Efficient Sparse-Matrix Multi-Vector Product on GPUs

Changwan Hong[1], Aravind Sukumaran-Rajam[1], Bortik Bandyopadhyay[1], Jinsung Kim[1], Süreyya Emre Kurt[1], Israt Nisa[1], Shivani Sabhlok[1], Ümit V. Çatalyürek[2], Srinivasan Parthasarathy[1], P. Sadayappan[1],
[1] The Ohio State University, USA, { hong.589, sukumaranrajam.1, bandyopadhyay.14, kim.4232, kurt.29, nisa.1, shivanisabhlok.1, parthasarathy.2, sadayappan.1 }@osu.edu
[2] Georgia Institute of Technology, USA, umit@gatech.edu

ABSTRACT

Sparse Matrix-Vector (SpMV) and Sparse Matrix-Multivector (SpMM) products are key kernels for computational science and data science. While GPUs offer significantly higher peak performance and memory bandwidth than multicore CPUs, achieving high performance on sparse computations on GPUs is very challenging. A tremendous amount of recent research has focused on various GPU implementations of the SpMV kernel. But the multi-vector SpMM kernel has received much less attention. In this paper, we present an in-depth analysis to contrast SpMV and SpMM, and develop a new sparse-matrix representation and computation approach suited to achieving high data-movement efficiency and effective GPU parallelization of SpMM. Experimental evaluation using the entire SuiteSparse matrix suite demonstrates significant performance improvement over existing SpMM implementations from vendor libraries.

CCS CONCEPTS

• **Computing methodologies** → *Shared memory algorithms*; • **Computer systems organization** → *Single instruction, multiple data*;

KEYWORDS

Sparse Matrix-Vector Multiplication, Sparse Matrix-Matrix Multiplication, Sparse Matrix Multi-Vector Multiplication, GPU

ACM Reference Format:
Changwan Hong[1], Aravind Sukumaran-Rajam[1], Bortik Bandyopadhyay[1], Jinsung Kim[1], Süreyya Emre Kurt[1], Israt Nisa[1], Shivani Sabhlok[1], Ümit V. Çatalyürek[2], Srinivasan Parthasarathy[1], P. Sadayappan[1], [1] The Ohio State University, USA, { hong.589, sukumaranrajam.1, bandyopadhyay.14, kim.4232, kurt.29, nisa.1, shivanisabhlok.1, parthasarathy.2, sadayappan.1 }@osu.edu [2] Georgia Institute of Technology, USA, umit@gatech.edu . 2018. Efficient Sparse-Matrix Multi-Vector Product on GPUs. In *HPDC '18: The 27th International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3208040.3208062

HPDC '18, June 11–15, 2018, Tempe, AZ, USA

ACM ISBN 978-1-4503-5785-2/18/06…$15.00
https://doi.org/10.1145/3208040.3208062

1 INTRODUCTION

Sparse Matrix Vector (SpMV) multiplication and Sparse Matrix Multi-vector (SpMM) multiplication (also called SpMDM - Sparse Matrix Dense Matrix multiplication) are key kernels used in a range of applications. SpMV computes the product of a sparse matrix and a (dense) vector. Although SpMM may be viewed as a set of independent SpMV operations on different dense vectors, and therefore be computed by repeatedly using an SpMV kernel, it is very beneficial in terms of achievable performance to implement a separate SpMM kernel, because of the significantly greater data reuse that can be exploited by doing so. However, as shown in the next section using *Roofline* [35] performance bounds, the achieved performance with the Nvidia cuSPARSE library SpMM implementation achieves significantly lower fraction of roofline limits when compared to the cuSPARSE SpMV implementation.

In this paper, we seek to answer the following question: *Is it feasible to achieve a high fraction of the roofline upper-bound for SpMM on GPUs, just as has already been accomplished by for SpMV [22, 25, 31] on GPUs?*

We undertake a systematic analysis of the SpMM problem and develop a new implementation that is demonstrated to be significantly faster than other state-of-the-art GPU implementations of SpMM – in cuSPARSE [1], MAGMA [3], and CUSP [13]. A key observation that drives the new implementation is that non-zeros in sparse matrices drawn from a variety of application domains are not uniformly randomly spread over the row/column index space, but exhibit non-uniform clustering of elements. We exploit this property to partition sparse-matrix elements into two groups: one group containing clustered segments and the other group holding the remaining scattered elements. Different processing strategies are used for the two partitions, with much higher data reuse and lower overheads achieved for the clustered partition.

2 BACKGROUND AND RELATED WORK

Graphics Processing Units (GPUs) are very attractive for sparse matrix computations due to their high memory bandwidth and computing power. However, achieving high performance is non-trivial due to the irregular data access pattern. A number of recent efforts have addressed the development of efficient SpMV for GPUs [4, 7, 12, 21, 22, 25, 31, 36, 39]. Like SpMV, SpMM is also a key kernel in sparse computations in computational science and machine-learning/data-science, but very few studies have so far focused on it [5, 6, 9, 18, 24, 26, 29, 37, 38]. SpMM is a core kernel for the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method [2, 3, 19], aerodynamic design optimization

[30], PageRank algorithm [5], sparse convolutional neural networks (CNNs) [16, 29], image segmentation in videos [32], atmospheric modeling [5], etc.

Roofline [35] performance upper-bounds based on global-memory bandwidth on GPUs can be developed for SpMV and SpMM as follows. Consider single-precision floating-point computation with a square $N \times N$ sparse matrix with nnz non-zero elements. Each element of the sparse matrix in standard CSR representation requires $8 \times nnz + 4 \times (N+1)$ bytes of storage. Including the storage for the input and output dense vectors, the total data footprint for SpMV is $8 \times nnz + 12 \times N + 4$ bytes. The total floating-point operation count is $2 \times nnz$, i.e., one multiply-add operation for each non-zero element in the sparse matrix. The input vector and sparse matrix reside in GPU global memory before execution of the SpMV kernel and the result vector is stored in global-memory after execution. Thus, a minimum volume of $8 \times nnz + 12 \times N + 4$ bytes of data must be moved between global memory and GPU registers across the memory. Dividing this data volume by the peak global-memory bandwidth BW_{GM} of the GPU (e.g., 732 Gbytes/sec. for an Nvidia Pascal P100 GPU) gives the minimum time for the global-memory data transfers, and the roofline upper bound performance of $\frac{2 \times nnz \times BW_{GM}}{8 \times nnz + 12 \times N + 4}$. Similarly, for SpMM, the total data footprint is $8 \times nnz + 8 \times K \times N + 4 \times N + 4$ bytes and the total floating-point operation count is $2 \times K \times nnz$, giving a roofline performance upper-bound of $\frac{2 \times K \times nnz \times BW_{GM}}{8 \times nnz + 8 \times K \times N + 4 \times N + 4}$ where K is the number of vectors (the width of dense matrices).

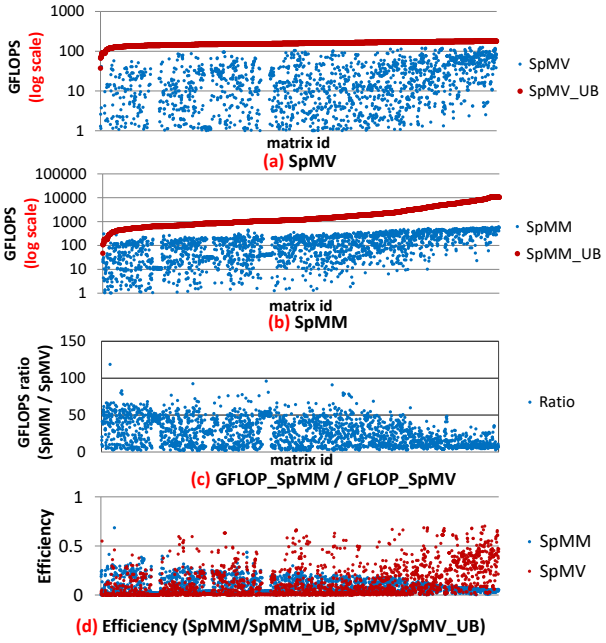


Figure 1: cuSPARSE SpMV/SpMM performance and upper-bound: Nvidia Pascal P100 GPU

Fig. 1 displays achieved SpMV and SpMM performance in GFLOPs by Nvidia's cuSPARSE library on a Pascal GP100 GPU, for the entire set of 2720 sparse matrices of the SuiteSparse [11, 14] collection (formerly known as the University of Florida Sparse Matrix collection). In these charts, data for the sparse matrices is plotted by sorting the matrices along the X-axis in increasing order of the number of non-zeros (nnz), with one point in the scatter-plot for each matrix. In Fig. 1(a), for each sparse matrix, a blue dot represents the achieved performance in GFLOPs and a corresponding red dot placed vertically above it marks the roofline performance upper-bound for that matrix. The performance upper-bound is around 170 GFLOPs (does not vary too much across matrices). cuSPARSE SpMV performance approaches the roofline bound for around 670 matrices.

Fig. 1(b) displays achieved performance and roofline upper-bounds for the same matrices with cuSPARSE SpMM. The achieved absolute performance can be seen to be much higher than SpMV, but the gap between actually achieved performance and the roofline upper-bound is also much higher. Fig. 1(c/d) present the same data as in Fig. 1(a/b), respectively, but expresses achieved performance as a fraction of the roofline upper-bound (efficiency). It may be seen that especially for large matrices cuSPARSE SpMM achieves a much lower fraction of the roofline bound than SpMV.

Algorithm 1: SpMV: Sparse Matrix-Vector Multiplication.

```
input  : CSR S[M][N], float D[N]
output : float O[M]
1 for i = 0 to S.rows-1 do
2     for j = S.rowptr[i] to S.rowptr[i+1]-1 do
3         O[i] += S.values[j] * D[S.colidx[j]]
```

Algorithm 2: SpMM: Sparse Matrix Multi-Vector Multiplication.

```
input  : CSR S[M][N], float D[N][K]
output : float O[M][K]
1 for i = 0 to S.rows-1 do
2     for j = S.rowptr[i] to S.rowptr[i+1]-1 do
3         for k = 0 to K-1 do
4             O[i][k] += S.values[j] * D[S.colidx[j]][k]
```

Pseudocodes for sequential SpMV and SpMM are shown in Alg. 1 and Alg. 2, respectively. The sparse matrix S is stored in the standard CSR (Compressed Sparse Row) format. The nonzero elements are compacted and stored in S.values[:], with all non-zeros in a row being placed contiguously. S.rowptr[i] points to the first nonzero element from row i in S.values[:]. S.colidx[i] holds the column index of the corresponding nonzero located in S.values[i]. Fig. 2(b) shows an example of a sparse matrix stored in the CSR format. The DCSR (Doubly Compressed Sparse Row) format [8] further compresses CSR by only storing non-empty rows. As seen in Fig. 2(c), indices of non-empty rows are placed in S.rowidx[:] and S.rowptr[i] points to the first nonzero elements for the row S.rowidx[i].

SpMV (Alg. 1) iterates over the rows of S, forming the sparse dot-product of the ith row of S with the input dense vector D to produce the ith element of the output dense vector O.

SpMM (Alg. 2) iterates over the rows of S, forming the sparse dot-product of the ith row of S with the kth dense vector, $D[][k]$, to produce the ith element of the kth output dense vector, $O[i][k]$.

Aktulga et al. [2] describe an SpMM scheme based on the compressed sparse blocks (CSB) format, optimized for both transposed

and non-transposed SpMM ($O = SD$ and $O = S^T D$). The elements of the sparse matrix are partitioned into blocks of size $\beta \times \beta$. Each block is represented in coordinate format (COO). For non-transposed SpMM ($O = SD$), threads process row blocks of size $((num\_threads \times \beta) \times N)$. For transposed SpMM ($O = S^T D$), threads process column blocks of size $(N \times (num\_threads \times \beta))$.

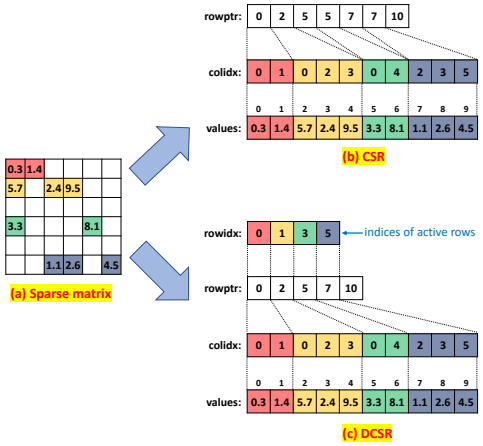


Figure 2: CSR and DCSR formats.

Anzt et al. [3] developed an SpMM scheme optimized for GPUs, based on the SELL-P format. The SELL-P format is built by partitioning rows of the sparse matrix into blocks, and then each row block is converted into ELLPACK format, with the rows being padded so that the row length of each block is a multiple of the number of threads. The threads are organized into 3D blocks, where the x-dimension maps to a row within a SELL-P block, the y-dimension maps to columns, and the z-dimension maps to multiple vectors. Since multiple threads are assigned to process the elements of the same row, reduction operations are required and performed in shared memory.

FastSpMM [28, 34] uses ELLPACK-R [33] to enhance performance by storing the sparse matrix in a regular data structure. However, this strategy may suffer when processing very irregular sparse matrices. cuSPARSE [1] from Nvidia also supports SpMM. It offers two modes i) T: Transposed and ii) NT: Non transposed. In BidMach [10], SpMM is implemented as one of the many kernels and it internally uses the cuSPARSE format.

3 RS-SPMM

In this section, we present the proposed GPU SpMM algorithm, labeled RS-SpMM for *Row-Segmented SpMM*. The name derives from the fact that the sparse matrix is partitioned into two parts, one holding clustered nonzero row-segments, enabling higher data reuse in the GPU and thus lower data movement from global-memory than previously developed SpMM approaches.

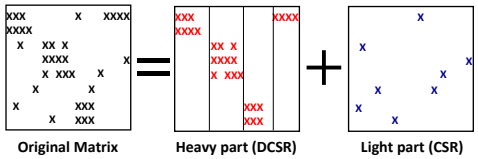


Figure 3: Splitting sparse matrix into heavily clustered row-segments and remainder.

Fig. 3 illustrates the splitting of a sparse matrix into two matrices, one holding nonzeros in heavily clustered row-segments, and the other holding the remaining nonzeros that are randomly scattered over the column-index space in each row. The rationale for this splitting is elaborated below, and is based on the observation that large sparse matrices found in practice generally do not exhibit fully random distribution of their nonzeros in the row/column space. A sizable fraction of nonzeros tend to be grouped in clusters in the row/column index space.

In SpMM, a sparse matrix is multiplied with a dense matrix to produce a dense matrix. SpMV can be seen as a special case of SpMM where the width of the dense input matrix is one. In the rest of the section, we refer to the input sparse matrix as **S** ($M \times N$), the input dense matrix as **D** ($N \times K$), and the output dense matrix as **O** ($M \times K$).

When compared to SpMV, SpMM has the following significant differences: i) Unlike SpMV, with SpMM, the sparse matrix elements have a high reuse factor of K; the reuse factor of the input/output dense matrix elements is similar to that for the input/output vector with SpMV; ii) Unlike SpMV, it is feasible to achieve coalesced access of the dense input/output matrices (due to width K).

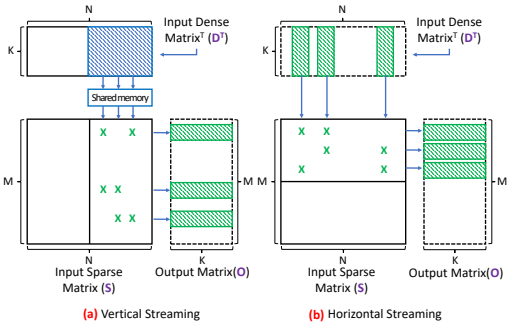


Figure 4: Vertical and horizontal streaming.

Here, we discuss two options for SpMM: *vertical streaming* and *horizontal streaming* (Fig. 4). As with dense matrix multiplication, tiling is also crucial to optimizing performance of SpMM. In general, tiling of all the three loops $\{i, j, k\}$ is feasible in Alg. 2. However, the nature of reuses for the three arrays $\{D, S, O\}$ in SpMM is along distinct loops: the i loop for D, the j loop for O, and the k loop for S. Since the data footprint of an array is invariant with respect to iterations of the "reuse" loop index, it means that one of the three arrays will have an invariant data footprint and therefore achieve complete reuse as the innermost tile is changed in a 3D tiled execution. This maximal reuse is independent of the chosen tile size

and therefore it is best to minimize this tile size, thereby enabling maximization of the other two tile sizes to achieve as much reuse as possible for the other two arrays. This observation leads to what is referred to as *streamed* execution along that innermost tile dimension, which is equivalent to just performing 2D tiling over two out of the three loops in Alg. 2. Of the three arrays in SpMM, S has a reuse factor of K, while D and O have reuse factors corresponding to the average number of non-zeros in S along a column/row, respectively. Typically K is much larger than the average number of non-zeros in a row/column. Further, achieving reuse along k for each element is S is much easier since the dense matrix elements in D and O are contiguously located in global memory. Therefore, it is best to use i or j as the streaming direction for SpMM. Streaming along i is referred to as *vertical streaming* and streaming along j is called *horizontal streaming*.

For vertical streaming (Fig. 4(a)), the input sparse matrix (**S**) is partitioned into column panels - sets of contiguous columns of the sparse input matrix. Each column panel is processed by a thread block. A thread block collectively brings the **D** elements corresponding to the column panel into shared-memory. Different warps in a thread block process different rows within the column panel. The threads of a warp are mapped across "K", that is, each thread is responsible for computing a partial result contribution for a single output element in the result dense matrix. All threads in a warp compute the product of the same non-zero element of the sparse matrix with a distinct input matrix element, and then they move to the next sparse matrix element in the row. Threads accumulate partial results in thread local registers, and at the end of processing of each row-segment in a column-panel, the partial contribution are moved from registers to global-memory corresponding to the output dense matrix, using atomic operations. Even though this scheme achieves coalesced access on input and output (with sufficiently large K), the downside is that it requires atomic operations for accumulating partial results to global memory. For the rest of this section, we refer to this scheme as the vertical scheme.

In horizontal streaming (Fig. 4(b)), the input sparse matrix (**S**) is partitioned into row panels. Each row panel is processed by a thread block. Threads cyclically processes elements along the horizontal dimension of **S**. Each row is processed by a warp and different threads in the warp are distributed across "K". Each thread accumulates the partial results in thread local registers, and at the end of each row the threads move the partial contribution from registers to global-memory corresponding to the output dense matrix using atomic operations. For the rest of this section, this scheme is referred to as the horizontal scheme.

In both the vertical and horizontal SPMM schemes, the threads in a warp are spread across K. This helps in i) avoiding intra-warp communication that would otherwise be required for reduction of partial products across the threads in the same warp, and ii) avoiding the need for shared memory for holding the intermediate results. If the threads in a warp are distributed across a row of the sparse matrix, the amount of work for different threads can vary widely, leading to intra-warp load imbalance. The downside of distributing the threads in a warp across the columns of the input matrix is that, if the number of columns of the input matrix (K) is not a multiple of 32, the last column slice will not be load balanced. However, in practice, the last scheme works better as the load imbalance across the non-zero elements in a row-segment of a column partition is much worse than the load imbalance across the rows of the input dense matrix.

The vertical scheme is not beneficial if the average number of elements in a row of a column panel is low. With the vertical scheme, at the end of processing each row in the column panel, there is an expensive atomic update to global memory. If the number of elements per row-segment is small, the relative overhead of the atomic operations is more prominent. However, if the average number of elements in a row-segment in a column panel is high, then the vertical scheme is beneficial as it gets full reuse of elements in **D**. The disadvantage of the horizontal scheme is that there is no reuse of elements in **D** (except possibly from cache).

The above observations motivate a dual scheme: rows with sufficiently high non-zero count in a column panel are processed using vertical streaming and the rest are processed using horizontal streaming. In the rest of the section, rows whose non-zero count within a column-panel is greater than a parametric threshold are called **heavy rows** and the others are called **light rows**.

3.1 Data Structure

We use a CSR representation for the light rows and a DCSR representation for the heavy rows. All the heavy row-segments in each column panel are represented by a DCSR structure. All the light rows in the entire sparse matrix **S** are represented by a CSR matrix. Fig. 5 illustrates the hybrid data representation. For the illustration, the threshold for a row to be classified as heavy is two; column partition width is 4. Blocks with yellow background are the heavy rows in column partition 0. Similarly, blocks with green background are the heavy rows in column partition 1. The blocks with white background represent the light rows. Details of the representation of the heavy and light blocks are shown in Fig. 5 (b-3,b-4) and Fig. 5 (c-2), respectively.

The number of columns in **S** that are processed by a thread block (W) is chosen such that $W \times k \times sizeof(data\_type) \times no\_of\_active\_tb\_per\_sm$ is equal to the shared-memory capacity. In our experiments, "k" (k columns of input and output are processed by a thread block) is chosen as 64 and no_of_active _tb_per_sm is chosen as 2. "k" should be a multiple of 32 for coalesced access (to avoid thread divergence). Choosing no_of_active _tb_per_sm as 2 helps in achieving maximum occupancy.

3.2 Algorithm

The pseudocode for the RS-SpMM scheme for heavy rows (vertical) is shown in Listing 1. Rows of **D** of size K are divided into data-tiles or slices of size k. Each column panel is processed by K/k thread blocks. All threads in a thread block collectively bring a slice of the input dense matrix corresponding to the column panel (**D**) to shared-memory (line 5-7). The columns of **D** that are brought to shared-memory depend on the slice id along K.

Each warp then processes the rows in the column panel in a cyclic fashion (line 9-20). Each thread initializes the partial result (which is held in a thread-local register) to 0 (line 10). All threads in a warp process the same non-zero element in the sparse matrix. Reading each sparse element one by one will result in uncoalesced access. In order to avoid this, the threads read 32 non-zero elements (line 15 to

18) and then use warp shuffles to exchange elements (line 19). In the first iteration, all threads in a warp need index_matrix[start] and value_matrix[start], and these values are stored in the first lane (tid % WARP_SIZE == 0). Hence, *value* and *index* held by the first lane are broadcast to other threads. At the second iteration, all threads in a warp need index_matrix[start+1] and value_matrix[start+1], which are stored in the second lane, Hence, values of the second lane are broadcast.

Each thread then computes the partial product in thread-local register (line 19). Finally, the accumulated partial results are updated to global memory using atomic operations (line 21 - 23). Atomic operations are required as different thread blocks could concurrently update the same element.

The RS-SpMM algorithm for light rows (horizontal scheme) is shown in Listing 2. Each row of the light matrix is processed by a warp. In order to minimize inter-warp load imbalance, we chose the smallest thread block size (64 on Pascal) that maintains full occupancy. For example, on an Nvidia Pascal GPU, since one thread block has only two warps, row 0 and 1 are processed by the first block, row 2 and 3 are processed by the second block, and so on. Each thread initially computes the row and slice along K that it should process. Each thread initializes the partial result (held in a thread local register) to 0 (line 6). Similar to the SpMM heavy scheme, the column indices and values are collectively read by all threads in the warp (line 10, 11) and broadcast to all threads in a warp (line 13). The partial products are then computed and accumulated in the thread local register (line 13). Finally, the partial result is accumulated to global memory using atomic operations (line 15). Atomic operations are still needed since the thread blocks for the heavy scheme and light scheme are launched concurrently.

Listing 1: SPMM Pseudocode (Heavy row segments)

```
1. row_offset = tb_idx * IN_TILE_ROW_SIZE;
2. slice_offset = tb_idy * IN_TILE_SLICE_SIZE;
3. warp_id = tid/WARP_SIZE;
4. lane_id = tid%WARP_SIZE;
5. for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
6.   sm_input_value[i][lane_id] =
     input_value[row_offset+i][slice_offset+lane_id];
7. end
8. __syncthreads;
9. for i=seg_start_num[tb_idx] to seg_start_num[tb_idx+1]-1 step
        tb.size()/WARP_SIZE do
10.  val = 0;
11.  start = start_seg_position[i];
12.  end = start_seg_position[i+1];
13.  for j=start to end-1 do
14.    mod = (j - start)%WARP_SIZE
15.    if mod == 0 then
16.      index_buf = seg_index[j + lane_id];
17.      value_buf = seg_value[j + lane_id];
18.    end
19.    val += sm_input_value[__shfl(index_buf, mod)][lane_id]
             * __shfl(value_buf, mod);
20.  end
21.  row_idx = seg_row_position[i];
22.  //directly accumulate results in global memory
23.  atomicAdd(&dest_value[row_idx][slice_offset+lane_id], val);
24. end
```

Listing 2: SPMM Pseudocode (Light rows)

```
1.  row_offset = (tb_idx*tb.size() + tid) / WARP_SIZE;
2.  slice_offset = tb_idy * IN_TILE_COL_SIZE;
3.  lane_id = tid%WARP_SIZE;
4.  start = csr_row_pointer[row_offset];
5.  end = csr_row_pointer[row_offset+1];
6.  val = 0;
7.  for i=start to end-1 do
8.    mod = (i - start)%WARP_SIZE
9.    if mod == 0 then
10.     index_buf = csr_column_idx[i + lane_id];
11.     value_buf = csr_column_val[i + lane_id];
12.   end
13.   val += input_value[__shfl(index_buf, mod)][lane_id]
            * __shfl(value_buf, mod);
14. end
    //directly accumulate results in global memory
15. atomicAdd(&dest_value[row_offset][slice_offset+lane_id], val);
```

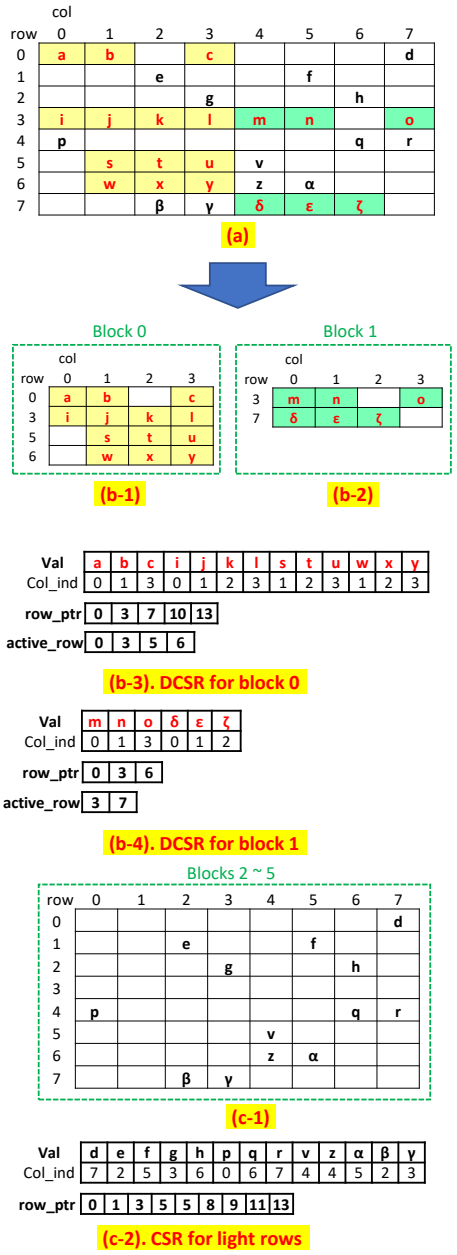


Figure 5: SpMM overview

Thread coarsening is used for both the vertical and horizontal schemes to improve performance. Coarsening is done along "K" and i) enhances ILP (instruction level parallelism) to achieve better overlap to mask global memory latency, and ii) reduces the number of warp shuffle operation needed. A thread coarsening factor of two causes each thread to process an element from two adjacent slices.

In the rest of the paper, we refer to the thread coarsening factor for the vertical and horizontal schemes as CF_V and CF_H, respectively. Listing 3 shows pseudocode corresponding to thread coarsening factor CF_V of 2. As seen in line 6, 7 in Listing 3, since there is no dependence between two global-memory load instructions, they can be loaded concurrently, which helps in tolerating memory access latency. The number of warp-shuffle operations (line 20, 21) processed by a thread block remains the same and thread coarsening reduces the number of thread block launched. Hence, the warp-shuffle operations are halved.

Listing 3: SPMM Pseudocode (Heavy rows, $CF_V = 2$)

```
1. row_offset = tb_idx * IN_TILE_ROW_SIZE;
2. slice_offset = tb_idy * IN_TILE_SLICE_SIZE * 2;
3. warp_id = tid/WARP_SIZE;
4. lane_id = tid%WARP_SIZE;
5. for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
6.   sm_input_value[i][lane_id] =
     input_value[row_offset+i][slice_offset+lane_id];
7.   sm_input_value[i][lane_id+WARP_SIZE] =
     input_value[row_offset+i][slice_offset+lane_id+WARP_SIZE];
7. end
8. __syncthreads;
9. for i=seg_start_num[tb_idx] to seg_start_num[tb_idx+1]-1 step
       tb.size()/WARP_SIZE do
10.  val1 = 0;
11.  val2 = 0;
12.  start = start_seg_position[i];
13.  end = start_seg_position[i+1];
14.  for j=start to end-1 do
15.    mod = (j - start)%WARP_SIZE
16.    if mod == 0 then
17.      index_buf = seg_index[j + lane_id];
18.      value_buf = seg_value[j + lane_id];
19.    end
20.    shfl_index = __shfl(index_buf, mod);
21.    shfl_value = __shfl(value_buf, mod);
22.    val1 += sm_input_value[shfl_index][lane_id] * shfl_value;
23.    val2 += sm_input_value[shfl_index][lane_id+WARP_SIZE] *
       shfl_value;
24.  end
25.  row_idx = seg_row_position[i];
26.  //directly accumulate results in global memory
27.  atomicAdd(&dest_value[row_idx][slice_offset+lane_id], val);
28.  atomicAdd(&dest_value[row_idx][slice_offset+lane_id+WARP_SIZE],
     val2);
29. end
```

4 MODELING IMPACT OF SLICE-SIZE CHOICE

In this section, we describe how we determine the threshold for classifying a row as light or heavy and choosing CF_V and CF_H. The process we describe below for choice of parameters need only be done once for a target platform, at library installation time. The factors were chosen based on a subset of matrices (training set) from SparseSuite [11]. All matrices having fewer than 100,000 non-zeros were removed. The remaining matrices were sorted in increasing order of number of non-zeros and every 20th matrix was included in the training set. In total, 43 matrices (5%) were selected.

Figure 6 presents an overview of the approach.

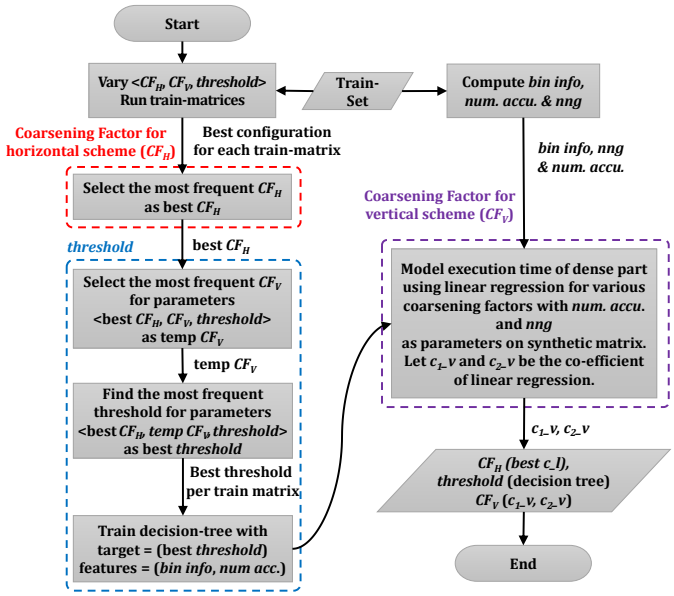


Figure 6: Modeling overview.

4.1 Coarsening factor for horizontal scheme (CF_H)

For the horizontal scheme, the coarsening factor along K was determined empirically, based on the training set. We ran each training matrix with various CF_H and CF_V and thresholds to identify the best performing configuration per matrix. From the best performing configuration per matrix, the most frequent CF_H was chosen as the CF_H (c_l). We observed that using a coarsening factor of 1 (no coarsening) achieves the best performance for most matrices for double precision. Hence, no thread-coarsening was chosen for double precision. For single precision, we observed that a coarsening factor of 2 achieves the best performance in most cases. Based on NVPROF performance metrics, the difference between the best coarsening factor for single and double precision is due to the fact that the relative warp shuffle cost for single precision is higher than that for double precision.

4.2 Threshold for classifying row as light or heavy

The threshold used to classify a row as light or heavy was computed using a decision tree. To train the decision tree, we chose clustering information and number of accumulations as the input features and the best threshold as the target. In order to model clustering, for each matrix, we calculated the distance between every two adjacent elements and we assigned it to a bin based on the distance. The bins were organized as powers of 4 (1, 1-3, 4-15, 16-63, . . ., 262,144-1,048,576).

The best threshold was found in two phases. In the first phase, we fixed CF_H and varied CF_V (1, 2, 4) and threshold (1, 2, 3, . . ., 16). We

evaluated the training set over these configurations to determine the best parameters for each matrix. The most frequent vertical coarsening factor among the best parameters was then identified. In the second phase, CF_V and CF_H are fixed on the basis of the previous experiments and the corresponding best threshold per matrix was identified.

The decision tree (Weka 3.8 [15] "J48 -C 0.25 -M 2" without any filtering) was then trained with the bin counts (distance between adjacent elements and number of accumulations as input feature and the best threshold computed in phase 2 as the output feature. In order to determine the threshold of test set we used this trained decision tree.

4.3 Coarsening factor for vertical scheme (CF_V)

Increasing the coarsening factor along K for the vertical scheme helps to hide memory latency and reduce shuffle overhead. However, increasing the coarsening factor increases the amount of shared memory required for storing the dense matrix elements, and this decreases the column panel size. When the size of column panel is decreased, the number of rows which are classified as heavy rows decrease, and this can adversely affect performance. In order to determine the coarsening factor, for the vertical scheme, we estimated the execution time for each coarsening factor and then selected the coarsening factor that minimized execution time. The execution time is estimated as:

$$\text{exec\_time} = C_{1,v} \times acc_{dense} + nnz_d \times C_{2,v} + \text{exec\_time\_sparse}$$

where $C_{,v}$ is the cost when CF_V is v, and acc_{dense} is the number of accumulations in the dense part. Since all the elements are processed by the heavy scheme, the execution time of light scheme (exec_time_sparse) is zero. $C_{1,v}$ and $C_{2,v}$ are constants which depend on v.

In order to determine $C_{1,v}$ and $C_{2,v}$ we used two synthetic matrices i) 4K X 4K fully dense matrix and ii) 4K X 4K sparse matrix. Both matrices were processed only by the vertical scheme. The sparse matrix was designed such that for every row in a column panel of size 256, the number of non-zero elements equals the threshold. The exact column id is chosen at random.

Assuming that Num_Col_Panel = 4K / 256, the execution time for the two synthetic matrices, for a coarsening factor of 1, can be represented as

$$\begin{aligned} T_{1,1} &= C_{1,1} \times 4K \times Num\_Col\_Panel + C_{2,1} \times 4K \times 4K \\ T_{2,1} &= C_{1,1} \times 4K \times Num\_Col\_Panel + \\ &\quad C_{2,1} \times 4K \times Num\_Col\_Panel \times THRESHOLD \end{aligned}$$

The execution time of $T_{1,1}$ and $T_{2,1}$ was determined by executing these synthetic matrices using the vertical scheme (with coarsening factor of 1) and the above equations were solved to find $C_{1,1}$ and $C_{2,1}$. Similarly, the other $C_{*,*}$ values were computed.

In order to find CF_V for a given matrix, the execution time was estimated by applying the above equations using the $C_{*,*}$ values. The coarsening factor was then selected as the one with the lowest predicted execution time.

4.4 Effectiveness of Model

Fig. 7 presents performance of RS-SpMM for all SparseSuite matrices, for single precision with $K = 128$. We present the performance of RS-SpMM without any modeling, with the best parameters (separately selected for each individual case), and with our modeling. As seen in the figure, the modeling matches or closely follows the performance of the best parameter case.

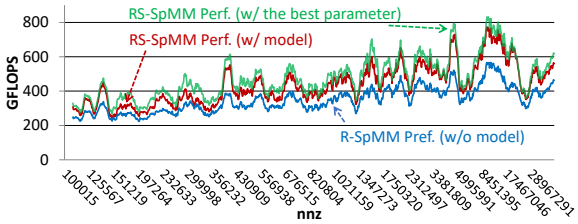


Figure 7: Modeling effect (WIDTH=128, single precision).

5 EXPERIMENTAL EVALUATION

This section details the experimental evaluation of the RS-SpMM scheme. The experiments were performed on an Nvidia Pascal P100 GPU. In all experiments, ECC was turned off and we used the NVCC compiler optimization flag -O3 with NVCC 8.0 and GCC 4.9.2. For all the schemes, we only include the kernel execution time; preprocessing time and data transfer time from CPU to GPU are not included (we document preprocessing time separately). All tests were run 5 times and average numbers are reported.

5.1 Performance of RS-SPMM

Fig. 8 (a) shows measured performance of RS-SpMM for single-precision computation, with four values of K (the width of the dense matrices O and D): 8, 32, 128, 512. Use-cases for SpMM vary depending on application. For applications in computational science, such as LOBPCG, widths in the tens, and below 100 are typical. For machine learning applications, the width K often corresponds to the number of latent features in models, and values of K around 100 are common, with interest in going higher to several hundreds or thousand. Hence, we evaluate RS-SpMM using four values in the range from 8 to 512. The performance of RS-SpMM improves as we increase dense matrix width from 8 to 32 and 128, tending to saturate at that point - the performance curves for K=128 and K=512 are nearly indistinguishable. For large matrices, performance is around 200 GFLOPs for K=8, approximately doubling to around 400 GFLOPs for K=32, and further increasing to almost 800 GFLOPs for several matrices for K=128/512.

Fig. 8 (b-e) show relative performance improvement over Nvidia's cuSPARSE library implementation of SpMM. RS-SpMM uses the standard C row-major representation for the input/output dense matrices, while cuSPARSE uses the FORTRAN column-major convention for the dense matrices. cuSPARSE also provides two variants: $O = SD$ (denoted cuSPARSE(NT)) and $O = SD^T$ (denoted cusPARSE(T)). As can be seen below, performance of cuSPARSE for $O = SD^T$ (transposed product) is often much higher than performance for $O = SD$ (non-transposed product). A few applications require SpMM products with both transposed and non-transposed

forms of one of the input matrices, while most applications only require one of them. For applications not requiring both transposed and non-transposed products, an application user has the choice of using the higher performing variant through storing the dense matrix in normal or transposed form. Hence, in this section we compare RS-SpMM performance with both the non-transposed ($O = SD$) and transposed ($O = SD^T$) variants of cuSPARSE SpMM.

The speedup of RS-SPMM(NT) vs. cuSPARSE(NT and T) is shown for different values of K; Since cuSPARSE(NT) tends to achieve much lower performance than cuSPARSE(T), higher speedup is achieved over it. RS-SpMM speedup is generally in the range between 1 and 2 over cuSPARSE(T), and around 4x over cuSPARSE(NT). The speedup over cuSPARSE tends to be higher for larger K.

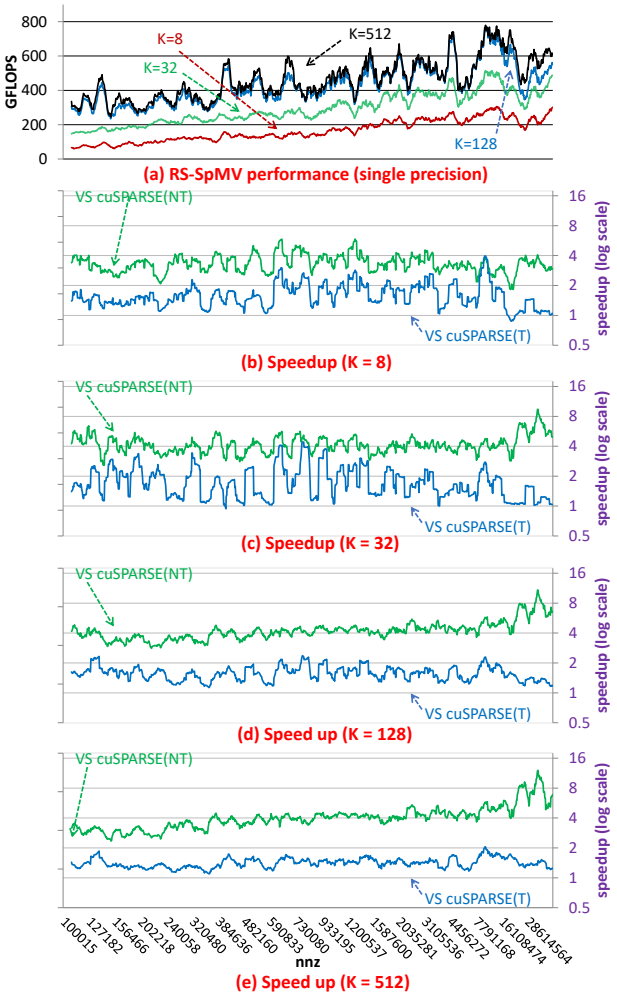


Figure 8: Performance comparison: RS-SpMM (NT) vs CuSPARSE (NT and T); Single Precision

Fig. 9 (a-e) present performance of RS-SpMM for double-precision. Overall performance in GFLOPs for double-precision is slightly lower than for for single-precision, but well over 0.5x, compared to single-precision. RS-SpMM is quite consistently faster than cuSPARSE(T and NT).

In contrast to single precision, the cuSPARSE(T) version is not consistently faster – quite often the speedup over cuSPARSE(T) is higher than the speedup over cuSPARSE(NT). For sparse matrices exhibiting high variance in row lengths, performance of cuSPARSE(NT) and cuSPARSE(T) degrades considerably because of load imbalance. On the other hand, performance of RS-SpMM is less degraded since rows having large nnz are normally classified as heavy rows and processed by multiple thread blocks. Hence, there is a fluctuation of achieved speedup with different sparse matrices in Fig. 8 and 9. We note that in Fig. 8 and 9, only the kernel execution time is reported; the pre-processing overhead to create the needed representation for RS-SpMM is discussed later.

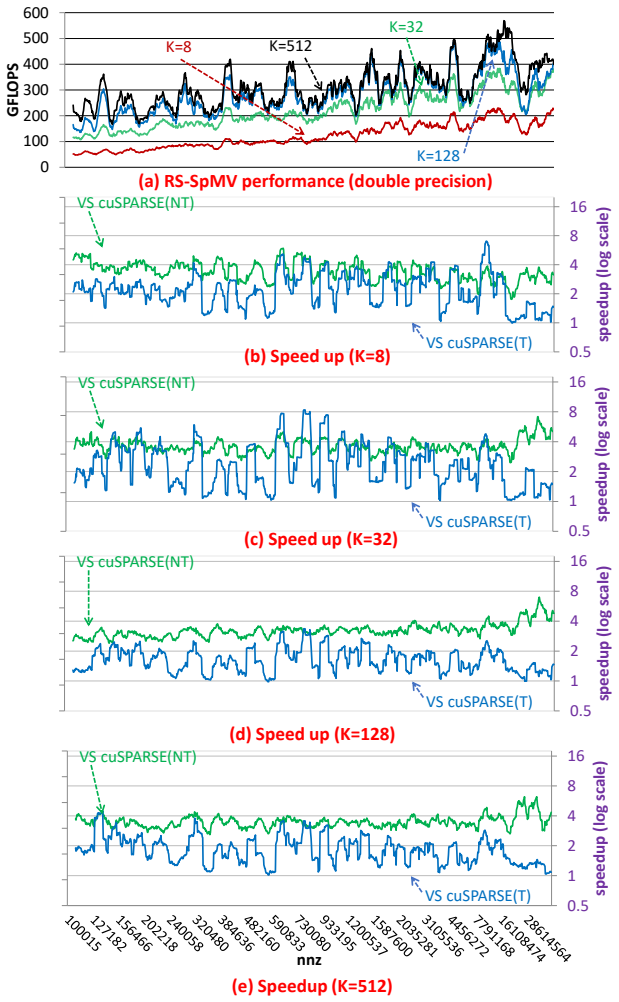


Figure 9: Performance comparison: RS-SpMM (NT) vs CuSPARSE (NT and T); Double Precision

In Fig. 10, we present a performance profile comparing RS-SpMM with cuSPARSE SpMM (both (T) and (NT) variants). In each row, the different charts are for varying K (8,32,128,512). For each matrix, the best performing version among RS-SpMM and cuSPARSE SpMM variants is used as a normalizer to compute performance loss of the

other instances. For each SpMM implementation, the cumulative curve shows the fraction of cases for which the slowdown with respect to the best performer is less than the X-axis value. Thus, a point (x,y) implies that a fraction y of all matrices achieved a slowdown less than x relative to the fastest implementation. Better performing variants have curves that rise rapidly to hit y=1.0, while relatively poor performance results in a shallow curve that may not get to y=1.0 even for the largest slowdown value in the range of the plot. The top (red) curve shows that RS-SpMM quite significantly outperforms the cuSPARSE variants. We note that for $K = 8$ and $K = 32$, we also compared RS-SpMM with CUSP [13] SpMM. The Nvidia CUSP library is an open-source library for linear algebra and graph computations on GPUs. CUSP implements SpMM but we find its results are only correct for dense matrix widths of 2,4,8,16, and 32 and incorrect for other widths; further even for the few correct cases, CUSP-SpMM performance is consistently lower than cuSPARSE(T) performance, as seen in Fig. 10 (a,b,e,f).

The first two rows of charts document performance over all SuiteSparse matrices with more than 100K nonzeros, for single and double precision. The third and fourth rows show performance for an iterative execution scenario that is common with applications like block Krylov solvers like GMRES. Here, the output dense matrix O from a previous iteration is modified by scaling or other simple point-wise operations and then becomes the input matrix D for the following iteration. With the cuSPARSE(T) form, an explicit transpose will be required. The time for cuBLAS [27] dense matrix transpose is added to cuSPARSE(T) for this scenario. Experimental results are only presented for square matrices from SuiteSparse, since this is only applicable to square matrices. Finally, the last row shows profile for the $S^T D$ computation and the case where both SD and $S^T D$ are required. In this case, performance of RS-SpMM is considerably higher than cuSPARSE.

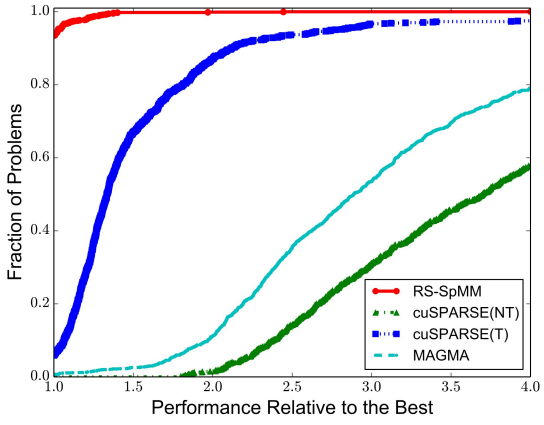


Figure 11: Performance of RS-SpMM compared with MAGMA and cuSPARSE; K=128, single precision

The MAGMA library [3] has high-performance implementations for dense and sparse linear algebra functions for GPUs. We attempted a performance comparison of RS-SpMM with MAGMA, in addition to cuSPARSE. However, we were unable to successfully process all the matrices from SuiteSparse with MAGMA; several matrices resulted in MAGMA error messages. However, we were able to run a significant subset of matrices successfully with MAGMA's SpMM. In Fig. 11, we present a profile comparison on just the successful subset with MAGMA, for single-precision and K=128. It may be seen from this profile that cuSPARSE(T) is faster than MAGMA and RS-SpMM is faster than cuSPARSE(T).

bhSPARSE [23] and Merge-based CSR [25] provide very high-performance GPU implementations of SpMV, but do not provide an SpMM implementation. SpMM can be implemented as a "loop over SpMV", with an outer loop over the width of the dense matrix. Fig. 12 (a-d) show measured performance using such a loop-over-SpMV approach, with bhSPARSE, Merge-based CSR and cuSPARSE-SpMV for single-precision computation, with four values of K: 8, 32, 128, 512. For K values beyond 8, RS-SpMM is significantly faster than loop-over-SpMV (bhSPARSE, Merge-based CSR and CuSPARSE-SpMV) because of significantly higher data reuse achieved by SpMM primitives (the maximum value of X-axis in Fig. 12 is 16). Similar trends can be seen for double precision as shown in Fig. 12 (e-h).

5.2 Pre-processing overhead

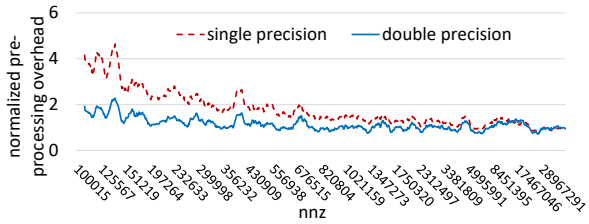


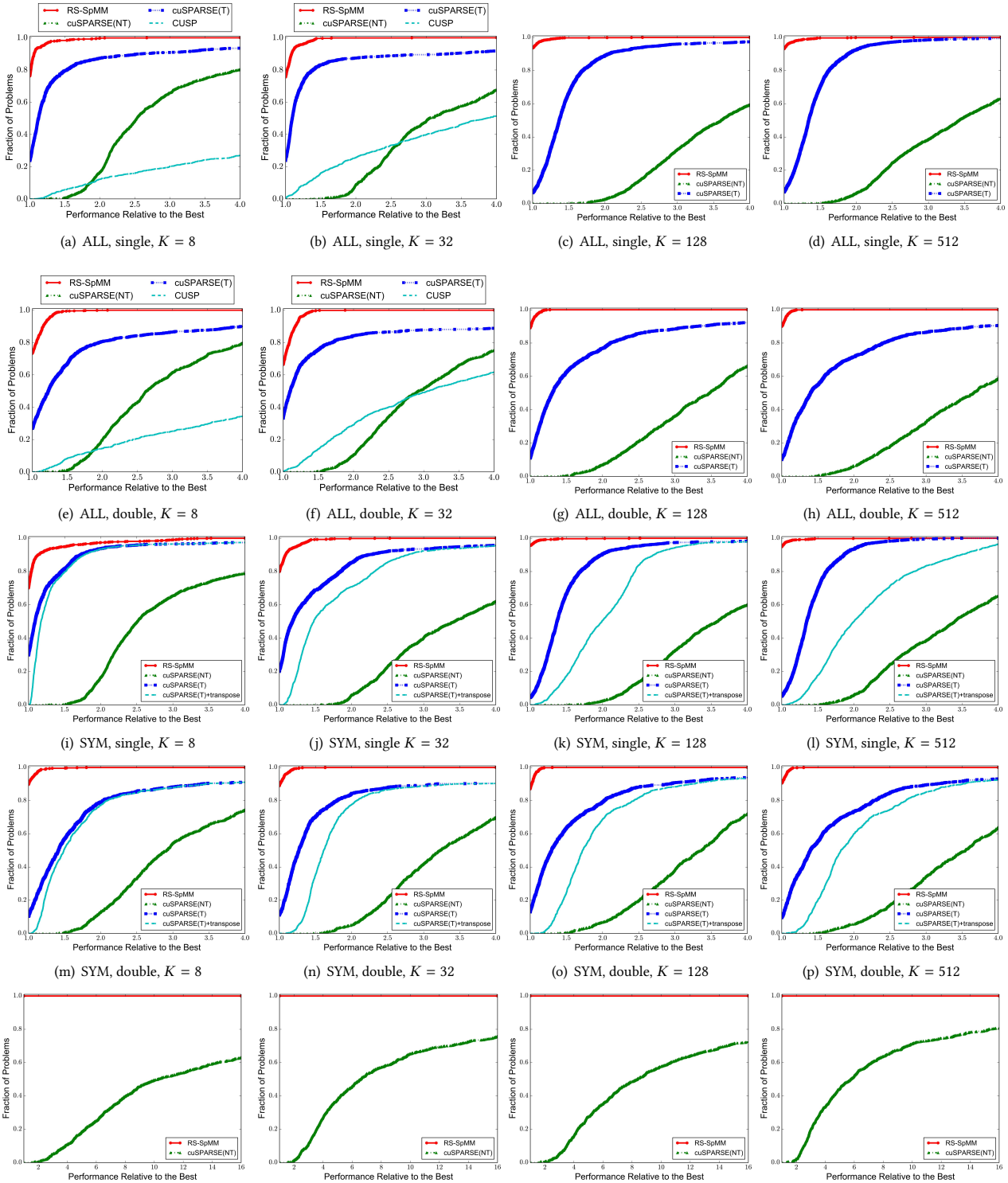
Figure 13: Preprocessing overhead

Constructing the data structures required for RS-SPMM scheme incurs an additional overhead. Fig. 13 shows the preprocessing overhead normalized to one iteration of RS-SpMM. Note that typical applications involving SpMM can execute a large number of iterations such as [6, 29]. For example, with sparse convolutional neural networks in inference mode, even though the input dense matrix changes (holding the new data to be processed), the structure of the sparse coefficient matrix remains unchanged. Hence, the preprocessing overhead is relatively insignificant.

Pre-processing is done on the GPU - converting from standard CSR structure (column indices within each row assumed sorted) to the structure which splits out heavy row-segments (in DCSR) and the remainder (standard CSR format). For threshold T, heavy row-segments are extracted by scanning over rows and checking if col_ptr[i] and col_ptr[i+T] belong to the same partition in the same row; if so the elements between index i to i+T get included in a heavy row. Fig. 13 shows our Pre-processing cost normalized to one SpMM iteration with K=128.

6 SPMM FOR $O = S^T D$

Some applications require both original and transposed sparse matrices to be multiplied with the dense matrix ($O = S \times D$ and $O = S^T \times D$) [2, 3]. Explicitly transposing the sparse matrix just for



(a) ALL, single, $K = 8$ (b) ALL, single, $K = 32$ (c) ALL, single, $K = 128$ (d) ALL, single, $K = 512$

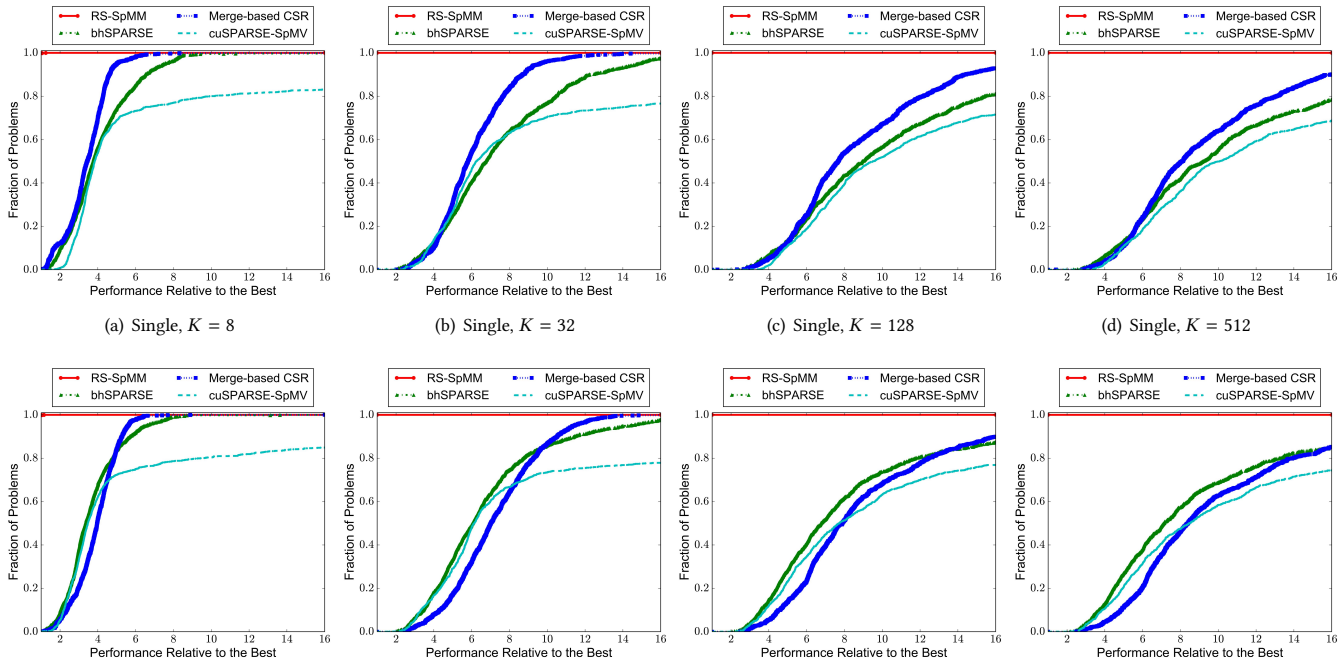
(e) ALL, double, $K = 8$ (f) ALL, double, $K = 32$ (g) ALL, double, $K = 128$ (h) ALL, double, $K = 512$

(i) SYM, single, $K = 8$ (j) SYM, single $K = 32$ (k) SYM, single, $K = 128$ (l) SYM, single, $K = 512$

(m) SYM, double, $K = 8$ (n) SYM, double, $K = 32$ (o) SYM, double, $K = 128$ (p) SYM, double, $K = 512$

(q) $O = S^T \times D$, single (r) $O = S^T D$ & $O = SD$, single (s) $O = S^T D$, double (t) $O = S^T D$ & $O = SD$, double

Figure 10: Performance Profiles. (a)-(h) all matrices (ALL) for single and double precision with varying K, (i)-(p) symmetric (SYM) matrices for single and double precision with varying K, (q)-(t) $O = S^T D$ & $O = SD$ for single and double precision.



(a) Single, $K = 8$ (b) Single, $K = 32$ (c) Single, $K = 128$ (d) Single, $K = 512$

(e) Double, $K = 8$ (f) Double, $K = 32$ (g) Double, $K = 128$ (h) Double, $K = 512$

Figure 12: Performance Profiles: RS-SpMM and Loop-over-SpMV; Single and Double; K=8,32,128,512.

SpMM can be expensive. The RS-SpMM scheme has been adapted to perform $O = S^T \times D$ without explicit transpose and is called RS-SpMMT. RS-SpMM and RS-SpMMT use the same data structure. Thus the row-segments of non-transposed version corresponds to the column segments of transposed version.

Pseudocodes for heavy row segments and light row segments are shown in Listing 4 and 5 respectively. In order to perform $O = S^T \times D$, we use shared memory corresponding to the columns of column panel to store the output. The shared memory is initialized to zero (line 5-7 in Listing 4). Each row in a column panel is processed by a warp. The input dense matrix and sparse matrix elements are read from the global memory and the partial products are computed (line 17-22). Each such partial product is accumulated in shared memory using atomic operations (line 23). At the end of each column panel, accumulated results in shared memory are updated to global memory using atomic operations (line 28).

In order to process the light rows, each thread loads the corresponding element of the input matrix to a thread local register (line 6 in Listing 5) and computes partial products (line 9-14). Each partial product is updated to global memory using atomic operations (line 15).

Listing 4: SPMMT Pseudocode (Heavy row segments)

```
1. row_offset = tb_idx * IN_TILE_ROW_SIZE;
2. slice_offset = tb_idy * IN_TILE_SLICE_SIZE;
3. warp_id = tid/WARP_SIZE;
4. lane_id = tid%WARP_SIZE;
5. for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
6.   sm_output_value[i][lane_id] = 0;
7. end
8. __syncthreads;
9. for i=seg_start_num[tb_idx] to seg_start_num[tb_idx+1]-1 step
       tb.size()/WARP_SIZE do
10.  val = 0;
11.  start = start_seg_position[i];
12.  end = start_seg_position[i+1];
13.  column_idx = seg_row_position[i];
14.  column_value = input_value[column_idx][lane_id];
15.  for j=start to end-1 do
16.    mod = (j - start)%WARP_SIZE
17.    if mod == 0 then
18.      index_buf = seg_index[j + lane_id];
19.      value_buf = seg_value[j + lane_id];
20.    end
21.    row_offset = __shfl(index_buf, mod);
22.    val = column_value * __shfl(value_buf, mod);
23.    atomicAdd(&sm_output_value[row_offset][slice_offset+lane_id], val);
24.  end
25.  __syncthreads;
26.  row_idx = seg_row_position[i];
27.  for i=warp_id to IN_TILE_ROW_SIZE step tb.size()/WARP_SIZE do
28.    atomicAdd(&output_value[row_offset+i][lane_id],
       sm_out_value[i][lane_id]);
29. end
```

Listing 5: SPMMT Pseudocode (Light row segments)

```
1.  column_offset = (tb_idx*tb.size() + tid) / WARP_SIZE;
2.  slice_offset = tb_idy * IN_TILE_COL_SIZE;
3.  lane_id = tid%WARP_SIZE;
4.  start = csr_row_pointer[column_offset];
5.  end = csr_row_pointer[column_offset+1];
6.  column_value = input_value[column_offset][slice_offset+lane_id];
7.  for i=start to end-1 do
8.    mod = (i - start)%WARP_SIZE
9.    if mod == 0 then
10.     index_buf = csr_column_idx[i + lane_id];
11.     value_buf = csr_column_val[i + lane_id];
12.   end
13.   row_offset = __shfl(index_buf, mod);
```

```
14.      val = column_value * __shfl(value_buf, mod);
    //directly accumulate results in global memory
15.   atomicAdd(&dest_value[row_offset][slice_offset+lane_id], val);
16. end
```

Fig. 14 and 15 compare RS-SpMM with cuSPARSE for $O = S^T D$, for single and double precision, respectively. Two scenarios are evaluated: i) only $O = S^T D$ is required, and ii) both $O1 = S^T D$ as well as $O2 = SD$ required. The performance of RS-SPMM is significantly higher than that of cuSPARSE for both scenarios, for both precisions.

When non-zeros are clustered, performance of cuSPARSE can be higher than 200 GFLOPs. However, when non-zeros are scattered (which results in low cache utilization) the performance of cuSPARSE drops to 10 GFLOPs. On the other hand, performance of RS-SpMM is consistently higher than 80 GFLOPs when concurrency is not very low. Hence, as shown in Fig. 10 (q) and (s), the performance of cuSPARSE is over 16x times slower than ours for more than 20% of matrices.

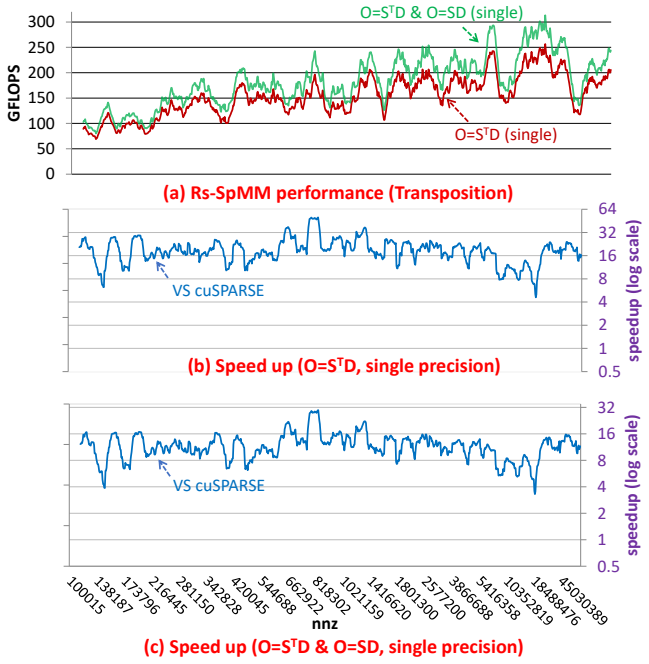


Figure 14: $O = S^T D$ performance ($K = 128$, single precision)

7 DISCUSSION

Although performance of RS-SpMM is often significantly higher than cuSPARSE SpMM, it is still considerably below the upper-bound discussed in Sec. 1. In this section, we analyze the SpMM bottlenecks and possible improvements. The peak bandwidth of shared memory and unified L1 cache of the P100 GPU has been reported to be 1977.25 Gwords / sec and 594.75 Gwords / sec, respectively [17, 20]. For the vertical streaming scheme, one shared memory load is needed for every 2 floating point operations, as can be seen in line 19 in Listing 1. Therefore, an upper-bound on performance of RS-SpMM with vertical streaming is $2 \times 1977.25 =$

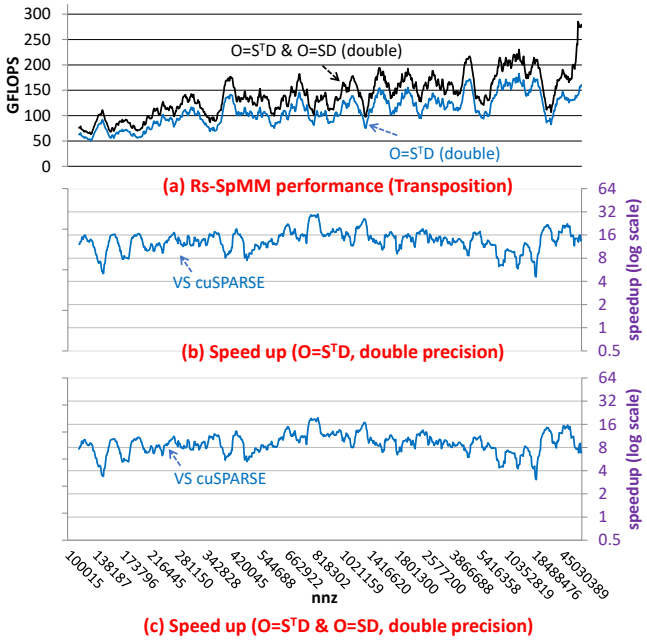


Figure 15: $O = S^T D$ performance ($K = 128$, double precision)

3954.50 GFLOPs. We tested RS-SpMM with an 8K X 8K dense matrix (stored in DCSR format) to determine the achievable performance. The achieved performance of RS-SpMM with that dense matrix was 2495.81 GFLOPs (when $K = 65536$ and $CF_V = 4$), which is 63.1% of the upper-bound GFLOPs.

For the horizontal scheme, to process the sparse non-clustered non-zeros, one load/store operation is associated with two floating point operations, as seen in line 13 in Listing 2. If all elements of the input dense matrix are brought in from the L1 unified cache, an upper-bound on performance with the horizontal streaming of RS-SpMM is $2 \times 594.75 = 1189.5$ GFLOPs.

Let the sparse matrix for SpMM have M rows and N columns, and let K be the width of the input/output dense matrices. Assume that the vertical scheme is used to process SpMM. The minimum volume of data transfer through the LD/ST units can be computed as follows. In this scheme, each element of the output matrix **O** is accessed at least once. Hence, the minimum number of transactions (in words) for **O** is $M \times K$. Each **D** element is multiplied by all the elements in the corresponding column of **S**. Assuming that each **D** element gets an average reuse of R from registers, the minimum number of transactions (in words) for **D** is $\frac{nnz \times K}{R}$. Each **S** element has a K-way reuse. Assuming that the coarsening factor is C, and each **S** access gets full reuse across a full warp (of 32 threads) the minimum number transactions for **S** is $\frac{nnz \times K}{C \times 32}$. Thus the minimum number of load/store transactions for this scheme is $\frac{nnz \times K}{R} + M \times K + \frac{nnz \times K}{C \times 32}$ words. When $M \times R < nnz$ and $R < C \times 32$, the dominant term would be $\frac{nnz \times K}{R}$.

Based on this analysis, we make the following remarks:
1) Any SpMM implementations that do not use shared memory for accessing the input dense matrix will therefore be limited to a

performance lower than 1189.5 GFLOPs.
2) If a sparse matrix is also loaded from shared memory / cache / global memory, performance is limited to 3954.50 / 2 = 1977.25 GFLOPs, since 2 load operations are required per 2 floating point operations.
3) When warp shuffling is used, thread coarsening along the K dimension is not very helpful, since it does not much reduce the number of accesses for LD/ST units.
4) Register reuse is a key to achieve better performance, and thread coarsening along the M dimension can be beneficial, since it can reduce LD/ST transactions.

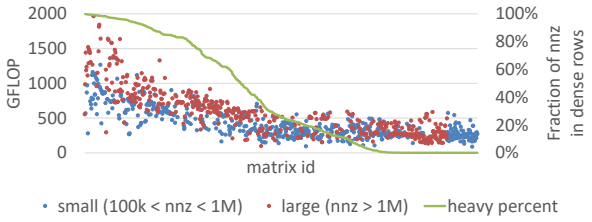


Figure 16: RS-SpMM performance: impact of row-segment density

Fig. 16 presents RS-SpMM performance data for the matrices ordered along the X-axis by the fraction of the non-zeros in the heavy row-segments. Matrices on the left side of graph have higher non-zeros in the heavy row-segments and the matrices on the right have lower non-zeros in heavy row-segments. It can be seen that there is a clear trend showing that matrices with a larger fraction in the heavy part tend to perform better. This suggest that column reordering schemes that increase row-segment density might be a promising direction to further improve performance.

8 CONCLUSION

This paper has presented the development of RS-SpMM, an efficient GPU implementation of the sparse matrix multi-vector multiplication algorithm that exploits the non-uniform distribution of nonzeros in sparse matrices. RS-SpMM was designed to attain good reuse of the elements from the input and output dense matrices, as well as the sparse matrix. The extensive experimental evaluation demonstrates the superior performance of RS-SpMM when compared to other GPU SpMM implementations.

ACKNOWLEDGMENTS

We thank the reviewers for the valuable feedback and the Ohio Supercomputer Center for use of their GPU resources. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Contract D16PC00183, and the National Science Foundation (NSF) through awards 1404995, 1513120, 1629548, 1645599, and 1747447.

REFERENCES

[1] The API reference guide for cuSPARSE, the CUDA sparse matrixlibrary.(v8.0 ed.). http://docs.nvidia.com/cuda/cusparse/index.html. (2018).
[2] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1213–1222.
[3] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2015. Accelerating the LOBPCG method on GPUs using a blocked Sparse Matrix Vector Product. In *Proceedings of the Symposium on High Performance Computing*. Society for Computer Simulation International, 75–82.
[4] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan. 2014. Fast Sparse Matrix-vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 781–792.
[5] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. 2000. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. SIAM.
[6] Allison H Baker, John M Dennis, and Elizabeth R Jessup. 2006. On improving linear solver performance: A block variant of GMRES. *SIAM Journal on Scientific Computing* 27, 5 (2006), 1608–1626.
[7] A. Benatia, W. Ji, Y. Wang, and F. Shi. 2016. Sparse Matrix Format Selection with Multiclass SVM for SpMV on GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*. 496–505. https://doi.org/10.1109/ICPP.2016.64
[8] Aydin Buluc and John R Gilbert. 2008. On the representation and multiplication of hypersparse matrices. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–11.
[9] Carmen Campos and Jose E Roman. 2012. Strategies for spectrum slicing based on restarted Lanczos methods. *Numerical Algorithms* 60, 2 (2012), 279–295.
[10] John Canny and Huasha Zhao. 2013. Bidmach: Large-scale learning with zero memory allocation. In *BigLearn workshop, NIPS*.
[11] SuiteSparse Matrix Collection. 2011. https://sparse.tamu.edu. (2011).
[12] M. Daga and J. L. Greathouse. 2015. Structural Agnostic SpMV: Adapting CSR-Adaptive for Irregular Matrices. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 64–74.
[13] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. 2014. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *URL: http://cusplibrary. github. io/(accessed: 01.02. 2016)* (2014).
[14] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.
[15] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.
[16] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).
[17] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).
[18] Wei Jiang and Gang Wu. 2010. A thick-restarted block Arnoldi algorithm with modified Ritz vectors for large eigenproblems. *Computers & Mathematics with Applications* 60, 3 (2010), 873–889.
[19] Andrew V Knyazev. 2001. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing* 23, 2 (2001), 517–541.
[20] Elias Konstantinidis and Yiannis Cotronis. 2016. A quantitative performance evaluation of fast on-chip memories of GPUs. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*. IEEE, 448–455.
[21] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: An Extended Compression Format for Spmv on Shared Memory Systems. *SIGPLAN Not.* 46, 8 (Feb. 2011), 247–256. https://doi.org/10.1145/2038037.1941587
[22] Weifeng Liu and Brian Vinter. 2015. CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. *CoRR* abs/1503.05032 (2015). http://arxiv.org/abs/1503.05032
[23] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors. *J. Parallel and Distrib. Comput.* 85 (2015), 47–61.
[24] Karl Meerbergen and Raf Vandebril. 2012. A reflection on the implicitly restarted Arnoldi method for computing eigenvalues near a vertical line. *Linear Algebra Appl.* 436, 8 (2012), 2828–2844.
[25] Duane Merrill and Michael Garland. 2016. Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 43, 2 pages. https://doi.org/10.1145/2851141.2851190
[26] Ronald B Morgan and Dywayne A Nicely. 2011. Restarting the nonsymmetric Lanczos algorithm for eigenvalues and linear equations including multiple right-hand sides. *SIAM Journal on Scientific Computing* 33, 5 (2011), 3037–3056.
[27] CUDA Nvidia. 2008. Cublas library. *NVIDIA Corporation, Santa Clara, California* 15, 27 (2008), 31.

[28] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. 2013. Fastspmm: An efficient library for sparse matrix matrix product on GPUs. *Comput. J.* 57, 7 (2013), 968–979.

[29] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster CNNs with Direct Sparse Convolutions and Guided Pruning. (2016). arXiv:1608.01409 arXiv:1608.01409v5.

[30] Xavier Pinel and Marc Montagnac. 2013. Block Krylov methods to solve adjoint problems in aerodynamic design optimization. *AIAA journal* (2013).

[31] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally Homogeneous, Locally Adaptive Sparse Matrix-vector Multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 13, 11 pages. https://doi.org/10.1145/3079079.3079086

[32] Narayanan Sundaram and Kurt Keutzer. 2011. Long term video segmentation through pixel level spectral clustering on gpus. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*. IEEE, 475–482.

[33] Francisco Vazquez, G Ortega, José-Jesús Fernández, and Ester M Garzón. 2010. Improving the performance of the sparse matrix vector product with GPUs. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 1146–1151.

[34] F V'zquez, G Ortega, JJ Fern'ndez, Inmaculada García, and Ester M Garzón. 2012. Fast sparse matrix matrix product based on ELLR-T and gpu computing. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 669–674.

[35] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[36] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu. 2010. Efficient PageRank and SpMV Computation on AMD GPUs. In *2010 39th International Conference on Parallel Processing*. 81–89. https://doi.org/10.1109/ICPP.2010.17

[37] Ichitaro Yamazaki, Tingxing Dong, Raffaele Solcà, Stanimire Tomov, Jack Dongarra, and Thomas Schulthess. 2014. Tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. *Concurrency and computation: Practice and Experience* 26, 16 (2014), 2652–2666.

[38] Ichitaro Yamazaki, Hiroto Tadano, Tetsuya Sakurai, and Tsutomu Ikegami. 2013. Performance comparison of parallel eigensolvers based on a contour integral method and a Lanczos method. *Parallel Comput.* 39, 6 (2013), 280–290.

[39] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaspmv: Yet another spmv framework on gpus. In *Acm Sigplan Notices*, Vol. 49. ACM, 107–118.