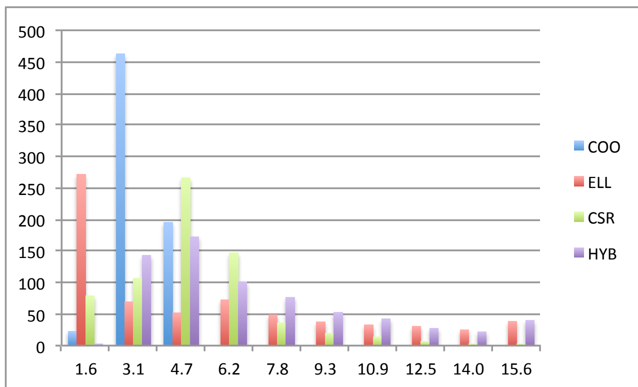

Figure 6: K40c, CUSP

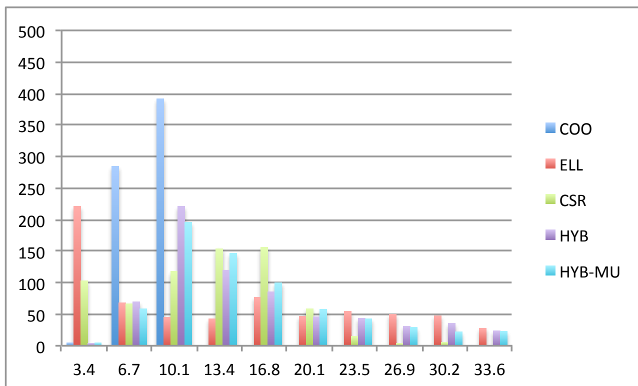

Figure 7: K20c, cuSPARSE

We observe that CSR is almost never in any of the three right-most buckets, and COO is mainly represented at the low end of the spectrum, while both HYB and ELL are spread across the entire spectrum. In other words, the GPU computing power may be better harnessed for matrices where HYB and ELL perform best. We remark that for the cases where ELL failed to convert from MatrixMarket format on

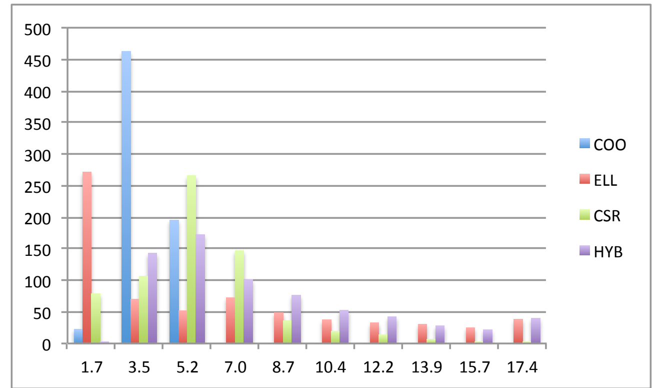

Figure 8: K20c, CUSP

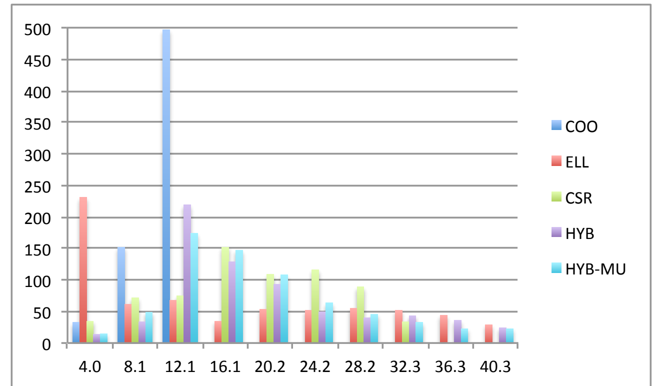

Figure 9: Fermi, cuSPARSE

the GPU, we report its GFLOP/s as 0 which artificially increases its representation in the first bucket. Such cases of failure of conversion occur typically because of very large nnz_max values, where ELL is not appropriate to use. While we could simply have removed ELL for these matrices, for consistency of the protocol, we chose to simply set their GFLOP/s to 0 (and their speedup compared to other representations also to zero).

We conducted additional study to observe if there is any correlation between the fraction of non-zero in the matrix and the GF/s achieved by each representation. We separated the set of matrices in 10 classes, by increasing range of the value of nnz_frac, and computed the average GF/s of each representation in this class. We observed that while for COO and to a more limited extent CSR the higher the nnz_frac the higher the GF/s on average, this trend is reverted for ELL and the hybrid schemes: the lower the nnz_frac the better the average GF/s. Note that there are numerous outliers to these trends: while previous results showed that, on a restricted dataset, nnz_frac was a possible predictor of the best representation [23] the extensive study we conduct in this paper showed the requirement to consider additional features to properly determine the best representation in general.

5. MACHINE LEARNING MODELING

We now elaborate on the machine learning approach we use to automatically predict the best representation, based only on efficiently computed features of the input matrix.

5.1 Problem Learned

Based on our findings presented above, we concluded on the need to develop an approach which would predict, at the very least, which of CSR, ELL or HYB is the best representation for a matrix. While this is a simpler problem instance (3 categories), in order to achieve the truly best overall performance we keep each possible representation for prediction. We build a predictor of the form:

$$Predictor(\vec{x}) = c$$

where $c \in$ { COO, ELL, CSR, HYB, HYB-MU } for cuSPARSE, and we remove HYB-MU for CUSP. $\vec{x}$ is the vector of features we consider as input; these are metrics computed for each dataset.

Learning Algorithms. A classification tree is a form of decision tree where the predicted value is a category. It takes the form of a series of conditions on the input feature values, leading to a leaf containing the predicted category when the conditions are met. Figure 10 shows an example of an excerpt of a tree built in our experiments, the number in comment (e.g., `#(22)`) shows the number of matrices correctly predicted by matching the input features against each conditional, until a leaf is reached. For example a matrix with nnz_sigma = 2, nnz_frac = 0.00005 is predicted as HYB here.

```
nnz_sigma < 14.9
|  nnz_mu < 5.3
|  |  nnz_sigma < 2.05
|  |  |  nnz_frac < 1.0E-4
|  |  |  |  nnz_frac < 3.0E-5: HYB-MU #(3)
|  |  |  |  nnz_frac >= 3.0E-5: HYB   #(4)
|  |  |  nnz_frac >= 1.0E-4: ELL      #(22)
```

Figure 10: Classification tree example

Tree models have numerous advantages over other algorithms, including (1) the ability for a human to read and understand them, favoring knowledge transfer; (2) their excellent evaluation speed and quick training speed, enabling use with large datasets; (3) their widespread availability in machine learning libraries and ease of implementation.

In this work we conducted an extensive study using the Weka library version 3.6.12 [19] on most of the available decision tree algorithms implemented. This included RandomTree, J48, BFTree, and SimpleCart. We determined that BFTree and SimpleCart outperformed the other algorithms and therefore focused particularly on those two. BFTree [14] is a best-first decision tree classifier, while SimpleCart [7] implements a hierarchical optimal discriminant analysis using minimal cost-complexity pruning [19]. Both these algorithms can be parameterized to control the size of the tree generated, trading off generalization and prediction quality.

5.2 Training and Testing

Our experimental protocol is as follows. First the GFLOP/s for all representations and all matrices is computed, and each matrix is given a class attribute corresponding to the actual best performing representation (c above). Then, the entire set of 682 matrices is split into two sets, following the 80%-20% rule. The training of the model is done on the largest bucket containing 80% of the dataset, while the evaluation is done on the remaining 20%. This process is repeated 5 times, each with another 80-20 split, so that each matrix is used for testing exactly once. We use a stratified cross-validation with randomization to create each case, using the available Weka tools. This seeks to approximately retain the original class (c) distribution in both sets.

In our experiments the training was completed in less than 2 seconds on a standard laptop, and the testing in less than 0.1s, for one particular machine learning algorithm instance.

6. SELECTING A REPRESENTATION

We now present the evaluation of BFTree and SimpleCart on the prediction problem.

6.1 Experimental Protocol

Feature sets. We evaluated several feature sets, described in Table 13. The Simple feature set is meant to capture features that are extremely easy to compute, only nnz_sigma needs a scan of the matrix or sampling to be obtained. The Advanced1 and Advanced2 sets on the other hand consider statistics on the number and size of blocks of non-zeros, and typically requires a scan of the matrix, or a good knowledge of the regularity in the sparsity pattern.

Name	Description
Simple	nnz_frac, nnz_mu, nnz_sigma
Advanced1	nnz_frac, nnz_mu, nnz_sigma, nnzb_mu, nnzb_sigma, snzb_mu, snzb_sigma
Advanced2	nnz_frac, nnz_max, nnz_mu, nnz_sigma, nnzb_mu, nnzb_sigma, snzb_mu, snzb_sigma

Table 13: Feature sets evaluated

Algorithm parameters. Both algorithms take two parameters as argument for the training stage, determining (1) minNum the minimal number of instances at the terminal node; and (2) numFold the number of folds used in the pruning. We have explored the Cartesian product of combinations generated with the values $minNum = \{2, 3, 4, 5\} \times numFold = \{5, 10, 15\}$.

Invalidating the random approach. We have evaluated a classical random classifier to ensure the problem cannot be addressed using a simple random selection of the best representation. As expected this approach failed, giving an average slowdown compared to the best representation of 1.7x for cuSPARSE and 4.7x for CUSP — significantly worse than always using HYB.

6.2 Evaluation of Specialized Models

We first analyze the results of a first approach of training a different model for each GPU and each library independently, before discussing the use of a single model for all cases.

Tables 10-12 presents statistics about the best performing models founds, after exploring all algorithm parameters. The criterion retained for optimality was the minimization of the number of matrices for which a 2x slowdown or more was experienced, compared to the best representation. A total of 12 configurations was tested per entry in these tables, for a grand total of 144 models trained and evaluated per GPU, from which we report the 12 best. The parameters used to train the best model are shown, along with the percentage of testing matrices that were mispredicted from the testing set, the number of matrices achieving a 2x slowdown or more compared to the best performing matrices, and the average slowdown compared to the best per-matrix representation. These tables should be put in perspective with Tables 7-8.

			minNum	numFold	misprediction	> 2x slowdown	Avg. slowdown
cuSPARSE	**BFTree**	Simple	**2**	**5**	34.8%	**6**	**1.07x**
		Advanced1	2	10	34.2%	7	1.08x
		Advanced2	**2**	**5**	20.4%	**3**	**1.04x**
	SimpleCart	Simple	2	10	35.2%	6	1.08x
		Advanced1	2	10	18.8%	3	1.04x
		Advanced2	2	10	19%	3	1.04x
CUSP	BFTree	Simple	2	10	24%	4	1.06x
		Advanced1	2	10	18.8%	3	1.04x
		Advanced2	2	5	16.2%	1	1.02x
	SimpleCart	Simple	**5**	**15**	25%	**3**	**1.05x**
		Advanced1	2	10	19%	3	1.04x
		Advanced2	**2**	**5**	16.2%	**1**	**1.02x**

Table 10: Results for K40c, training one model per library

			minNum	numFold	misprediction	> 2x slowdown	Avg. slowdown
cuSPARSE	**BFTree**	Simple	**2**	**15**	33.6%	**6**	1.08x
		Advanced1	2	15	30.2%	5	1.07x
		Advanced2	**2**	**5**	26.2%	**1**	**1.03x**
	SimpleCart	Simple	2	5	29.8%	7	1.08x
		Advanced1	3	15	24.2%	2	1.03x
		Advanced2	3	5	24%	2	1.03x
CUSP	**BFTree**	Simple	**2**	**5**	31.4%	**3**	**1.06x**
		Advanced1	2	10	24%	4	1.05x
		Advanced2	**2**	**10**	21.2%	**1**	**1.02x**
	SimpleCart	Simple	2	10	29.6%	4	1.05x
		Advanced1	2	10	24.4%	4	1.05x
		Advanced2	2	15	23.6%	2	1.03x

Table 11: Results for K20c, training one model per library

			minNum	numFold	misprediction	> 2x slowdown	Avg. slowdown
cuSPARSE	BFTree	Simple	3	5	35.4%	6	1.08x
		Advanced1	2	10	33.5%	5	1.07x
		Advanced2	2	5	26.6%	2	1.03x
	SimpleCart	Simple	**4**	**5**	35.1%	**5**	**1.08x**
		Advanced1	2	5	27.3%	2	1.03x
		Advanced2	**2**	**15**	26.6%	**2**	**1.03x**

Table 12: Results for Fermi

General observations. First and foremost, we observe the very strong gain of our approach compared to using a single representation (HYB), both in terms of average slowdown and especially in terms of number of matrices for which we use an ineffective representation. Even limiting to a simple feature set made of nnz_frac, nnz_mu and nnz_sigma, we can achieve on average 92% or more of the maximal performance and limit the number of ineffective cases to 6 or less, for both libraries and GPUs. Considering a more complicated feature set, these numbers improve to 96% and 3.

Another interesting aspect is the somewhat marginal gain that increasing the feature set provides: considering features on the block size and distribution result in a decrease of the number of ineffective cases only from 6 to 3 or less. So it appears that simple features are overall good enough for a predictor, which enhances the simplicity of the approach: such features can be computed from the size of the data structures and provided by a domain expert.

Regarding mispredictions, we observe an overall high number of matrices being mispredicted, ranging from 16% to 38% of the testing set in the worst case. We initially used this metric as the criterion to select the best parameter configuration. However, reducing this metric only led to larger trees, with no improvement in average slowdown. The reason for the lack of apparent correlation between the misprediction rate and the average slowdown is found in the performance distribution, and the overall comparable performance of different representations in numerous cases. For these, our models may not predict the absolute best representation, but at least predict a representation achieving very good performance, close to the best of the selected representations. Allowing for higher misprediction rate while preserving the average slowdown and number of ineffective cases enabled the building of simpler and smaller trees, improving generalization and knowledge transfer.

Finally, we observe that while BFTree performs overall best for the K20c GPU, SimpleCart slightly outperforms BFTree with the Fermi and K40c for CUSP. Nevertheless, with differences being so marginal, we can use either model. A more challenging observation relates to the algorithm parameters. Clearly, different parameters are needed across GPUs, and even across libraries, to achieve the best performance.

Single parameter set. Table 14 shows performance statistics for use of a single parameter configuration and classification algorithm. As no algorithm/parameter setup performs best across all configurations, we chose the one that minimizes the total number of ineffective configurations.

	> 2x slowdown	Avg. slowdown
K40c/cuSPARSE	6	1.07x
K40c/CUSP	5	1.06x
K20c/cuSPARSE	8	1.09x
K20c/CUSP	6	1.05x
Fermi/cuSPARSE	6	1.08x

Table 14: BFTree minNum=2, numFold=5, Simple **features**

These results show a slight degradation of the results when parameter selection is avoided, making it a good candidate for single-shot training (i.e., no auto-tuning on the parameters). We however remark that, as the training time is a matter of a few seconds per configuration, such training auto-tuning on the proposed range of parameter remains feasible at library installation time. Nevertheless, running all SpMV cases on all representations to collect the classified data for this supervised learning will remain a task-consuming task.

A trained tree for K40c. We conclude our presentation by showing the tree produced for K40c / cuSPARSE in Fig. 11. This very simple tree was trained on the entire dataset and as such can be embedded as-is as a predictor in the cuSPARSE library, to provide the user guidance on the possible fastest representation. It is particularly interesting to note how 102 (122-18) ELL cases are correctly predicted using only conditions on nnz_mu and nnz_sigma, and 34 more considering nnz_frac.

```
nnz_sigma < 14.9
|  nnz_mu < 5.3
|  |  nnz_sigma < 2.05
|  |  |  nnz_frac < 1.0E-4: HYB-MU #(5.0/5.0)
|  |  |  nnz_frac >= 1.0E-4: ELL #(22.0/4.0)
|  |  nnz_sigma >= 2.05
|  |  |  nnz_sigma < 2.75: HYB-MU #(7.0/6.0)
|  |  |  nnz_sigma >= 2.75: HYB-MU #(27.0/15.0)
|  nnz_mu >= 5.3
|  |  nnz_sigma < 5.95
|  |  |  nnz_mu < 34.45: ELL #(122.0/18.0)
|  |  |  nnz_mu >= 34.45: HYB #(19.0/13.0)
|  |  nnz_sigma >= 5.95
|  |  |  nnz_mu < 26.9
|  |  |  |  nnz_frac < 0.01521
|  |  |  |  |  nnz_frac < 0.00319
|  |  |  |  |  |  nnz_sigma < 6.95: HYB #(4.0/1.0)
|  |  |  |  |  |  nnz_sigma >= 6.95: HYB-MU #(5.0/5.0)
|  |  |  |  |  nnz_frac >= 0.00319: HYB-MU #(8.0/2.0)
|  |  |  |  nnz_frac >= 0.01521: ELL #(6.0/7.0)
|  |  |  nnz_mu >= 26.9
|  |  |  |  nnz_frac < 0.63702: ELL #(45.0/11.0)
|  |  |  |  nnz_frac >= 0.63702: CSR #(4.0/1.0)
nnz_sigma >= 14.9
|  nnz_frac < 0.0411
|  |  nnz_mu < 79.4
|  |  |  nnz_mu < 20.95
|  |  |  |  nnz_frac < 0.00144: HYB-MU #(14.0/1.0)
|  |  |  |  nnz_frac >= 0.00144
|  |  |  |  |  nnz_sigma < 97.3
|  |  |  |  |  |  nnz_mu < 13.7
|  |  |  |  |  |  |  nnz_mu < 8.15
|  |  |  |  |  |  |  |  nnz_sigma < 33.6: HYB #(9.0/1.0)
|  |  |  |  |  |  |  |  nnz_sigma >= 33.6: HYB-MU #(3.0/2.0)
|  |  |  |  |  |  |  nnz_mu >= 8.15: HYB-MU #(23.0/11.0)
|  |  |  |  |  |  nnz_mu >= 13.7: HYB #(9.0/1.0)
|  |  |  |  |  nnz_sigma >= 97.3: HYB-MU #(20.0/7.0)
|  |  |  nnz_mu >= 20.95: HYB-MU #(27.0/10.0)
|  |  nnz_mu >= 79.4: HYB #(5.0/0.0)
|  nnz_frac >= 0.0411
|  |  nnz_frac < 0.69183
|  |  |  nnz_sigma < 102.8
|  |  |  |  nnz_sigma < 31.45
|  |  |  |  |  nnz_mu < 21.4: HYB-MU #(3.0/3.0)
|  |  |  |  |  nnz_mu >= 21.4
|  |  |  |  |  |  nnz_sigma < 24.4
|  |  |  |  |  |  |  nnz_mu < 52.2: HYB #(4.0/5.0)
|  |  |  |  |  |  |  nnz_mu >= 52.2: ELL #(4.0/2.0)
|  |  |  |  |  |  nnz_sigma >= 24.4: ELL #(15.0/2.0)
|  |  |  |  nnz_sigma >= 31.45: CSR #(22.0/29.0)
|  |  |  nnz_sigma >= 102.8
|  |  |  |  nnz_sigma < 209.2: CSR #(9.0/5.0)
|  |  |  |  nnz_sigma >= 209.2
|  |  |  |  |  nnz_sigma < 1519.35
|  |  |  |  |  |  nnz_frac < 0.06378: HYB #(3.0/2.0)
|  |  |  |  |  |  nnz_frac >= 0.06378: HYB-MU #(12.0/9.0)
|  |  |  |  |  nnz_sigma >= 1519.35: COO #(4.0/1.0)
|  |  nnz_frac >= 0.69183: CSR #(35.0/8.0)

Size of the Tree: 59
Number of Leaf Nodes: 30
```

Figure 11: Tree trained on all input data, K40c

7. RELATED WORK

There is an extensive amount of work in the literature on customizing sparse matrix representations and optimizing SpMV on different platforms. This paper follows our previous work [23] and raises the question of how to choose between these representations in an automated, portable and systematic way.

In the CPU domain, Vuduc [25] studied SpMV on single-core CPUs and presented an automated system for generating efficient implementations of SpMV on these platforms. Williams et al. [26] moved toward multi-core platforms with the implementation of parallel SpMV kernels. SpMV kernels have been studied and optimized for GPUs as well. Bell and Garland implemented several sparse matrix representations in CUDA [6] and proposed HYB (hybrid of ELL and COO), that generally adapts well to a broad class of unstructured matrices.

Baskaran and Bordawekar [4] optimized CSR so that it performs about the same (modestly faster with more memory cost) as CSR-Vector [6] (in which a row is assigned to each warp, which then performs segmented reduction). Choi et al. [8] introduced blocking to CSR and ELL (BCSR and BELLPACK). These representations outperform HYB for matrices with dense block substructure but on average their auto-tuned version is not as good as HYB for general unstructured and sparse matrices.

Liu et al. [17] proposed ELLPACK Sparse Block (ESB) for the Intel Xeon Phi Co-processor, and show that on average ESB outperforms cuSPARSE on NVIDIA K20X.

Yang et al. [28] developed a representation suited for matrices with power-low characteristics, by combining Transposed Jagged Diagonal Storage (TJDS) [13] with COO and blocking. Ashari et al. [1] presented a CSR-based approach (ACSR) crafted for power-law matrices, using binning and dynamic parallelism on Kepler GPUs. Liu et. al. [16] recently proposed CSR5 (a CSR-based representation) for SpMV on different target platforms (CPU, GPU and Xeon Phi), which is insensitive to the sparsity structure of the input sparse matrix. Reguly and Giles [20] also presented a CSR-based solution with an auto-tuning algorithm to select the number of rows for a thread to work on based on available resources on the device. Additionally, there have been developments towards having a runtime to select the best-performing representation for a given matrix (e.g. Cocktail Format [24]).

As briefly summarized above, sparse storage representations have been studied extensively for GPUs. Proposed optimizations have followed changes in the GPU architecture. GPU implementation of such representations are significantly impacted by the way nonzeros are distributed across the thread-blocks and threads [6, 15, 5, 8, 27, 28, 1].

While we restricted ourselves to a set of standard representations available in NVIDIA cuSPARSE [10] and CUSP [9], our approach does not depend on the representation used and we believe it can be effectively extended to cover additional sparse matrix representations.

8. CONCLUSION

Implementing an effective SpMV algorithm on GPUs has proven to be challenging task which is only feasible in the presence of efficient sparse representation and exploitation of the matrix sparsity. NVIDIA cuSPARSE (among others) implements several such representations (e.g. CSR, ELLPACK, COO and a hybrid scheme ELL-COO). But the choice of which representation is the best for a given matrix remains mostly unknown.

In this work we have addressed the problem of automatically selecting the best sparse representation for effective SpMV execution on GPU, by using input-dependent features on the sparse matrix to be operated on. Our approach allows to achieve within 95% on average of the best performance available. We have extensively characterized the feature distribution of the popular UFL repository, selecting 682 matrices suitable for GPU acceleration. We have analyzed the performance distribution of two popular Sparse toolkits from NVIDIA, cuSPARSE and CUSP, on three high-end GPUs, GTX 580, K20c and K40c. We showed that no representation performs best across this set, and that a one-representation-fits-all approach, while providing encouraging performance, can actually result in numerous cases of very poor performance. We proposed a machine learning approach to automatically predict the best sparse representation, using classification trees. Our experiments show that, using only basic features, our approach shows no more than 6 out of 682 matrices to have poor performance using our predictor. More complex features can lead to a 98% efficiency, with only 1 matrix with poor performance using our predictor.

Acknowledgments. We thank NVIDIA Corporation for donating the K40c GPU. This work was supported in part by National Science Foundation awards CCF-0926127, CCF-1240651, CCF-1321147, CCF-1418265 and ACI-1404995.

9. 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 *SC'09*, 2009.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth International Group, Belmont, California, 1984.

[8] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *PPoPP'10*, January 2010.

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

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

[11] 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.

[12] 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.

[13] A. Ekambaram and E. Montagne. An alternative compressed storage format for sparse matrices. In *ISCIS*, pages 196–203, 2003.

[14] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression : A statistical view of boosting. *Annals of statistics*, 28(2):337–407, 2000.

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

[16] W. Liu and B. Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *ICS'15*. ACM, 2015.

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

[18] N. I. of Standards and Technology. The matrix market format.

[19] U. of Waikato. Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/.

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

[21] D. M. Y. Roger G. Grimes, David Ronald Kincaid. *ITPACK 2.0: User's Guide*. 1980.

[22] Y. Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2. 1994.

[23] N. Sedaghati, A. Ashari, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Characterizing dataset dependence for sparse matrix-vector multiplication on gpus. In *PPAA'15*, pages 17–24. ACM, 2015.

[24] B.-Y. Su and K. Keutzer. clspmv: A cross-platform opencl spmv framework on gpus. In *ICS'12*, pages 353–364, 2012.

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

[26] 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.

[27] S. Yan, C. Li, Y. Zhang, and H. Zhou. yaspmv: Yet another spmv framework on gpus. In *PLDI'10*, pages 107–118. ACM, 2014.

[28] 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.