Optimizing Sparse-Matrix Sparse-Vector Multiplication for GPUs

Changwan Hong, Aravind Sukumaran Rajam, Kunal Singh, Shivani Sabhlok, P. Sadayappan
Department of Computer Science and Enginnering, The Ohio State University, USA
{*hong.589, sukumaranrajam.1, singh.980, shivanisabhlok.1, sadayappan.1*}*@osu.edu*

***Abstract*—Sparse matrix-vector (SpMV) multiplication is a fundamental primitive for applications in computational science and data science because it exploits the fact that matrices that arise in practice are often very sparse. Additional savings in space and execution time can be achieved by performing Sparse-Matrix-Sparse-Vector multiplication (SpMSpV) when the input vector for an SpMV computation also exhibits sparsity. An API for SpMSpV has therefore been devised in the GraphBLAS library of primitives for expressing graph algorithms in the language of sparse linear algebra. However, high-performance SpMSpV implementations are not yet available due to several challenges. In this paper, we develop an efficient SpMSpV scheme for GPUs. The experimental results demonstrate superior performance over the current state-of-the-art alternatives.**

I. Introduction

Graphs represent a key data structure [1], and algorithms that work on graphs play a critical role in many applications such as social media [2] and bioinformatics [3]. Because of the duality between a graph and a matrix, a matrix representation of a graph [4] can be used to cast a wide range of graph algorithms using a small handful of sparse matrix primitives [1]. This led to the establishment and popularity of the CombBLAS and subsequent GraphBLAS [1], [5]–[8] libraries. In GraphBLAS, a small core set of matrix-based operators is defined, which can be used to implement a large number of graph algorithms [1]. Among these matrix-based operators, Sparse matrix-vector multiplication (SpMV) and Sparse-matrix sparse-vector multiplication (SpMSpV) are arguably the most important ones in GraphBLAS since very many graph algorithms can be modeled as a sequence of matrix-vector multiplications [1], [9]. While a tremendous amount of research has been directed at efficient GPU implementations of SpMV (product of a sparse matrix and dense vector), the SpMSpV variant (product of sparse matrix and sparse vector) has not yet been adequately addressed.

Graphics Processing Units (GPUs) are very attractive for computations on sparse matrices and vectors because of high computing power and memory bandwidth. However, actually achieving the high potential performance of GPUs is challenging because of several factors such as uncoalesced memory access, insufficient concurrency to tolerate high memory access latency, load imbalance, and thread divergence.

SpMSpV can be defined as $y = Ax$, where A is a sparse matrix of dimension $M \times N$, x is a sparse input vector of dimension $N \times 1$, and y is a (possibly) sparse output vector of dimension $M \times 1$. The SpMSpV primitive can be used in graph algorithms such as bipartite graph matching [10], breadth-first search (BFS) [11], maximal independent sets (MIS) [12], and single-source shortest path (SSSP) [13]. It is also a critical kernel in machine learning and data science, e.g., Support-Vector Machines (SVM), where a sequential minimal optimization strategy (SMO) can be used. Recently, several research efforts have been directed towards the development to efficient SpMV on GPUs [14]–[19]. However, to the best of our knowledge, there has only been one prior published work on SpMSpV for GPUs [20], and there is no available GPU implementation of SpMSpV in GraphBLAS. Further, very few efforts have focused on SpMSpV on either parallel or sequential machines [8], [9], [21] and most linear algebra libraries do not implement a specialized SpMSpV primitive.

The performance of matrix-vector multiplication is inherently severely limited by memory-bandwidth. Let us first consider the dense case, for multiplying an $N \times N$ matrix by a vector of N elements. The total number of arithmetic operations is $2N^2$, and the minimum number of data elements to be moved from/to main memory is $N^2 + 2N$. This means that the upper bound on operational intensity (OI) is $\frac{2N^2}{N^2+2N}$, i.e. under 2 FLOPs/word, or 0.5 FLOPs/byte for single-precision. This is far below the machine balance parameter (ratio of peak arithmetic performance to peak memory bandwidth) for current CPUs and GPUs, but represents a fundamental and unavoidable limit. Next, let us consider the case of SpMV, where the $N \times N$ matrix is sparse and contains only nnz nonzero elements. When a sparse-matrix representation like CSR (Compressed Sparse Row) is used, the matrix requires $8nnz + 4N$ bytes of storage. The total number of FLOPs is $2nnz$, giving an OI upper bound of $\frac{2nnz}{8nnz+4N+4N+4N}$, i.e., around 0.25 FLOPs/byte for $nnz >> N$. This means that as long as the fraction of nonzeros is less than 50%, a sparse CSR representation should offer the best performance allowed by the inherent roofline limit, while a zero-filled dense matrix representation would offer a higher OI (since the meta-data for CSR sparse representation is avoided) and therefore be preferable if the fraction of nonzeros is greater than 50%. Finally, let us consider SpMSpV, where the input vector is also sparse, with only a fraction α of the elements of the input vector being non-zero. Now only the nonzero elements in the sparse matrix columns corresponding to nonzero indices in the input vector will be used for computing the sparse matrix-vector product. The result of such a computation will often also be sparse. If we denote the non-zero fraction of the result vector to be β, we have an OI upper bound that is $\frac{2nnz_{active}}{8nnz_{active}+4\alpha N+4\beta N}$, where $8nnz_{active}$, $4\alpha N$, and $4\beta N$ are the minimum data movements required for accessing the sparse matrix, input vector, and output vector, respectively.

Unlike the case for SpMV (dense input/output vectors), there is no benefit from an OI perspective to consider *zero-filling* of the input vector and perform SpMV instead of SpMSpV. However, many factors such as inadequate concurrency, uncoalesced data accesses, thread divergence, use of atomic operations, etc., cause achievable performance for SpMSpV to be considerably below the roofline limit.

The SpMSpV approach developed in this paper has the following features:

- Two different data structures and processing strategies are used to represent the input matrix and vector, depending on the fraction of non-zero values in the input vector. When that fraction is sufficiently low, a *sparse mode* is used; otherwise a *non-sparse mode* is used and explicit

zero-filling is done to create a dense input vector for processing.
- For the non-sparse mode, a new Non-uniform Tiled (NuT) representation is developed, to achieve good load balancing, efficient data access, and enhanced reuse of data.
- A machine learning approach is used to choose between non-sparse and sparse modes, based on the fraction of non-zero values in the input vector and properties of the sparse matrix.

Experimental evaluation is presented on a collection of sparse matrices used in several recent studies [9], [13], [16], [17], demonstrating superior performance over the existing state-of-the-art alternatives for SpMSpV.

II. Related work

Before presenting details on the SpMSpV scheme developed in this paper, we discuss related work.

A. SpMSpV

SpMSpV is performed by merging a set of "active" columns in the matrix. To help this merging, there are several data structures proposed such as bitvector in GraphMat [21], SParse Accumulator (SPA) and heap in CombBLAS [8], and buckets in Bucket-SpMSpV [9]. Among them, the current state-of-the-art SpMSpV framework would be Bucket-SpMSpV [9], where buckets are used to achieve work-efficiency in parallel SpMSpV on multi-core machines. Bucket-SpMSpV has been shown to perform better than other SpMSpV implementations on multi-core machines [8], [21] across various fractions of non-zeros in the input vector, and its code is publicly available. SpMSpV on GPUs has been also developed [20], where sorting is used to merge a set of active columns in the matrix, but the code is not publicly available. Therefore, we perform an indirect comparison with Bucket-SpMSpV in this paper. Note that GraphMat's SpMSpV has an O(N) cost no matter how sparse the input vector is.

B. SpMV

SpMV (Sparse Matrix-Vector product) can be defined as $y = Ax$, where A is a sparse matrix, and x and y are dense vectors. Due to high memory bandwidth and highly parallel architecture, GPUs are well suited for SpMV. There have been several works on improving SpMV performance in GPUs to enhance load balancing and to reduce data movement for the sparse matrix A.

To the best of our knowledge, the current state-of-the-art SpMV frameworks are CSR5 [15], MergeBased [16], CSR-adaptive [14], and Hola-SpMV [17]. Although the precise details of each scheme are different, they all employ similar high level strategies for load balancing and reduction of data movement while loading the input matrix and storing the output vector.

When the number of columns of the sparse matrix is high, the elements of the input vector may not fit in cache, causing repeated global-memory accesses to elements of the input vector. Further, since the access pattern to the input vector depends on the sparsity structure of the input sparse matrix, the scattered accesses of the input vector are normally uncoalesed. Thus, the reuse of the input vector is another critical factor to achieve high performance. A key difference between our work and previous works is that we take into consideration the data movement of the input and output vector.

Machine learning has been used in several studies for choosing among alternate sparse matrix formats for SpMV. Li et. al. developed an auto-tuning system on multi-core processors, and ruleset classifiers [22] have been used as a machine learning model to choose the optimal data representation for a sparse matrix [23], using characteristics of sparse matrices as input features for the classifiers. Sedaghati et. al. evaluated the effectiveness of using a decision tree classifier to choose the best sparse matrix representation for a given sparse matrix on GPUs [19]. Zhao et. al. showed how Convolutional Neural Network (CNN) can be applied to sparse matrix format selection on both of multi-cores and GPUs [24]. In contrast to these efforts in choosing between alternative sparse matrix representations for SpMV, our work in this paper uses a trained SVM (Support Vector Machine) to dynamically choose between one of two concurrently maintained representations for execution, based on the extent of sparsity of the input sparse vector for SpMSpV. We define a clustering factor to capture relevant features of the input sparse matrix and show that it significantly affects the accuracy of SVM in Sec. VI-E.

C. Graph processing frameworks

We next discuss specialized graph-processing frameworks, since an efficient SpMSpV implementation allows an alternate formulation of many graph algorithms.

The CSC (Compressed Sparse Column) structure is widely used in graph processing frameworks, both on multicore processors and GPUs. Graph processing frameworks using CSC on GPUs [13], [25]–[27] mainly target efficient load balancing and the efficient formation of next wavefront, since these are expensive on GPUs. However, the graph frameworks using a CSC representation still suffer from uncoalesced data accesses and use of atomic operations [28]. A data-centric abstraction is used in Gunrock [13] to achieve high productivity and high performance. Gunrock is currently one of the state-of-the art graph processing frameworks for GPUs. Groute [25] is an asynchronous graph processing framework, and very high performance has been demonstrated for a few graph algorithms like BFS and SSSP, achieved by using a soft priority queue for matrices having very high diameters. A CSC representation is used in Gunrock and Groute.

To address performance degradation that comes from CSC representation, several alternative data structures for the input matrix are developed. Cusha [29] is one of the frameworks to address the above limitations. It both refines a data structure used in [30] and uses the structure for alleviating the limitations by sacrificing extra data movement overheads. Multi-graph [28] uses a 2D partitioning structure for representing the graph edges. It is shown to be more effective than Cusha.

Since many graph algorithms such as BFS can be equivalently implemented by use of an SpMSpV primitive, in this paper we present direct performance comparisons with the state-of-the-art GPU graph processing framework Gunrock.

III. Overview

This subsection provides an overview of Hy-SpMSpV, a hybrid scheme for SpMSpV that uses a combination of a conventional *sparse* mode processing of the input vector, along with a new data representation and processing strategy for a zero-filled *non-sparse* processing mode of the input vector when it is sufficiently dense. As mentioned earlier, the key motivation for this hybrid strategy is that a zero-filled full input vector ensures the same data access pattern for the sparse matrix irrespective of the actual non-zero pattern in the input sparse vector and enables a pre-processed efficient representation to be used for such a non-sparse processing mode. The selection of which mode should be used for an invocation instance is done by use of a pre-trained SVM.

TABLE I
BFS FRONTIER SIZE VARIATION: KRON_G500_LOGN21

Iter	# of active columns	# of nonzeros in active columns	with sparse vector(ms)	with dense vector(ms)	Best possible time(ms)	with hybrid (ms)
1	1	3	0.03	9.22	0.03	0.03
2	3	23431	0.05	9.24	0.05	0.05
3	21299	62444686	9.43	9.29	9.29	9.43
4	1316563	117152950	19.73	9.27	9.27	9.27
5	214054	2103704	0.69	9.19	0.69	0.69
6	302	357078	0.06	9.20	0.06	0.06
total time			30.99	55.41	19.39	19.39

We present some "look-ahead" data in Table I to justify our hybrid approach to SpMSpV. The typical use-case for SpMSpV is in iterative algorithms where the same sparse matrix is used in matrix-vector products with different input vectors of differing sparsity patterns. Table I shows data from performing a breadth-first search (BFS) on the kron_g500_logn21 dataset. Each step in the BFS algorithm is implemented as an SpMSpV operation, where the sparse matrix has a non-zero structure corresponding to the edges in the graph and the sparsity pattern of the input vector at each iteration corresponds to the active frontier nodes in the BFS. The number of nonzeros in the sparse input vector can be seen to vary very significantly across the iterations. Therefore, the number of traversed edges (number of active nonzeros in the sparse matrix) also varies significantly. The execution time for each step using both the standard *sparse* mode and the *non-sparse* mode are shown. It can be seen that when the fraction of active columns is low, the sparse mode is much faster, but for many cases where the number of active columns is high, the non-sparse mode is considerably faster. For such applications, it is important to be able to quickly decide which mode should be used. We show the total time if an oracle was used to select the better of the two ("Best possible") and the actual time with our implementation of Hy-SpMSpV, with the use of a trained SVM for making the selection. It may be seen that significant speedup is obtained by using the hybrid approach - the decisions made by the trained SVM were perfect in this case.

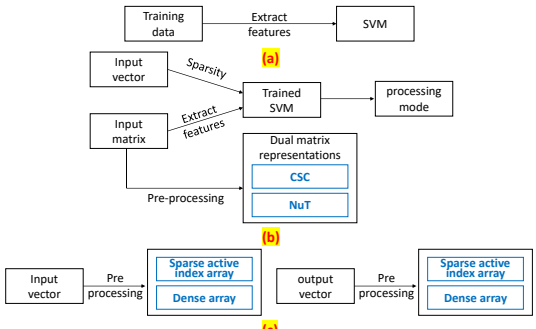

Fig. 1. Overview of Hy-SpMSpV

Fig. 1 gives a high-level overview for Hy-SpMSpV and pseudocode for Hy-SpMSpV shown in Listing 1.

The first step in using Hy-SpMSpV on a new GPU is to perform a one-time training of the associated SVM that uses a collection of sparse matrices capturing a range of sparsity patterns (Fig. 1 (a)). In Fig. 1 (a), "Training data" includes 35 sparse matrices from various application domains with different nonzero patterns and sizes. The suitable processing mode, sparse or non-sparse, is determined based on actual execution time of our sparse and non-sparse mode, given the sparsity of the input vector and features of the sparse matrix. The features used for the SVM classifier include: number of nonzeros (nnz), number of rows (N), number of columns (M), standard deviation of nnz in the column, and the clustering factor. We will collectively call these features as the deciding features.

Once the SVM has been trained using a representative set of sparse matrices, it is ready to be used for dynamic selection with an invocation of SpMSpV on an arbitrary matrix and input sparse vector (Fig. 1 (b)). Note that SVM was tested using the 5 fold cross-validation approach. The input sparse matrix is first processed to save it in the dual representations. In Fig. 1 (b), "Input vector" and "Input matrix" are assumed to be stored in the list format [8] and CSC representation, respectively. The two representations in which the input matrix is pre-processed and restructured are the standard CSC representation and our proposed NuT structure ("Dual matrix representations" in Fig. 1 (b)). Once the input matrix is stored in the dual representations, the deciding features of the input vector and the matrix are given as input to the trained classifier (in Fig. 1 (b)). Based on the calculated cross-over point for the sparsity of the input vector, the SVM appropriately sets "processing mode" to the non-sparse mode or sparse mode (line 1 in Listing 1).

Listing 1. Algorithms including SVM

```
1. processing_mode = SVM_classifier(N,M,nnz,...);
2. if processing_mode == non_sparse_mode then
3.   if current_mode == sparse_mode then
4.     out_vec = fill_nonactive_index_with_zero();
5.   end
6.   non_sparse_mode(in_vec, out_vec, NuT_mat);
7. else // next_mode = sparse_mode
8.   if current_mode == non_sparse_mode then
9.     act_idx = generate_sparse_active_index();
10. end
11. act_idx = sparse_mode(in_vec, out_vec, CSC_mat, act_idx);
12.end
```

As shown in Fig 1 (c), the input and output vectors are processed to appropriately initialize the "Sparse active index array" and "Dense array". "Sparse active index array" is of the same size as nnz in the vector and keeps indices of the nnz entries of the vector. "Dense array" is of the same size as that of the vector. The i-th element of this array contains the nnz value at the i-th index of the vector. In case of the non-sparse mode, "Sparse active index array" is ignored since all elements are assumed to be non-zeros and every uninitialized index of "Dense array" is initialized to 0 (line 4). In case of the sparse mode, 'Sparse active index array' (keeping non-zero indices) is generated if it does not exist (line 9).

If the non-sparse mode is chosen, SpMSpV is processed with NuT representation (line 6). In NuT, the sparse matrix is divided into non-uniform 2D tiles. Each tile contains almost the same number of non-zero elements of the input vector and the output vector needed for processing that tile. This guarantees the full reuse of the input vector and the output vector within that tile. Load balancing and efficient access of the sparse matrix is also considered to achieve high performance (motivated by [15], [16]). If the sparse mode is chosen, SpMSpV is done with CSC representation (line 11).

Due to space constraints, we do not present the details of the sparse mode here, but the full details will be available in the technical report [31]. In the sparse mode, load balancing and managing active indices of the input and output vectors are critical factors for high performance. In terms of load balancing and managing active indices, many current state-of-the-art implementations for the sparse mode on GPUs are quite similar [13], [25], [27], [28], [32], [33]. Our implementation for the sparse mode is also quite similar to these.

IV. HYBRID MODE (HY-SPMSPV)

As explained in the previous section, Hy-SpMSpV has two modes, each tailored for different input vector sparsity. In this

section, we provide more details on how switching between the sparse and non sparse mode is done based on the sparsity of the input vector and sparsity structure of the sparse matrix.

As mentioned in Sec. III, SVM is used for the decision making. Based on empirical measurements using a number of matrices, we observed some factors that influence the cross-over threshold: 1) If the standard deviation of non-zero elements in each row in the sparse matrix varies significantly across rows, the non-sparse representation tends to be better, since elements of the row having greater fraction of nnz elements can be accessed by multiple columns. 2) Sparsity distribution of the matrix – CSC is very sensitive to the nonzero distribution in the input. In general, when size of the cache is too small to hold the output vector and the nnz elements of the matrix are scattered, CSC tends to perform poorly. In this case, because the input vector is accessed one or few times, but the output vector can be accessed many times, the non-sparse mode is more favorable. If the nonzero elements of the matrix are clustered within rows, then cache locality can be improved. On the other hand, if the matrix has scattered elements, cache locality is decreased. To capture how many nonzeros of the sparse matrix are in tight row-clusters, we count the number of nonempty row segments across column panels (sets of contiguous columns of the sparse input matrix), and the clustering factor can be defined as $\frac{nnz}{the\_number\_of\_nonempty\_row\_segments}$. If the number of nonempty row segments is much smaller than nnz, many non-zero elements are expected to be clustered in the same row segments, and the clustering factor is high.

To extract the information about how edge distribution affects cache, irrespective of machine configurations, the input matrix was partitioned into column panels 13 times and different clustering factors are computed with different sizes of column panels, with width of the column panel being 128, 256, ..., 512K.

The SVM selects the desired processing mode for each iteration in repeated use scenarios. If the sparse mode was chosen in the previous iteration and non-sparse mode chosen in the current iteration, "Sparse active index array" is not maintained anymore since all elements of the input vector are assumed to be active. On the other hand, if the processing mode is changed from non-sparse to sparse, a parallel prefix-sum is used to regenerate indices of nonzeros for "Sparse active index array". In addition, when the non-sparse mode is chosen, sampling is done to estimate the fraction of non-zeros in the input vector.

V. Non-sparse mode

Regarding data access for y=Ax, existing implementations of SpMV for GPUs do not achieve good performance, if the input vector x is too large to fit in cache and the average reuse distance (in distinct number of elements accessed) between successive accesses to x_i is larger than the L2 cache capacity. The key contribution of our SpMV scheme is to improve the reuse of elements in the input vector by using a non-uniform 2D-tiling approach for SpMV, where tile sizes are dependent on the non-zero pattern of the sparse matrix.

Note that since the processing of the non-sparse mode can be viewed as a variant of SpMV , we will call the approach for the dense mode as SpMV.

A. Approach with NuT

Experiments using MergeSpMV, one of the state-of-the-art SpMV schemes for GPU, on soc-orkut and indochina-2004 from the Gunrock [13] dataset (characteristics listed in Table II) showed that even when the number of non-zeros for the matrices is similar, performance is very different. While soc-orkut achieves 9.5 GFLOPs due to large number of global memory transactions, indochina-2004 achieves 63.3 GFLOPs. This proved that the sparsity structure of the matrices significantly affected the performance on GPUs. Our scheme explicitly tries to improve the reuse of input elements rather than rely on cache, and results showed that soc-orkut achieved 37.9 GFLOPs and indochina-2004 97.96 GFLOPs using our non-uniform tiling scheme.

	soc-orkut	indochina-2004
rows / cols	3.0M	7.4M
nnz	213M	194M
clustered nnz	5%	72%
dram_trans / edge	7.98	2.12
Perf(Mergespmv)	9.54 GFLOPs	63.34 GFLOPs
Perf(non-sparse mode)	37.93 GFLOPs	97.96 GFLOPs

TABLE II
DATASET CHARACTERISTICS

Note that, in order to quantify clustering, we partition each sparse matrix into column panels, each with 192 columns. The number of rows with at least 16 elements in a column panel is counted and this count is then divided by the total number of non-zero elements to obtain the clustering percentage. Table II shows the clustering percentage for various matrices. Note that indochina-2004 has better clustering when compared to soc-orkut. Hence, soc-orkut may have much more uncoalesced accesses.

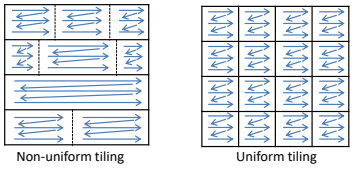

Fig. 2. Overview of NuT

A naive uniform tiling can have the following issues i) Due to tiling in most cases, many columns can be empty. Depending on the scheme, even unused input elements corresponding to the tile boundaries can be loaded ii) Load balance can be affected as each tile will only have a few non-zero elements. Our solution is to use non-uniform tiling that can explore data locality of both the input and the output vector elements without sacrificing good load balance. The tile size is determined by the sparsity structure. Fig. 2 shows the conceptual view of our non-uniform tiling. In our approach, we divide the sparse matrix into row panels (sets of contiguous rows of the sparse input matrix) . The adjacent rows of the sparse matrix are first grouped into row panels. Each row panel will have the equal number of rows (possibly except the last row panel). In a row panel, a column is considered to be active if there is at least one non-zero element in that column for the set of rows in that row panel. Contiguous columns in a row panel are grouped together, based on the number of active columns. Note that the column grouping is based on the number of active columns (not based on the number of columns itself). Each such partition will have the equal number of active columns (possibly except the last partition). Thus, the entire sparse matrix is divided into 2d blocks. The number of columns in each 2d block is determined, based on the number of active columns. We also load the elements on the input vector which are active to shared memory reducing the number of uncoalesced accesses.

B. Data structure

In our SpMV scheme, the input sparse matrix is represented by a set of 2d blocks. We use shared-memory to keep the

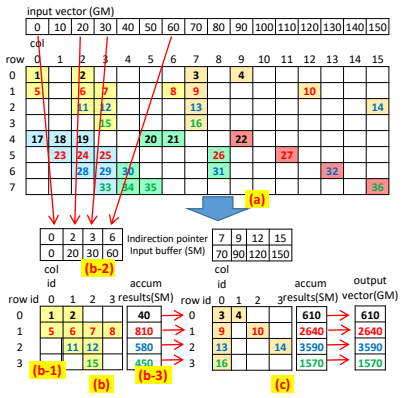

Fig. 3. Overview (SpMV)

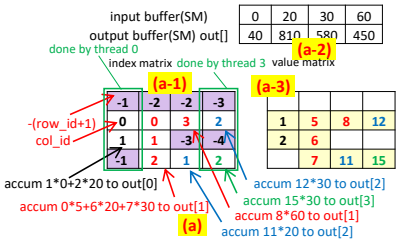

Fig. 4. Details (SpMV)

elements of both the input vector and output vector. Since the reuse of input vector elements is only across the rows of 2d block, the higher the number of rows, the higher the reuse of shared-memory. Hence, maximizing the number of rows in a 2d block can be beneficial. However, the number of rows in each block is limited by the amount of available shared memory. We determine the number of rows in a 2d block as $\frac{shared\_memory\_size\_per\_tb}{sizeof(data\_type)} \times \frac{1}{3}$ since we empirically found that this configuration is the best. $shared\_memory\_size\_per\_tb$ is obtained as $\frac{shared\_memory\_size\_per\_SM}{no\_of\_active\_tb\_per\_sm}$. We use thread blocks of size 1024, and then the number of active thread blocks per SM will be 2. In our SpMV scheme, only the input elements corresponding to the active columns are brought to shared-memory. The number of columns in a 2d block is selected such that the number of active columns in the selected row panel is equal to $\frac{shared\_memory\_size\_per\_tb}{sizeof(data\_type)} \times \frac{2}{3}$. The data structures used to store the actual non-uniform tiles are index-matrix (to hold the values of indices of input and output vector) and value-matrix (to hold the non-zero values from the sparse matrix).

The dimensionality and total number elements in both the matrices are the same. The total number of elements is computed as the sum of the number of threads in a thread block, number of nnz in the current 2d block and number of output indices which have at least one contribution from the current 2d block. The number of columns is equal to the number of threads in a thread block. The number of rows is $\lceil \frac{numberofelements}{numberofthreads} \rceil$. In the index-matrix, the output indices are encoded as $-(output\_index + 1)$. Input indices are represented as such (no encoding). Thus negative values represent the output indices and non-negative values indicate the input indices. The first element is the negated value of output index which receives a contribution from the current edge-block. This is followed by placing all the indexes of the input that contribute to the selected output index. Then the negated value of next output index is placed and that value is followed by the corresponding input indices and so on. While the matrix is being filled, if the row limit is reached, the output index is placed at the beginning of next column, which is followed by the remaining edges. We also maintain an indirection array per 2d block. The indirection array is a map from shared-memory index to input column index, and is used to identify the active columns in that 2d block.

The figure 3 depicts our scheme with row panel consisting of 4 rows, and each 2D block containing 4 active columns. In the first row panel in Fig. 3 (b) (rows 0 to 3), columns 0 to 6 form the first 2D block, as there are 4 active columns (0,2,3,6). Similarly, the second 2D block consists of columns 7 to 15 (active columns :7,9,12,15). The second row panel has three 2D blocks (0-3, 4-8, 9-15). The example shows the case where 4 elements from the input and 4 elements from the output vector are brought to the shared memory.

Fig. 4 shows the actual data representation for the example where the 2D block consists of rows 0 to 3 and columns 0 to 3 (Fig. 3 (b)). The corresponding data representation is shown in Fig. 4 (a). The output index 0 gets 2 contributions from this block. The first entry in the index-matrix is -1 which corresponds to the output index 0. The contributions that output index 0 gets are from columns 0 and 2, which are mapped to index 0 and index 1 in shared-memory. The output index in 2D block is followed by (in transposed order) values 0 and 1 corresponding to the indirection indices that contribute to output index 0. Similarly, the output index 1 has four contributions. So the 2nd column in 2D block representation contains -2 (output index 1), 0,1,2 (corresponding to indirection indices 0,1,2 which in turn correspond to input indices 0,2,3). Since it has 4 contributions and the number of rows in the 2D representation is only 4, the output index 1 is copied to the beginning of the next column, which is then followed by the corresponding input edge indirection id 3. Corresponding to each input index, the value matrix contains the actual value of the sparse matrix. For example, in the first 2d block, the value in location row 0, col 0 is 1. This location is represented by the element row=1, col=0 in the index matrix. Correspondingly, row=1, col=0 contains 1 which is the non-zero value.

In Fig. 3 (b), the indirection array for the first 2d block (yellow) is shown by Fig. 3 (b-2) and has the values (0,2,3,6). This marks the active blocks and while processing the first 2d block, the elements (0,20,30,60) corresponding to indices 0,2,3,6 will be loaded from global-memory to input buffer in shared-memory.

C. Algorithm

The overview of our SpMV approach is as follows. All blocks in a row panel are processed by a single thread block. For a given row partition, the corresponding output elements in the shared memory are initialized to zero. Then, for each block in that row partition, the active elements of the input vector are brought to shared memory. The entire work in each block is distributed across threads such that the work distribution is load balanced. Each thread then processes the work assigned to it and accumulates the partial product in registers. Finally, the partial product held in registers is updated to the corresponding value in shared memory. After processing each such 2d block in a block partition, all threads in a thread block collectively move the elements from the shared memory to corresponding location of output vector in global memory. Pseudocode for our SpMV algorithm is shown in Listing 2. Each thread block identifies the set of rows that it should process (line 1, 2). Then all the threads in a thread block collectively initialize the corresponding output location in shared memory to zero (line 4 to 6). Then all the 2d block in the selected rows are

processed one after the other (loop in line 7). For a given 2d block the threads collectively bring in the elements of the active elements in the input vector, using the indirection pointer, to shared memory (line 9 to 11). The threads then compute the partial product in a load balanced manner. Each thread initially identifies the output index corresponding to its first contribution from the current 2d block (line 16). Then it processes all the rest of the rows in index-matrix (loop in line 18). Each thread accumulates the partial products in a thread local register. If the value in the index-matrix is negative, then it indicates the end of the current row in sparse matrix. Hence, the value in the thread local register is updated to shared-memory (line 20 to 22). Once the entire 2d block is processed, all the threads in a thread block collectively move the result from shared-memory to corresponding location in global memory. Note that in line 21, an atomic operation is not required (All the threads in the same warp cannot reach 21 at the same time for the same output vertex. Also, as the synchronization statement in line 27, threads across multiple warps cannot reach line 21 for the same output vertex simultaneously). When the input matrix is very small, a special strategy is used. In the previous strategy, we try to maximize the number of rows in each row partition to maximize the reuse of input vector. Note the number of threads blocks is equal to the number of row partitions. When the input matrix is small, this strategy will result in lower number of thread blocks. Low number of thread blocks can cause load imbalance across SMs. In order to avoid it, for small input matrices, we divide the sparse matrix into N row partitions such that each row partition has almost the same amount of non-zero elements where N = number of SMs X 2.

Listing 2. SpMV Pseudocode

```
1. row_start = start_row_position[tb_id];
2. row_end = start_row_position[tb_id+1];
3. // initialize shared memory for outputs
4. for i=row_start+tid to row_end-1 step tb.size() do
5.  sm_dest_value[i-row_start] = 0;
6. end
7. for i=seg_start_num[tb_id] to seg_start_num[tb_id+1]-1 do
8.  // load GM data to SM
9.  for j=tid to IN_TILE_ROW_SIZE-1 step tb.size() do
10.  sm_input_value[j] =
       input_value[indirect_pnt[i*IN_TILE_ROW_SIZE+j]];
11. end
12. __syncthreads;
13. start = start_seg_position[i];
14. end = start_seg_position[i+1];
15. num_rows =
      floor((end-start + tb.size() - 1 - tid)/tb.size());
16. dest_id = abs(seg_index[i][0][tid] + 1);
17. val = 0;
18. for j=1 to num_rows-1 do
19.   cur_val = seg_index[i][j][tid];
20.   if cur_val < 0 then
21.     sm_dest_value[dest_id] += val;
22.     dest_id = abs(cur_val + 1);
23.   else
24.     val +=
          sm_input_value[dest_id] * seg_value[i][j][tid];
25.   end
26. end
27. __syncthreads;
28. atomicAdd(&sm_dest_value[dest_id], val);
29.end
30.__syncthreads;
31.for i=row_start+tid to row_end-1 step tb.size() do
32. dest_value[i] = sm_dest_value[i-row_start];
33.end
```

VI. Experimental Evaluation

This section details the experimental evaluation of Hy-SpMSpV for three different scenarios 1) SpMSpV, and 2) SpMV, and 3) Graph algorithms. The experiments were performed on two GPU systems: an Nvidia P102 GPU (TITAN X) with 28 Pascal SMs, 1417MHz, 12GB global memory, global memory bandwidth of 480GB/sec, 4MB L2 cache, and 96K shared memory; and an Nvidia K40c GPU with 15 SMs, 12GB global memory, global memory bandwidth of 288GB/sec, 1.5MB L2 cache, and 48K shared memory. In all the experiments, ECC was turned off on K40c (P102 does not have ECC memory), and the compiler optimization flag -O3 with NVCC 9.1 and GCC 4.9.2 was used. For all the schemes, we only include the kernel execution time; preprocessing time and data transfer time from CPU to GPU are not included, for Hy-SpMSpV and for other implementations. The impact of Hy-SpMSpV pre-processing overhead is reported separately. All tests were run 5 times with single precision, and average numbers are reported.

For usage as SpMSpV primitive, the Hy-SpMSpV primitive is compared with a current state-of-the-art implementation, Bucket-SpMSpV [9]. This is not an apples-to-apples comparison because Bucket-SpMSpV only works on multicore CPUs and Hy-SpMSpV only works on GPUs. We perform this comparison since we know of no other publicly available GPU implementation of SpMSpV. Since the peak performance of the GPU and multicore CPU systems are vastly different, we also report the achieved fraction of roofline performance limits for the two codes. The experiments for Bucket-SpMSpV were performed on an Intel Xeon CPU E5-2680 v4 with 28 cores, 2400MHz, 35M L3 cache (shared across cores), 256K L2 cache (for each core), 32K L1 cache (for each core), memory bandwidth of 51.2GB/s. The comparison of Hy-SpMSpV with publicly available state-of-the-art alternatives is shown in Figure 5 and 6.

For usage as an SpMV primitive, the non-sparse mode of Hy-SpMSpV can be modified to compute SpMV. We were not able to compare the non-sparse mode with CSR adaptive [14] as its code was not available, and with Hola-SpMV [17] as we could not build it on our environment. We found that on our GPUs, MergeSpMV [16] quite consistently performs better than CSR5 [15]. So, we compare the non-sparse mode with MergeSpMV.

For usage in Graph algorithms, Hy-SpMSpV is used to implement Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), Betweenness Centrality (BC). Gunrock provides a "DO (Direction-Optimized)-BFS" option that is very powerful for scale-free graphs. However, we only implemented a normal BFS using Hy-SpMSpV and report two variants of Gunrock: with and without DO-BFS. Performance with Enterprise [33] is also reported. We did not include B40c [32] since performance with Enterprise has shown to be better.

We report performance in pseudo Millions of Traversed Edgesper Second (MTEPS) as has been done in the literature [13], [28]. For BFS and SSSP, MTEPS is computed as $\frac{nnz}{T \times 10^6}$, where nnz is the number of nonzeros in the matrix and T is the execution time (sec). For BC, MTEPS is computed as $\frac{2 \times nnz}{T \times 10^6}$.

A. Datasets

Since we compared Hy-SpMSpV against Gunrock and SpMSpV-bucket, we combined the datasets used in their work for evaluating both graph primitives and SpMSpV primitives. In the same way, we compared our SpMV primitive against MergeSpMV, and used the datasets presented in their paper. The first dataset corresponds to the one used in Gunrock [13]. These datasets are available at http://gunrock.github.io (except RMAT22, RMAT23, and RMAT24, which we generated with RMAT generator [26]). Since performance trends for RMAT22, RMAT23, and RMAT24 were very similar, we only report performance for RMAT24.

The second dataset corresponds to the one used in Bucket-SpMSpV [9]. Table III shows the corresponding datasets (two datasets are mixed and then sorted along diameters).

The third dataset consists of the common matrices used in the current state-of-the-art SpMV frameworks: Merge-SpMV [16] and Hola-SpMV [17]. These matrices can be downloaded from [34].

For evaluating graph algorithms, the index of the source vertex is always set to 0 as done in Gunrock. We did not use datasets used in [16], [17] for graph algorithms since the source vertex is sometimes connected to only a few vertices.

type	name	rows	cols	nnz	(pseudo) diameter
small diameter	rmat24	16.8M	16.8M	520.1M	6
	kron_g500-logn21	2.1M	2.1M	182.1M	6
	soc-orkut	3.0M	3.0M	212.7M	9
	hollywood-2009	1.1M	1.1M	113.9M	11
	wikipedia-20070206	3.6M	3.6M	45.0M	14
	web-Google	0.9M	0.9M	5.1M	16
	soc-LiveJournal1	4.8M	4.8M	69.0M	16
	amazon0312	0.4M	0.4M	3.2M	21
	indochina-2004	7.4M	7.4M	194.1M	26
	ljournal-2008	5.4M	5.4M	79.0M	34
	wb-edu	9.8M	9.8M	57.2M	38
	dielFilterV3real	1.1M	1.1M	89.3M	84
large diameter	G3_circuit	1.6M	1.6M	7.7M	514
	roadNet-CA	2.0M	2.0M	5.5M	849
	delaunay_n24	16.8M	16.8M	100.7M	1718
	rgg_n_2_24_s0	16.8M	16.8M	265.1M	3069
	hugetric-00020	7.1M	7.1M	21.4M	3662
	hugetrace-00020	16.0M	16.0M	48.0M	5633
	road_usa	23.9M	23.9M	57.7M	6262

TABLE III
DATASETS USED IN GUNROCK AND BUCKET-SPMSPV

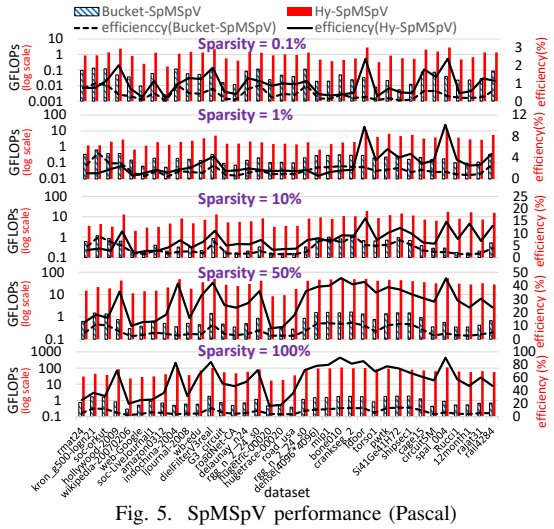

Fig. 5. SpMSpV performance (Pascal)

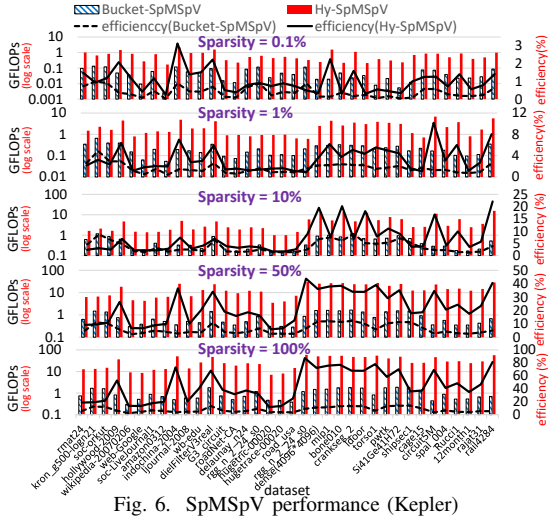

Fig. 6. SpMSpV performance (Kepler)

B. SpMSpV

Fig. 5 (for Pascal) and 6 (for Kepler), compares Hy-SpMSpV with Bucket-SpMSpV [9] (on a 28-core CPU, as detailed earlier). The vertical bars (log scale) show GFLOPs and the connected lines show achieved fractional efficiency. Reported GFLOPs are computed as $\frac{nnz_{active} \times 2}{execution\_time}$ where nnz_{active} is the number of elements of the sparse matrix involved in SpMSpV. The GPUs have significantly higher memory bandwidths, and Hy-SpMSpV performs much better. When the fraction of nonzeros in the input vector is increased, the speed-up over Bucket-SpMSpV increases. For example, on the Pascal GPU, when the fraction of non-zeros is 0.1%, Hy-SpMSpV is on average 25.62x faster than Bucket-SpMSpV. But, when fraction of non-zeros is 100%, Hy-SpMSpV is on average 67.96x faster. This is because the non-sparse mode is chosen when sparsity is sufficiently high, and SpMSpV computations are done more efficiently.

To compare Hy-SpMSpV with Bucket-SpMSpV, Roofline [35] upper bound GFLOPs based on memory bandwidth can be developed as follows. Suppose CSC representation (col_pointer[], row_index[], row_value) is used, and that act_{in} and act_{out}, P denote the number of non-zeros in the input vector, the output vector, and sparsity, respectively. The data footprint required for accessing col_pointer[] and row_index[] (or row_value[]) can be approximated to 4×min(N+1, act_{in}) and 4×nnz×P, respectively. The total floating-point operation count is 2×nnz×P. The data footprint required for accessing the input vector and the output vector would be 4×(act_{in}+act_{out}). Thus, the total data footprint would be $DATA$ = 4×MIN(N+1, act_{in})+8×nnz×P + 4×(act_{in}+act_{out}). Hence, this is the minimum possible volume of $DATA$ to be moved from memory to registers. Dividing this volume by the peak memory bandwidth BW_M gives the minimum execution time and the roofline upper bound of GFLOPs: $\frac{2 \times nnz \times P \times BW_M}{DATA}$. Finally, efficiency (= $\frac{achievable\_actual\_GFLOPs}{the\_roofline\_upper\_bound\_GFLOPs}$) is used as the comparison of metric. As shown in Fig. 5 and 6, efficiency of Hy-SpMSpV is mostly higher than that of Bucket-SpMSpV. Efficiency of Bucket-SpMSpV is not much affected by sparsity. On the other hand, efficiency of Hy-SpMSpV improves as the non-zero fraction in the sparse input vector gets larger, since the non-sparse mode is chosen.

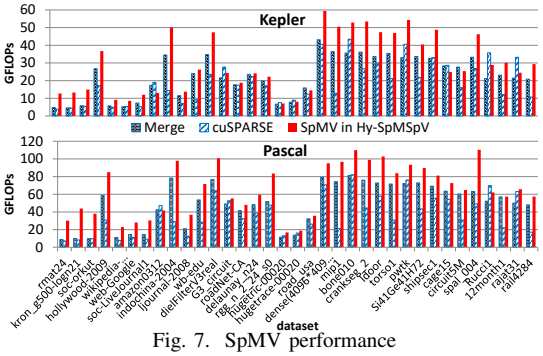

Fig. 7. SpMV performance

C. SpMV

Fig. 7 shows SpMV performance on the two GPUs. Since Hy-SpMSpV tries to capture the reuse of input, performance with the non-sparse mode is significantly better than performance with others for power-law graphs such as Soc-LiveJounal1 and soc-orkut. For these datasets, the accesses to the input vector are mostly uncoalesced. Hence, performance of other implementations is significantly degraded.

For other datasets, Hy-SpMSpV mostly outperforms others. We note that there are two popular cuSPARSE formats (CSR and HYB), and for each of dataset, we reported the performance of the better one. Since yaSpMV [18] is impractical, having up to 150,000× preprocessing overhead normalized to the time for one SpMV [17], we do not consider it in this work.

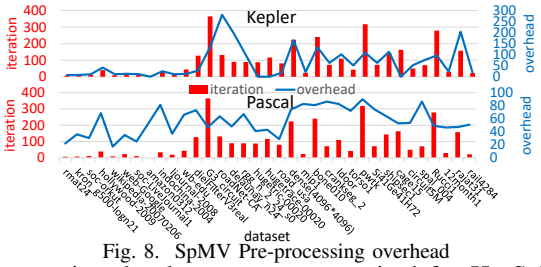

Fig. 8. SpMV Pre-processing overhead

Constructing the data structures required for Hy-SpMSpV incurs an additional overhead. Figure 8 shows the preprocessing time normalized to the time for one iteration with Hy-SpMSpV (blue curve) and the number of iterations for performing better than the best compared alternative (MergeSpMV or cuSPARSE) (red bar). Hence, each value of red bars is calculated by $\frac{pre-processing\_time}{min(cuSPARSE\_time, MergeSpMV\_time) - Hy\_SpMV\_time}$ if our SpMV is faster than others. We note that many applications using SpMV/SpMSpV execute a large number of iterations with the same sparse matrix. As seen in Fig. 8, the number of iterations for compensating pre-processing time is around 10 for many of the power-law graphs such as soc-LiveJournal and soc-orkut. Hy-SpMSpV can be especially useful for these kinds of datasets, as many graph algorithms are iterative. We note that pre-processing is done on the GPU and thus transfer time from host to the GPU is not included. We convert the CSC representation (column indices within each row assumed sorted) to NuT representation. For each row panel, sorting and prefix-sum are used to construct indirection-array to reorder and renumber edges.

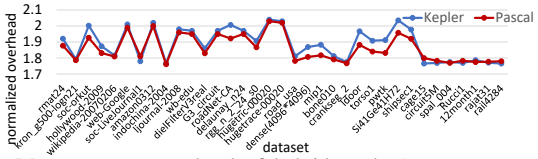

Fig. 9. Memory space overhead of hybrid mode (non_sparse and sparse modes)

The space overhead for the dual representation (CSC and NuT structures), normalized to that of the standard CSC structure, is presented in Fig. 9. The space required for the NuT structure is usually smaller than that for the CSC structure since each of nonzero indices is stored using a short-integer type (2 bytes). However, in some cases, the space overhead for NuT is larger than that of CSC since row indices can appear many times across threads. In addition, the space overhead on the Pascal GPU is smaller than that on the Kepler since row indices appear less frequently for the Pascal due to the larger shared-memory capacity.

D. Graph Primitives

We compared BFS performance with two current state-of-the-art BFS implementations on GPU: Gunrock [13] and Enterprise [33], and a CPU implementation with Bucket-SpMSpV. Fig. 10 (a) and 11 (a) present performance data in MTEPS for BFS. In Fig. 10 and 11, the vertical bars (log scale) show MTEPS and the dotted lines show the achieved speedup over the highest MTEPS among Bucket-SpMSpV, Gunrock, and Enterprise (speed-up(the best)), the achieved

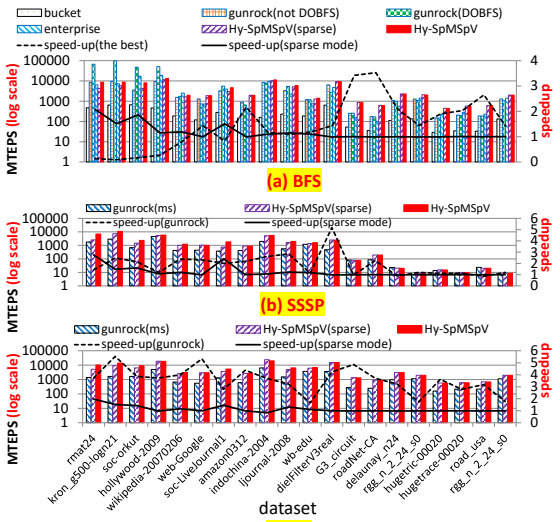

Fig. 10. BFS, SSSP, and BC performance (Pascal)

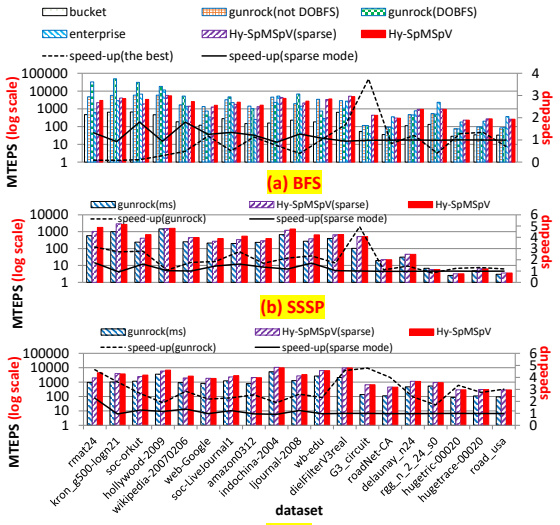

Fig. 11. BFS, SSSP, and BC performance (Kepler)

	BFS				SSSP		BC	
	gunrock (not DOBFS)	gunrock (DOBFS)	enterprise	Hy-SpMSpV (sparse mode)	gunrock	Hy-SpMSpV (sparse mode)	gunrock	Hy-SpMSpV (sparse mode)
overall	1.79	1.24	1.84	1.15	1.67	1.18	3.38	1.11
low dia.	1.42	0.73	1.42	1.26	2.14	1.33	3.69	1.20
high dia.	2.53	2.72	2.71	1.00	1.15	0.99	2.95	0.99

TABLE IV
OVERALL SPEED-UP ON PASCAL(GEOMETRIC MEAN)

	BFS				SSSP		BC	
	gunrock (not DOBFS)	gunrock (DOBFS)	enterprise	Hy-SpMSpV (sparse mode)	gunrock	Hy-SpMSpV (sparse mode)	gunrock	Hy-SpMSpV (sparse mode)
overall	1.79	1.24	1.84	1.11	1.72	1.15	2.67	1.08
low dia.	1.42	0.73	1.42	1.20	2.25	1.27	2.71	1.15
high dia.	2.53	2.72	2.71	1.00	1.15	0.99	2.60	0.99

TABLE V
OVERALL SPEED-UP ON KEPLER (GEOMETRIC MEAN)

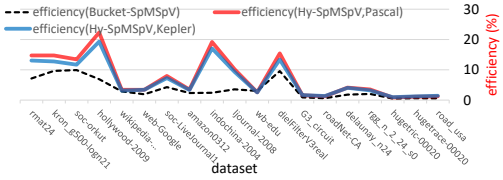

Fig. 12. Efficiency (BFS)

TABLE VI
CROSS-OVER TRAIL NUMBER WITH ASYMTOTIC SPEED-UP

	Kepler					Pascal				
	BFS		SSSP		BC	BFS		SSSP		BC
dataset	cross	asym	cross	asym	asym	cross	asym	cross	asym	asym
rmat24	X	0.09	4	1.81	2.06	X	0.13	2	3.92	3.72
kron_g500-logn21	X	0.08	7	3.13	4	X	0.09	4	3.67	5.53
soc-orkut	X	0.11	2	1.72	2.08	X	0.17	1	3.43	3.85
hollywood-2009	X	0.29	75	1.05	2.19	X	0.26	18	1.27	3.73
wikipedia-20070206	X	0.49	2	1.77	2.1	X	0.79	1	2.88	3.99
web-Google	19	1.18	2	1.28	2.24	8	1.43	1	2.26	5.33
soc-LiveJournal1	X	0.51	2	1.69	1.88	X	0.83	1	4.84	2.79
amazon0312	28	1.13	2	1.25	2.79	3	2.17	1	2.17	4.38
indochina-2004	X	0.74	5	2.53	2.62	45	1.17	3	2.78	3.7
ljournal-2008	X	0.39	2	1.38	2.11	108	1.09	1	3.5	3.23
wb-edu	201	1.04	3	1.66	2.4	14	1.18	5	1.33	1.66
dielFilterV3real	9	1.69	1	4.94	4.57	24	1.44	1	5.1	4.24
G3_circuit	1	3.74	1	1.14	4.8	1	3.42	X	0.96	4.85
roadNet-CA	X	0.83	1	1.49	4.08	1	3.53	1	2.25	3.71
delaunay_n24	7	1.21	X	0.84	2.42	3	2.09	X	0.88	3.23
rgg_n_2_24_s0	X	0.41	1	1.25	1.73	6	1.44	1	1.19	1.74
hugetric-00020	1	1.28	1	1.34	3.35	2	1.9	1	1.1	3.62
hugetrace-00020	5	1.34	1	1.2	2.78	3	2.03	1	1.12	2.78
road_usa	X	0.71	1	1.15	3.04	1	2.65	X	0.84	3.18

speedup over Gunrock (speed-up(gunrock)), or the achieved speedup over the sparse mode of Hy-SpMSpV (where the sparse mode is always chosen).

When DOBFS is enabled, Gunrock is often several times faster than Hy-SpMSpV for scale-free graphs with very small diameter, since only a subset of edges are swept. When diameter is longer than 9, DOBFS does not work well and Hy-SpMSpV BFS usually outperform other alternatives.

Similar to SpMSpV evaluation, efficiency for Bucket-SpMSpV and Hy-SpMSpV for BFS is presented in Fig. 12. In BFS, all vertices and edges are normally touched, the upper bound MTEPS could be approximated to $\frac{2\times nnz\times BW_{GM}}{8\times nnz+12\times N+4}$. As shown in Fig. 12, efficiency of Hy-SpMSpV is much higher than that of Bucket-SpMSpV especially for datasets having a short diameter.

For SSSP and BC, Hy-SpMSpV significantly outperforms them as shown in Fig. 10 (b,c) and Fig. 11 (b,c). Note that we implemented BFS, SSSP, and BC algorithm used in Gunrock.

In addition, computing exact BC requires the execution of a BFS like algorithm rooted from all vertices, which is prohibitively expensive. Hence, the approximated value of BC is computed as done in Gunrock.

BFS and SSSP primitives are used in many applications such as web crawling [36] and closeness centrality [37]. In those applications, hundreds to millions of BFSs or SSSPs with different source vertices of the same graph are required. In Table VI, we measure end-to-end runtime (including data transfer between device and host, and the pre-processing overhead of Hy-SpMSpV), and report the cross-over number of trails where Hy-SpMSpV outperforms all other implementations with the asymtotic speed-up (assuming the number of trial is ∞). As seen in Table VI, the cross-over number of trails are normally very small. Moreover, our SSSP implementation outperforms others only with one trail for roughly half of the datasets. Hence, our SSSP implementation can be used for both of single trial of SSSP as well as multiple trails of SSSPs. In BC, preprocessing time would be negligible since trails with all source vertices are needed.

Table IV and V present geometric means of the speedup of Hy-SpMSpV over other implementations. In addition, Fig. 13 shows the fraction of time spending on the non-sparse mode for BFS, SSSP, and BC. For BFS, when a diameter is short, Gunrock is faster than Hy-SpMSpV because of DO-BFS strategy. Also, when a diameter is short, the achievable speedup is high. This is because of exploiting two data representations as seen in Table I in Sec. III. As shown in Fig. 13, when a diameter is short, the time for processing

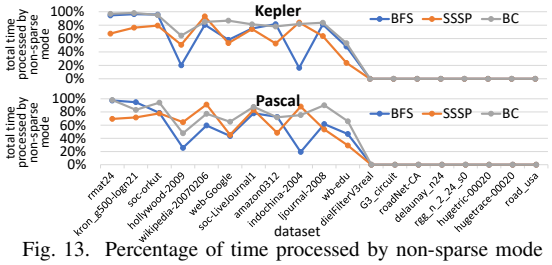

Fig. 13. Percentage of time processed by non-sparse mode

the non-sparse mode normally dominates the total execution time. When a diameter is long, sparsity for each iteration is not large in the datasets. Hence, the non-sparse mode is never chosen as shown in Fig. 13. That is why, the speedup over the sparse mode is almost 1 when diameters are long. Also, when diameters are short, the hybrid mode of Hy-SpMSpV performs better than the sparse mode of Hy-SpMSpV, which verifies the effectiveness of the hybrid mode. Note that the overhead for the switching cost is included in the hybrid mode.

E. Assessment of Effectiveness of SVM

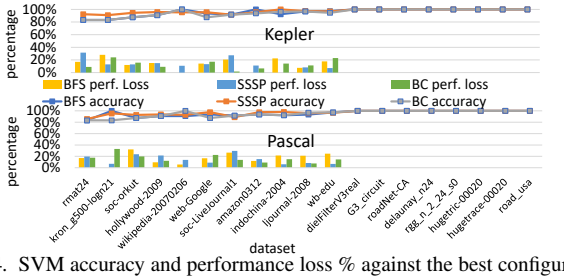

Fig. 14. SVM accuracy and performance loss % against the best configuration

TABLE VII
SUMMARY OF FIG. 14 (ARITHMETICAL MEAN)

	Kepler						Pascal					
	BFS		SSSP		BC		BFS		SSSP		BC	
	frac	loss	frac	loss	frac	loss	frac	loss	frac	loss	frac	loss
all	95.6%	8.1%	97.1%	7.9%	95.0%	6.9%	95.5%	9.8%	96.6%	8.4%	95.0%	8.7%
w/o nnz	95.5%	9.0%	97.1%	8.5%	94.8%	7.3%	95.5%	10.4%	96.5%	8.4%	94.7%	8.8%
w/o #row,#col	95.4%	9.2%	97.0%	8.3%	94.8%	8.0%	95.3%	10.5%	96.5%	9.2%	94.9%	9.4%
w/o stdev (nnz)	94.8%	12.0%	96.7%	10.8%	94.2%	10.2%	94.6%	13.9%	96.0%	12.1%	93.9%	12.6%
w/o cluster factor	93.5%	17.9%	95.7%	16.7%	92.6%	15.9%	93.2%	22.9%	95.0%	21.0%	92.2%	21.4%

Figure 14 shows the accuracy of the SVM (for each dataset for BFS, SSSP, and BC on two machines. The curves show the fraction of the correct SVM output (choosing the more efficient mode correctly) across all iterations. The vertical bars show the performance loss against the ideal configuration that was obtained by summing the minimum of the execution time with sparse mode and non-sparse mode for each iteration (i.e., the switching cost is not included). The summary is also presented in Table VII ("frac" and "loss" denote the fraction of the correct SVM output and performance loss, respectively). When diameter is short, SVM generally chooses the more efficient mode correctly, but performance loss is sometimes more than 20% as seen in Fig. 14. In those datasets, the execution time for a few iterations dominates the total execution time. That performance loss comes from misprediction for those iterations. When diameter is high, the sparse mode is always better than the non-sparse mode, as seen in Fig. 13. The SVM always chooses the more efficient one (sparse mode) as seen in Fig. 14.

To understand the utility of the various features used in SVM training, we performed experiments by training the SVM with the exclusion of one or more of the features. We classified features into four groups: N and M; nnz; standard deviation of nnz; and clustering factors with different column panel sizes.

We trained the SVM four times by excluding one grouped feature each time, to assess the impact of that grouped feature. As seen in Table VII, all features do contribute to effectiveness, with the set of clustering factors being the most significant among the grouped features.

VII. Discussion

The dual represenation of the sparse matrix and vector in Hy-SpMSpV results in a doubling of memory requirements. However, as shown in Sec. VI, the NuT representation enhances performance. Hence, there is a tradeoff between storage requirement and performance.

Another issue to consider is the impact of using non-standard sparse matrix representations. Applications often use many library functions, which are based on standard representations. For sparse matrices, CSR and CSC are standard representations. Our approach in developing Hy-SpMSpV is to build on the standard CSC representation, by adding additional data. Thus, it will be feasible to use just the CSC component of the Hy-SpMSpV sparse matrix representation for use with other libraries that expect a standard CSC representations, e.g., other GraphBLAS operations. Our goal is to incorporate the Hy-SpMSpV implementation into GraphBLAS.

VIII. Conclusion

In this paper, an efficient SpMSpV scheme for GPUs has been developed. This paper uses dual representations and execution strategy for efficient SpMSpV computations. One of two representations is effectively chosen by using trained SVM based on features of the matrix and the input vector. Experimental results demonstrate that the scheme in this paper can achieve high performance.

References

[1] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, "Mathematical foundations of the GraphBLAS," in *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*. IEEE, 2016, pp. 1–9.

[2] J. Riedy and D. A. Bader, "Multithreaded community monitoring for massive streaming graph data," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1646–1655.

[3] E. Georganas, A. Buluç, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick, "Parallel de bruijn graph construction and traversal for de novo genome assembly," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 437–448.

[4] R. A. Brualdi, "Kronecker products of fully indecomposable matrices and of ultrastrong digraphs," *Journal of Combinatorial Theory*, vol. 2, no. 2, pp. 135–139, 1967.

[5] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson *et al.*, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–2.

[6] T. G. Mattson, C. Yang, S. McMillan, A. Buluç, and J. E. Moreira, "GraphBLAS C API: Ideas for future versions of the specification," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–6.

[7] J. Kepner, D. Bader, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the GraphBLAS: Seven good reasons," *Procedia Computer Science*, vol. 51, pp. 2453–2462, 2015.

[8] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[9] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," in *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017, pp. 688–697.

[10] A. Azad and A. Bulu, "Distributed-memory algorithms for maximum cardinality matching in bipartite graphs," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 32–42.

[11] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 65.

[12] A. Buluc, E. Duriakova, A. Fox, J. R. Gilbert, S. Kamil, A. Lugowski, L. Oliker, and S. Williams, "High-productivity and high-performance analysis of filtered semantic graphs," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 237–248.

[13] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *ACM SIGPLAN Notices*, vol. 51, no. 8. ACM, 2016, p. 11.

[14] M. Daga and J. L. Greathouse, "Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices," in *High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 64–74.

[15] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 339–350.

[16] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2016, p. 58.

[17] M. Steinberger, R. Zayer, and H.-P. Seidel, "Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU," in *Proceedings of the International Conference on Supercomputing*. ACM, 2017, p. 13.

[18] S. Yan, C. Li, Y. Zhang, and H. Zhou, "YaSpMV: Yet another SpMV framework on GPUs," in *Acm Sigplan Notices*, vol. 49, no. 8. ACM, 2014, pp. 107–118.

[19] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan, "Automatic selection of sparse matrix representation on GPUs," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 99–108.

[20] C. Yang, Y. Wang, and J. D. Owens, "Fast sparse matrix and sparse vector multiplication algorithm on the GPU," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 841–847.

[21] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.

[22] R. Quinlan, "Data mining tools See5 and C5. 0," 2004.

[23] J. Li, G. Tan, M. Chen, and N. Sun, "SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 117–126.

[24] Y. Zhao, C. Liao, J. Li, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 94–108.

[25] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *ACM SIGPLAN Notices*, vol. 52, no. 8. ACM, 2017, pp. 235–248.

[26] F. Khorasani, R. Gupta, and L. N. Bhuyan, "Scalable Simd-efficient graph processing on GPUs," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 39–50.

[27] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 1–19.

[28] C. Hong, A. Sukumaran-Rajam, J. Kim, and P. Sadayappan, "Efficient graph processing on GPUs," in *PACT*, vol. 47, no. 8. ACM, 2017, pp. 117–128.

[29] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: vertex-centric graph processing on GPUs," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 2014, pp. 239–252.

[30] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC." USENIX, 2012.

[31] C. Hong, A. Sukumaran-Rajam, Kunal, S. Sabhlok, and P. Sadayappan, "Optimizing sparse-matrix sparse-vector multiplication for GPUs," The Ohio State University, Tech. Rep., 2018.

[32] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Notices*, vol. 47, no. 8. ACM, 2012, pp. 117–128.

[33] "Enterprise: breadth-first graph traversal on GPUs, author=Liu, Hang and Huang, H Howie, booktitle=High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for, pages=1–12, year=2015, organization=IEEE."

[34] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[35] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[36] X. Lu, T. Q. Phan, and S. Bressan, "Incremental algorithms for sampling dynamic graphs," in *International Conference on Database and Expert Systems Applications*. Springer, 2013, pp. 327–341.

[37] P. W. Olsen, A. G. Labouseur, and J.-H. Hwang, "Efficient top-k closeness centrality search," in *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 196–207.