Near-Memory Data Transformation for Efficient Sparse Matrix Multi-Vector Multiplication

Daichi Fujiki
dfujiki@umich.edu
University of Michigan, Ann Arbor

Niladrish Chatterjee
nchatterjee@nvidia.com
NVIDIA

Donghyuk Lee
donghyukl@nvidia.com
NVIDIA

Mike O'Connor
moconnor@nvidia.com
NVIDIA
University of Texas, Austin

ABSTRACT

Efficient manipulation of sparse matrices is critical to a wide range of HPC applications. Increasingly, GPUs are used to accelerate these sparse matrix operations. We study one common operation, Sparse Matrix Multi-Vector Multiplication (SpMM), and evaluate the impact of the sparsity, distribution of non-zero elements, and tile-traversal strategies on GPU implementations. Using these insights, we determine that operating on these sparse matrices in a Densified Compressed Sparse Row (DCSR) is well-suited to the parallel warp-synchronous execution model of the GPU processing elements.

Preprocessing or storing the sparse matrix in the DCSR format, however, often requires significantly more memory storage than conventional Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) formats. Given that SpMM kernels are often bottlenecked on DRAM bandwidth, the increase in DRAM traffic to access the larger DCSR formatted data structure can result in a slowdown for many matrices.

We propose a near-memory transform engine to dynamically create DCSR formatted tiles for the GPU processing elements from the CSC formatted matrix in memory. This work enhances a GPU's last-level cache/memory controller unit to act as an efficient translator between the compute-optimized representation of data and its corresponding storage/bandwidth-optimized format to accelerate sparse workloads. Our approach achieves 2.26× better performance on average compared to the vendor supplied optimized library for sparse matrix operations, cuSPARSE.

ACM Reference Format:
Daichi Fujiki, Niladrish Chatterjee, Donghyuk Lee, and Mike O'Connor. 2019. Near-Memory Data Transformation for Efficient Sparse Matrix Multi-Vector Multiplication. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3295500.3356154

SC '19, Nov. 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356154

1 INTRODUCTION

Numerous important algorithms from various application domains use linear algebra operations as their fundamental building blocks. These applications belong not only to the traditional domains of scientific and high-performance computing (HPC) [2, 3, 5, 14, 16, 20, 27, 36, 37], but also important enterprise domains, such as graph analytics [25, 28, 33, 34], and emerging paradigms like deep neural networks (DNNs) [18, 29]. More specifically, it is *sparse* linear algebra that is central to these applications as the nature of the problem domains often lead to input datasets that have few non-zero values in them. In large graphs, for example, the adjacency matrix representation is often naturally sparse because each vertex in the graph is connected to only a small subset of the aggregate collection of nodes in the graph. On the other hand, in DNNs, pruning of neural connections is a major focus of energy and performance optimization [11, 26] leading to sparse input tensors at various stages of the pipeline.

In this paper, we focus on analyzing and optimizing Sparse Matrix Multi-Vector Multiplication (SpMM), a prime representative of sparse linear algebra kernels, and a frequent substrate for many important applications, on a Graphics Processing Unit (GPU) platform. While the massive parallel execution facilities offered by GPUs are a natural fit for the inherently parallel nature of matrix-matrix multiply, the efficient utilization of the GPU's high bandwidth memory interface is often the key to achieving high overall performance for these memory bound workloads. Consequently, in our work, we aim to optimize the memory traffic through a combination of efficient algorithms that leverage tiling of the matrices to minimize off-chip requests and a hardware mechanism that converts data from a bandwidth-efficient format to a compute-efficient format during execution.

Through our exploration of tiled-SpMM, we discover that the *nature* of sparsity of the input sparse-matrix, *i.e.*, the distribution of the non-zero values, actually determines the optimal tiling and traversal strategy. With the help of an analytical model of the memory access patterns of different tiling approaches and traversal orders, we arrive at a heuristic that helps us choose the optimal algorithm based on the input matrix's sparsity pattern. However, tiling the sparse matrix, while beneficial from a parallelization point of view, actually creates a new computation bottleneck. When a sparse matrix is tiled into several vertical strips, several rows have no non-zero elements in the tile, despite having a few non-zero values in the untiled matrix. Using the conventional CSR format for

sparse matrices in these cases leads to redundant memory traffic for these rows that have been virtually emptied as a consequence of tiling and wasted cycles in the compute pipeline. As a consequence we leverage past work on a dense representation of a CSR format, densified CSR (*DCSR*) [12], to optimize the memory traffic and compute pipeline inefficiencies associated with tiling the sparse matrix. However, the tiled-DCSR format has non-trivial storage overhead due to the increased metadata (to track the empty rows). To solve these disparate needs, where on the one hand the CSR and CSC representations are good for storage and access bandwidth, while the DCSR format is conducive for execution time, we design a hardware unit placed in the GPU's memory controller to perform the conversion from the CSC format in the memory to the appropriate DCSR format for each tile of the sparse matrix, at runtime. Essentially, the software calls an online format conversion API that uses this hardware unit to deliver data to the compute units. Over a large suite of sparse matrices, we find that our techniques lead to significant improvements over the commercial cuSPARSE library [23].

In summary, the two most significant contributions of this paper are

- A detailed analysis of the impact of the nature of sparsity of the input matrix on the efficiency of tiling and traversal strategies for parallel SpMM on GPUs. This analysis is then used to design a hybrid algorithm.
- A hardware mechanism that performs near-memory data transformation to translate a storage and bandwidth efficient format of a sparse matrix to a compute efficient format tailored for the algorithm at hand. This technique can be reinterpreted in the context of other problems, and represents a new and tractable application for near-data processing paradigms.

2 BACKGROUND – SPMM ON GPUS

Sparse Matrix Multi-Vector Multiplication or SpMM is one of the key kernels used in a wide range of applications [3, 5, 25, 27, 32, 37]. These applications include many scientific or numeric applications such as blocked eigen solvers [2, 16] and non-negative matrix factorization [14], graph processing such as graph centrality calculations [28] and all-pairs shortest path [33], and data science such as pruned neural network [11, 26].

SpMM multiplies sparse matrix A by dense matrix B as shown in Algorithm 1. The result C can be either sparse or dense, but in many practical scenarios where at least 1 non-zero element exists in each row, the result C is likely to be as dense as B.

Real sparse matrices are highly sparse, with density less than 10% and are typically stored using a compressed format. We target these sparse matrices. While there have been many different kinds of compressed formats proposed in the past, Compressed Sparse Row or CSR has become the community standard.

CSR for a sparse matrix comprises three vectors, *value*, *colidx*, and *rowptr*. Figure 1 shows a sparse matrix with three rows and three columns and only five of the total nine elements having non-zero values (denoted by *a*,*b*,*c*,*x*, and *y*). The CSR representation of the matrix is shown in the same figure. The *value* vector contains all five of the non-zero elements in the matrix, and the

Algorithm 1 Sparse Matrix Multi-Vector Multiplication (SpMM).

Input: CSR A[M][N], float B[N][K]
Output: float C[M][K]
1: **for** $i = 0$ to $A.rows - 1$ **do**
2: **for** $j = A.rowptr[i]$ to $A.rowptr[i+1] - 1$ **do**
3: **for** $k = 0$ to $K - 1$ **do**
4: $C[i][k] += A.values[j] \times B[A.colidx[j]][k]$
5: **end for**
6: **end for**
7: **end for**

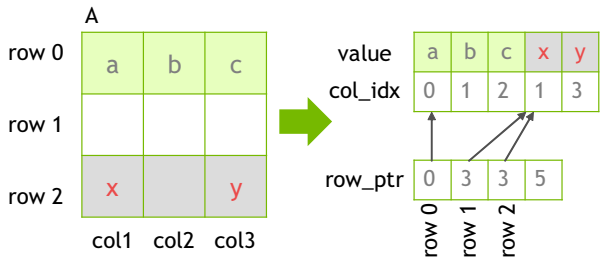

Figure 1: CSR format.

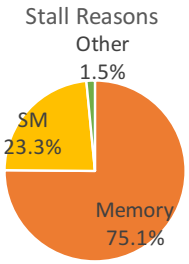

Figure 2: Stall reasons of SpMM (NVPROF).

colidx vector stores the column index for the corresponding entry in the *value* vector. Thus the *value* and *colidx* vectors each have as many elements as the number of non-zero elements in the matrix. The *rowptr* vector stores the indices that indicate the boundaries of rows in the *value* and *colidx* vectors. So each element of the *rowptr* corresponds to one row in the matrix and points to the start of the contents of that row in the *value* and *colidx* vectors. If *rowptr*[i] and *rowptr*[i+1] have the same index stored in them, it indicates that row[i] is empty.

SpMM implementations on GPUs are very often bottlenecked by memory accesses. This is illustrated in Figure 2 which shows that the majority of the stall time in a GPU can be attributed to fetching data from memory when executing SpMM. This can be attributed to two reasons. First, the inherent indirections in the CSR format necessitate multiple memory accesses to fetch each non-zero element in the sparse matrix. Second, as evident from Algorithm 1, which elements of B matrix are useful is dictated by the contents of the *colidx* vector. Therefore accesses to elements of the dense matrix are predicated on accesses to the elements of the CSR representation of the sparse matrix. These two factors compound to increase the demand for the memory subsystem during the execution of SpMM.

A simple analytical model can be used to estimate the extent of the memory-bound nature of an SpMM execution. The CSR representation of a $N \times N$ sparse matrix A requires a total of $8 \times nnz + 4 \times (N+1)$ bytes (nnz = number of non-zero elements) assuming 4 bytes for each element of the *rowptr*, *colidx* and *value* vectors. Assuming a dense matrix of the same dimension as the sparse matrix, accesses to the dense matrix and the output matrix generate an additional $8N^2$ byte of memory traffic. On the other hand, the total floating point operation count is $2 \times nnz \times N$ as each non-zero element in A needs to go through a multiplication with a vector of elements in a row of B as well as an addition operation to accumulate the result. Thus the bytes/FLOP for SpMM is given by $\frac{8 \times nnz + 8N^2 + 4 \times (N+1)}{2 \times nnz \times N}$. Using typical values we found in the suite of matrices we evaluated, $N = 20K$ and 0.1% density of matrix A, we find that SpMM has a byte/FLOP of 5.1 bytes/FLOP. Clearly, SpMM is memory bandwidth bound.

3 WORKLOAD ANALYSIS

In this section, we analyze approaches for SpMM on modern GPUs with special emphasis on tiling techniques that impact reuse and computation efficiency. This analysis helps us establish the proper baseline technique for SpMM on a GPU in terms of memory access behavior and sheds light on the type of data transformation that we can perform near memory to accelerate this workload.

3.1 Data Access Locality

Tiling is an obvious technique to distribute a data-parallel workload like SpMM amongst the many compute cores on a GPU. Each SM can be tasked to work on a slice of input sparse matrix and the input dense matrix to produce a tile of the output matrix (or part thereof). Matrix partitioning (*a.k.a.* tiling) and tile traversal order have a direct impact on data access locality and performance. By being aware of the tile access patterns, it is possible to allow a sub-matrix to stay in the shared memory in a Streaming Multiprocessor (SM) and/or in the cache hierarchy, enabling higher data reuse.

3.1.1 C-/B-/A-Stationary. Clearly, the three possible tiling strategies are to keep a tile of the output matrix (C), or the input sparse matrix (A), or the input dense matrix (B) in the shared memory to maximize reuse. We refer to these as C-stationary, A-stationary and B-stationary respectively.

C-stationary. The most intuitive approach, output (C) stationary, computes a tile or sub-matrix of output C in its entirety in one SM. Matrix A is horizontally partitioned (horizontal strips) and processed by a thread block. To calculate complete cells of C, the thread block reads a vertical partition (vertical strip) of B. Threads in a warp can be mapped across columns in the vertical strip of B processing with a row in the horizontal strip of A (row-per-warp) or rows in the horizontal strip of A processing with a column in the vertical strip of B (row-per-thread). The row-per-warp approach accumulates the partial sums in a thread local register and does not need to perform a warp reduce operation or atomic updates to a shared memory location to collect all partial contributions calculated in each thread in the warp. However, since this technique distributes the threads across the columns of B, the last column slice will be load imbalanced if the number of columns is not a multiple of 32 (the width of a warp). The row-per-thread approach can solve this issue, but variation in the number of non-zero elements (nnz) across rows imbalances the load for each thread. This type of load imbalance generally is more common than the load-balancing cause by the reminder columns of the row-per-warp approach. Thus, row-per-warp is the technique of choice when performing output-stationary tiling for SpMM.

The memory footprint of the input sparse matrix is generally smaller than the input dense matrix. To maximize the potential cache hit at the last level cache (LLC) of larger B strips, we iterate over the strips of A, reusing the same B strip. C-stationary approach minimizes the memory footprint of output dense matrix C and does not incur atomic operations, while B cannot be reused except for LLC hits.

B-stationary. In the B-stationary approach, a tile of B is loaded into the shared memory only once. Sparse A matrix has to be tiled, and tiles in a vertical strip of A are processed by a thread block, while different thread blocks in other SMs can concurrently compute partial sums for the same output C tile. Because of this concurrency, the B-stationary technique requires atomic operations to update C tiles. The size of the B tile is determined based on the shared memory size. As discussed above, we employ row-per-warp approach inside a thread block. The partial sums of C can be cached at LLC, and to maximize the reuse in LLC, different SMs execute a kernel for different tiles within a vertical strip of A and calculate partial contributions of the same set of C tiles. However, this technique is sensitive to the footprint of the output C matrix and the atomic bandwidth in the LLC/memory to compute C.

A-stationary. The third alternative, sparse matrix (A) stationary, stores a tile of A into the shared memory, multiplies its non-zero elements by the elements in a horizontal strip of B, and stores the partial contributions to a vertical strip of C. This option is not common, because B and C have to be visited multiple times, resulting in the largest number of memory accesses across all three tiling techniques.

3.1.2 Memory Traffic Comparison. Table 1 compares the compulsory memory traffic of the three alternatives. We do not consider data reuse in caches. We can see A-/B-/C-stationary just requires a single fetch of matrix A/B/C, respectively, and has to visit other matrices multiple times. Moreover, A and B stationary produce partial contributions of C, thus requiring up to 2× access latency for atomic operations. Since the input sparse matrix A has much smaller memory footprint than dense B and C, A-stationary is not efficient in bandwidth consumption. Thus, B- or C-stationary are the positive candidates.

Table 1 also shows an analytical model of the amount of memory traffic. In this model, d denotes the density of the input sparse matrix, n the number of rows in the matrices and k the number of rows in a tile. For simplicity, we assume all matrices have a dimension of $n \times n$. Typically, the memory traffic of B for B-stationary and C for C-stationary are similar, especially when nnz/row and nnz/col are greater than 1. Therefore, the memory traffic of C for B-stationary and B for C-stationary is the key factor for the performance.

With the uniform non-zero distribution with density d, C-stationary provides better performance than B-stationary because B-stationary

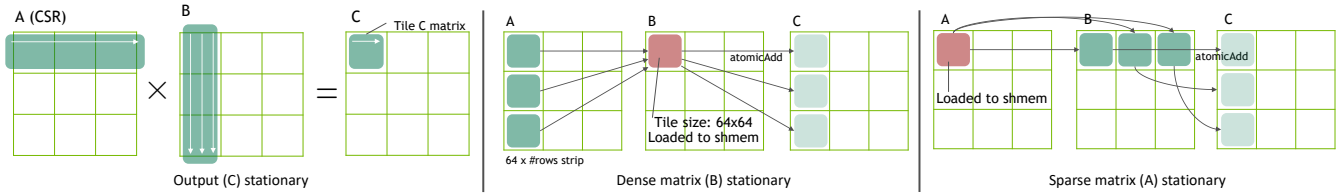

Figure 3: Alternatives of tiling approach for SpMM: C-stationary(left), B-stationary(middle), A-stationary(right).

Table 1: Compulsory Memory Traffic Comparison.

	A (small)	B (large)	C (large)
A Stationary	Single fetch $size(A.csr)$	Multiple fetches $A.nnz \times n$	Multiple updates $n_{nnzrow_{strip}} \times \frac{n}{k} \times n \times 2^*$
B Stationary	Multiple fetches $size(A.csr) \times \frac{n}{k}$	Single fetch $n_{nnzcol} \times n$	Multiple updates $n_{nnzrow_{strip}} \times \frac{n}{k} \times n \times 2^*$
C Stationary	Multiple fetches $size(A.csr) \times \frac{n}{k}$	Multiple fetches $A.nnz \times n$	Single update $n_{nnzrow} \times n$

matrix size = $n \times n$, tile size = $k \times k$,
*atomic bandwidth = 2× memory access,
$A.nnz = dn^2 < n^2$,
$n_{nnzrow} \approx n_{nnzcol} \approx n$ (uniform distribution),
$n_{nnzrow_{strip}} \approx \{1 - (1-d)^k\}n$.

suffers from the atomic bandwidth. However, in real matrices, their non-zero distribution is skewed and not all tiles in a row contribute to a C tile. This could more than amortize the cost of the atomic update and brings performance benefits compared to C-stationary. On the other hand, the skewness does not affect the performance of C-stationary because regardless of the non-zero distribution the necessary elements of B are fetched (n times) from the memory as in Table 1 and they cannot be reduced based on the skewness unlike the C tiles of C-stationary. We use a heuristic described later to determine which algorithm to use based on our profiling result.

3.1.3 Tile traversal strategies. The analytical model does not consider the effect of caches. We will discuss how to maximize cache hits, assuming we employ B-stationary. There are two ways to traverse the tiles on B, column-major traversal and row-major traversal.

Row-major traversal launches kernels for B tiles that align in a row, then proceeds to the next row. By traversing B in row major, different SMs read the same vertical A strip, while touching different output strips of C. This approach can possibly capture the locality of A in LLC, however, touching entire C multiple times is rather expensive.

On the other hand, column-major traversal traverses B tiles in a column. This approach reads strips of A most frequently but can utilize the locality of C to some extent by writing back to the same tiles until all partial sums are accumulated. Because of the difference in the memory footprint of A and C, the column-major traversal usually gives better performance. Similarly, we have a similar tradeoff of the traversal order of A and B in C-stationary.

There can be further opportunities for optimizations using 2D or hierarchical tiling to maximize cache reuse in LLC. They are orthogonal to our proposal and we do not discuss them in this paper.

3.1.4 Non-Zero Distribution and Heuristics. Non-zero distribution has considerable implications on the algorithm selection. For example, denser sparse matrix benefits from data reuse from B-stationary, while uniform non-zero distribution makes B-stationary less efficient due to the atomic bandwidth.

In this section, we analytically assess the number of memory access based on the analytical model in Table 1. We design a heuristic which produces larger value when it estimates B-stationary to be advantageous for an input matrix. We first discuss memory traffic of output C. While $n_{nnzrow} \approx n_{nnzcol}$ holds in many cases, some matrices have skewed (row-wise) non-zero distribution which results in very small n_{nnzrow}. This is advantageous for C-stationary but can be less efficient for B-stationary because $n_{nnzrow} < n_{nnzcol}$ will break the symmetricity of memory traffic of B for B-stationary and C for C-stationary. On the other hand, a smaller $n_{nnzrow_{strip}}$ reduces the number of atomic operations and is beneficial for B-stationary.

The bandwidth of loading B matrix is mainly determined by the density. Having larger density can increase the data reuse of B-tiles for B-stationary. On contrary to C for C-stationary, B for B-stationary is difficult to benefit from smaller n_{nnzcol} because figuring out empty columns and aligning B submatrices with skipped columns are challenging with row-major data formats (*i.e.* CSR and DCSR).

We also take the skewness of non-zero distribution into account. Matrices with the skewed distribution tends to have numbers of heavy row segments and empty row segements, which contributes to higher locality. We use normalized entropy H_{norm}, which can be obtained by dividing Shannon's entropy by Hartley's entropy and represents the randomness of the information. H_{norm} is derived as follows:

$$H_{norm} = -\sum_{t \in A.tiles} \sum_{r \in t.rows} \frac{r.nnz}{A.nnz} \log \frac{r.nnz}{A.nnz} \cdot \frac{1}{\log A.nnz}, \quad (1)$$

where $A.tiles$ denotes the set of tiles in A, $t.rows$ the row segments in tile t, and $x.nnz$ nnz within x.

From the discussion above, we use the following Sparsity Skewness Function (SSF) as the heuristic to determine the algorithm for

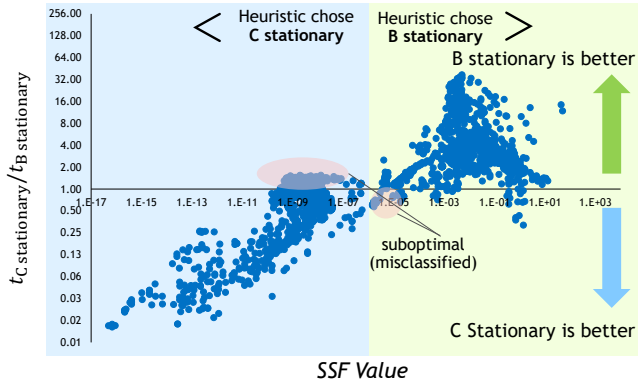

Figure 4: Performance vs. SSF value. A dot is plotted for each matrix in [9]. $y > 1$ means B-stationary performs better.

a given input sparse matrix

$$SSF = \frac{n_{nnzrow}/n}{mean(n_{nnzrow_{strip}}/n)} \cdot \frac{A.nnz}{n} \cdot (1 - H_{norm}). \quad (2)$$

SSF is meant to provide a good baseline algorithm. As in recent works (*e.g.* [12]), state-of-art kernels pick the algorithm based on input matrix profiles. We profile approximately 4,000 matrices from SuiteSparse Matrix Collection [9] and use the learned threshold value SSF_{th} of SSF to determine the algorithms. We fully scanned all matrices to evaluate the heuristics. We believe these parameters can be obtained through sampling to minimize profiling time, but we leave it for future work.

Figure 4 shows our profiling result. In this plot, the x-axis shows the value produced by SSF, and the y-axis shows the execution time of C-stationary normalized to that of B-stationary, which essentially says that B-stationary is better when the dot is above one, otherwise C-stationary is better. Using the learned threshold SSF_{th} we vertically split the graph and when SSF is larger than the threshold, we choose B-stationary, and otherwise, we choose C-stationary. There are some points at the upper left and lower right quadrants, where our heuristic misclassifies the matrices to the suboptimal approach. However, they are small in number and more than 93% are correctly categorized. Forthcoming results in this paper use this heuristic as well.

3.2 Computation Efficiency

While tiling allows the utilization of the parallel resources in the GPU and careful travel strategies can optimize reuse patterns, using the naive sparse matrix format (CSR) for tiles of A adversely impacts the computation and storage efficiency. Due to the limited shared memory capacity, it is often desirable to traverse narrow vertical strips of the sparse input matrix. Given a typical tile width of 64, Figure 5 depicts the sparsity of the strips of sparse matrices as a histogram of the non-zero row density in a strip. We observe that the vast majority of rows in a strip of A (the input matrices are from the SuiteSparse Matrix Collection [9]) are all zeros. Using CSR to represent these sparse tiles with a majority of empty rows has two pitfalls.

First, sparse strips have a lot of redundant information in the row pointer. As pointed out above, 99% of rows in the strips are

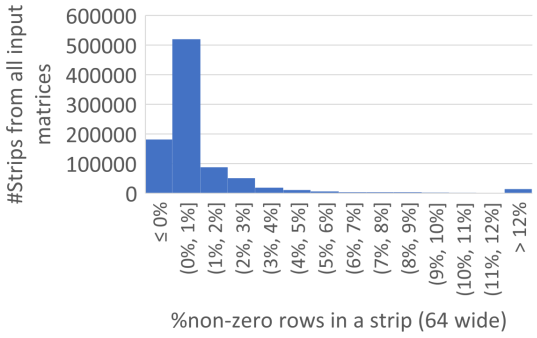

Figure 5: Histogram of density of vertical strips of A.

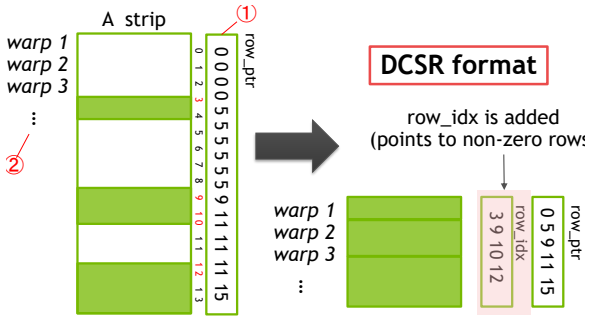

Figure 6: Inefficiency of CSR strips: ①Redundant row pointer data (Bandwidth intensive for low information content). ②Difficult to keep warps/lanes busy.

empty on average, resulting in having approximately 99 copies of redundant row pointers for every single entry that has a useful piece of information. For example, in Figure 6, while only rows 3, 9, 10, 12 have non-zero element(s), the CSR representation still needs a full set of *rowptr*, one for each row. Naturally this introduces an unwanted overhead in the memory access stream of A tiles. Second, it will be difficult to keep the compute lanes busy, since processing elements will spend a significant fraction of runtime trying to find work (*i.e.* non-empty rows) in the highly sparse strips of A.

To alleviate this problem we utilize the Densified-CSR (DCSR) format, introduced in previous work by Hong et al. [12]. In DCSR, another level of indirection is added in the data structure by the introduction of the *rowidx* vector. Each element of the *rowidx* vector is an index of a row in the tile of A that has at least one non-zero value. By paying the additional metadata cost for row indices to specify the non-zero rows, DCSR can compress the empty rows as in Figure 6 (right). When the tile is highly sparse as is the typical case for vertically stripped CSR, introducing DCSR not only removes the redundant entries in the *rowptr* vector and thus improves memory behavior, but also boosts computation efficiency, as warps can be devoted to work on rows that have non-zero values.

Figure 7 shows the percentage of inactive thread executions reduced after introducing DCSR. *Inactive* shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence. Large inactive executions in Tiled CSR are caused by the empty rows, where we have one active thread to skip over the `row_ptr`. We observe 90% reduction of the inactive thread execution. The reduction of storage is even

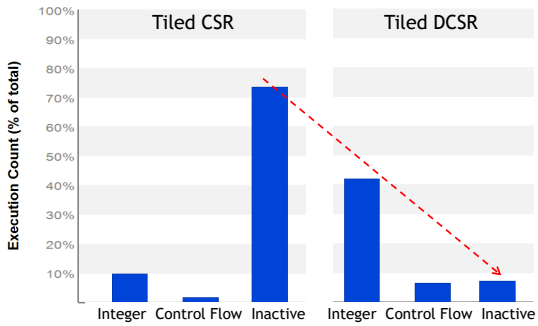

Figure 7: Reduced inactive thread executions by introducing DCSR (NVPROF).

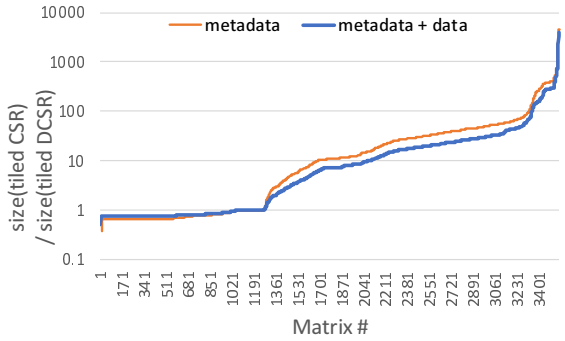

Figure 8: Metadata storage size of tiled DCSR normalized to that of tiled CSR.

dramatic. Figure 8 shows the footprint of tiled CSR normalized to DCSR. Tiled DCSR commonly has orders of magnitude smaller footprint compared to tiled CSR. There are some exceptions for matrices which have a large number of non-zero row segments in their non-zero strips.

3.3 Metadata Storage of DCSR

Tiled DCSR increases computation efficiency and reduces metadata storage for tiles. We could do this format transformation offline and store the DCSR metadata in the GPU main memory. However, in addition to non-trivial transformation cost [39], tiling inevitably adds additional metadata regardless of the type of the tile format (*i.e.* tiled-DCSR or tiled-CSR), because untiled (original) CSR is usually the most storage efficient data format. Since SpMM is bandwidth bottlenecked, this additional metadata does impact the performance if they are read from the memory, although we have shown tiled DCSR has a decent footprint compared to tiled CSR.

Figure 9 illustrates tiled DCSR footprint normalized to (original) CSR. On average, tiled DCSR has 1.3-1.4× (2× at the maximum) storage overhead for tiling, except for some tall skinny matrices which have few non-zero strips and a small number of non-zero rows. Importantly, as discussed above, the sparse input matrix A is most frequently read out from the GPU main memory. Since SpMM is bandwidth bottlenecked, this storage overhead is not negligible.

The storage overhead of the tiled data format is even higher for matrices with scattered non-zero distribution (*i.e.* less skewed data). However, under a situation where the storage overhead of the tiled data format is tolerable, the tiled algorithm can offer better

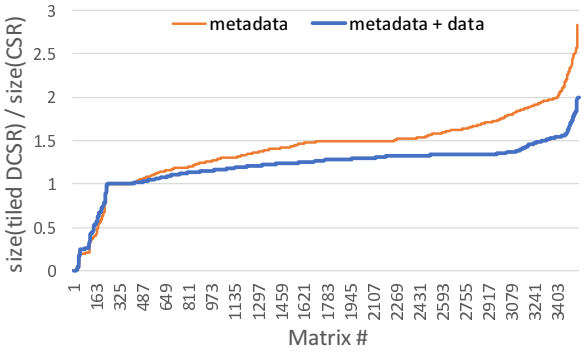

Figure 9: Storage overhead of tiled DCSR.

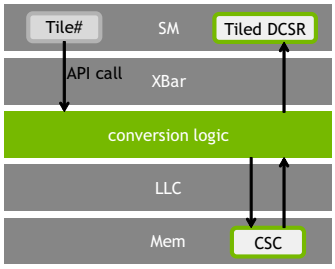

Figure 10: Overall system flow.

performance due to its ability to reuse tiles in the shared memory, especially when the density is relatively high.

This led us to the idea of online tiling and DCSR conversion, which dynamically generates tiles and annotates for DCSR from storage efficient untiled formats. In the following section, we present a technique that enables the small memory footprint of the untiled format with the compute / data-reuse performance and tiling capability of DCSR, without incurring preprocessing cost.

4 ONLINE DATA FORMAT CONVERSION

We introduce a technique of online data format conversion to address the discrepancy of the storage-efficient data format and the computation efficiency data format. While many of the near-memory transaction optimization approaches have explored the design space of efficient data compression schemes for relaxing memory bandwidth targeting general memory traffic, our approach converts the specific data format and annotates useful information for efficient computation.

The execution flow of our system is depicted in Figure 10. The data conversion unit is placed in a frame buffer (FB) partition and accessed by an API call from a user program on a GPU. An example API which requests a tile of A is shown in Figure 11. This API is an intrinsic function converted by the compiler into a message sent to the conversion unit along with the current frontier data (much like a warp vector store instruction) and the pointers to the input CSC and the output DCSR. The request is queued and processed in the order of arrival, and kicks off the conversion unit. The conversion unit fetches elements within a fixed number of rows starting from `row_start` in a strip specified by `strip_id`, keeping track of the current row frontier of CSC using `col_frontier`. Then, it returns

```
/* Device Code */
int col_frontier[64] = {0};
DCSR tiled_dcsr;

for (row_start = 0 ;
     row_start < num_rows ;
     row_start += DCSR_HEIGHT /*=64*/)
{
  GetDCSRTile(strip_id, row_start, col_frontier
    csc.val, csc.row_idx, csc.col_ptr,  //inputs
    tiled_dcsr.val, tiled_dcsr.col_idx, //outputs
    tiled_dcsr.row_ptr, tiled_dcsr.row_idx,
    &nnzrows, &nnz);

  /* Tiled DCSR kernel here */
}
```

Figure 11: Online conversion API.

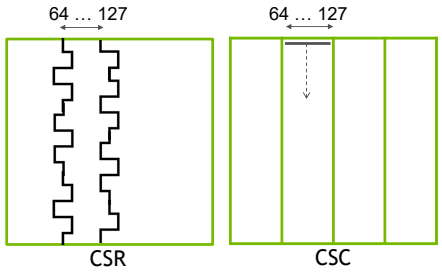

Figure 12: Column slicing approach of different baseline formats (a) CSR (b) CSC.

a stream of converted tiled DCSR data which will eventually be stored in the shared memory of the SM the request is sent from.

Since FB partitions do not communicate with each other, the entire data required to produce a tile have to be partitioned and stored into the same DRAM partition. We will discuss later in Section 6 the effect of this restriction to load balancing and our solutions.

4.1 Baseline Data Format

While widely-used CSR can be used as the baseline format for the conversion, this will cause a lot of challenging problems. Two approaches can be considered for the CSR-to-DCSR conversion: stateless and stateful approach. The stateless approach does not hold any state in the conversion logic. To construct a strip of columns c to $c+N$, the conversion logic needs to scan for each row and find non-zero entries such that $colidx[i] \in \{c \,..\, c+N\}$, because the number of non-zero entries in each row is different and the memory address of an element in a specific column cannot be statically figured out. Since a binary scan is $O(\log n)$, the overall complexity of one conversion is at least $O(n \log n)$. This incurs prohibitive bandwidth cost and also hardware complexity.

The stateful approach remembers the previous frontiers of rows to quickly find the next strip. In this case, the conversion logic has to manage the jagged frontier of entire row entries as in Figure 12 (a), and this requires large metadata storage. Moreover, this is only useful when strips are sequentially accessed, and random access to a strip is as costly as the stateless approach. This situation is likely to happen because multiple SMs can work on different strips.

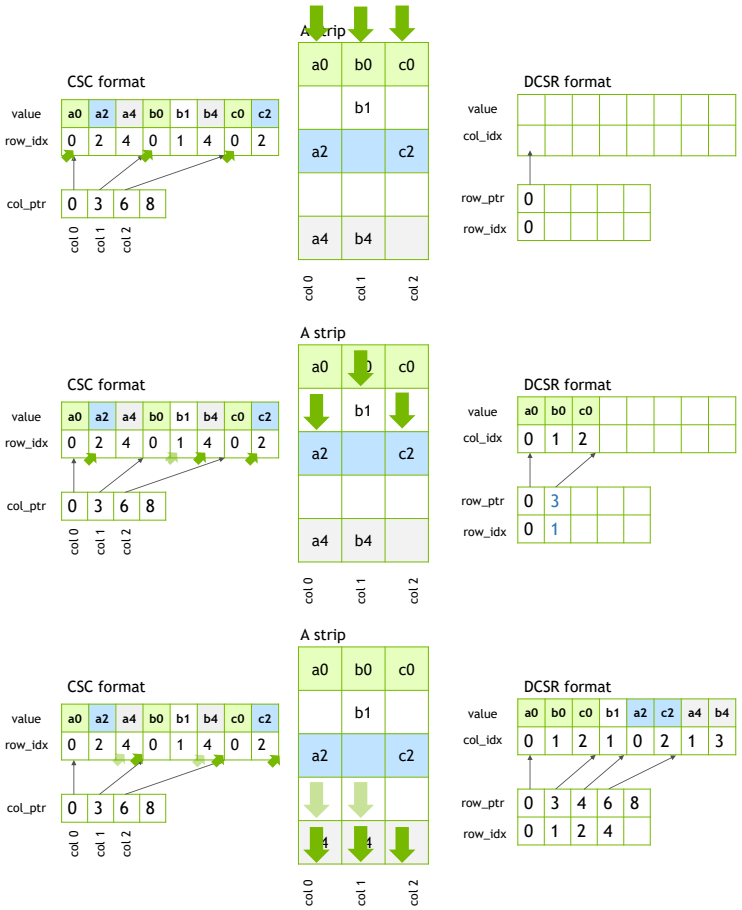

Figure 13: Walk-through example of CSC-to-DCSR conversion.

We propose to use CSC as the baseline data format. CSC is aligned with respect to its columns, and thus extracting a vertical strip from CSC is fairly easy compared to CSR. Conversion from CSC to DCSR just has to walk down the columns starting from the index in *colptr* as illustrated in Figure 12 (b). CSC is approximately the same size as CSR for square matrices, but it does not require additional metadata storage to efficiently build vertical strips unlike the stateless approach of CSR. Random access to a vertical strip can also be efficiently supported. For non-square matrices, CSC's `col_ptr` and CSR's `row_ptr` can have different storage size, and CSC becomes larger when the sparse matrix is wide. If this is common in a workload, a DCSC kernel can potentially be a host kernel at SMs, performing CSR-to-DCSC conversion using the same engine. Note that deserializing data from a serialized format for CSC is almost equivalent to CSR in complexity. For example, widely-used Matrix Market format [24] uses coordinate list (COO) format.

4.2 Microarchitecture

4.2.1 Walk-Through Example. Figure 13 shows a walk-through example of the online CSC-to-DCSR conversion. ① `col_frontier` of each lane is initialized with `col_ptr` of CSC. `col_frontier` points to the first element in each column of the strip (fat green

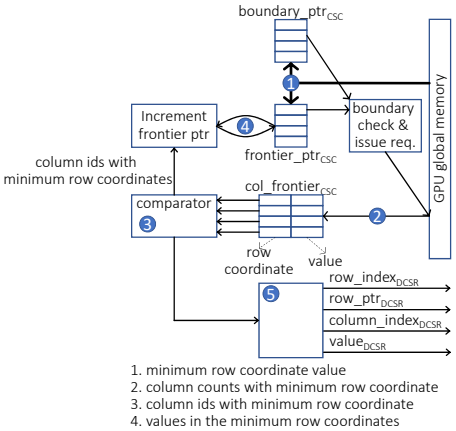

Figure 14: CSC-to-DCSR conversion engine.

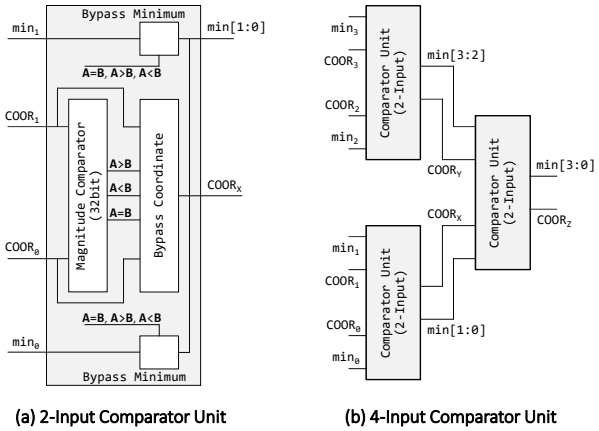

Figure 15: Detailed hardware design for comparator unit.

arrows in Figure 13). ② Find lanes with smallest `row_idx` and copy the element in the lanes to the output DCSR along with `col_idx`. `row_ptr` and `row_idx` are updated to point the first entry of the lanes. `col_frontier` of the lanes is incremented unless it reaches `col_ptr` of the next column. ③ Repeat ② until the lanes sweep all the elements in the designated matrix field. ④ Return DCSR.

4.2.2 Architecture Overview. We build an example hardware that converts a matrix from CSC form to N-column wide DCSR form as shown in Figure 14. `col_ptrs` of N columns are stored in two N-element wide arrays ❶. One array holds the original pointer values that show the range of elements in columns (`boundary_ptr`). The other vector holds N column pointers (`frontier_ptr`), corresponding elements of which form the `col_frontier`.

The first step is checking whether there are any elements in each column by comparing elements in `frontier_ptr` and `boundary_ptr`. If the element in `frontier_ptr` is smaller than that in `boundary_ptr`, we see that there is at least one remaining elements so we load its coordinate and element ❷. If the element of `frontier_ptr` turns to be the same as that of `boundary_ptr`, all elements in the column are translated to elements in DCSR format. Second, the comparator module takes `N` row coordinates, each of which is the frontier of a column in CSC. This comparator module consists of multiple stages of magnitude comparators with a minimum coordinate vector that returns *i)* the minimum row coordinate value and *ii)* column indices that include those minimum row coordinates ❸. The outputs of the comparator module are used for two processes. First, the column indices of the minimum row coordinates update `col_frontier` by increasing the corresponding column indices in `frontier_ptr` ❹ and generating requests for data of the updated indices ❷. Second, those are used for generating a row of DCSR format by transferring the minimum row coordinates as a row index, the number of the minimum column coordinates as the increment of row pointer, and the column indices of the minimum coordinates as column indices in DCSR format ❺.

Figure 15 shows the details of the comparator and its hierarchical organization. We extend 4-bit magnitude comparator to a 32-bit magnitude comparator to simultaneously evaluate two integer-type coordinate inputs. The 2-input comparator unit, shown in Figure 15 (a), consists of a 32-bit magnitude comparator, a coordinate bypass unit, and minimum coordinate vector bypass units (hereinafter called minimum bypass units). The coordinate bypass unit is a simple multiplexer that drives a minimum coordinate based on the output of the magnitude comparator. The minimum bypass unit generates a bit vector of the positions of the minimum coordinates. For example, if $COOR_0$ is smaller than $COOR_1$, output $COOR_X$ will be $COOR_0$ and min[1:0] will be 01_b. A hierarchy of the 2-input comparator unit builds an N-input comparator unit (❸ in Figure 14). For example, 4-input comparator unit in Figure 15 (b) takes four input coordinates. Each of the two 2-input comparator units outputs a smaller coordinate from their two inputs and its locations as a bit vector. The outputs are delivered to another 2-input comparator unit, which provides the smallest coordinate and its locations. If $COOR_3$ is the smallest, $COOR_Z$ will be $COOR_3$ and min[3:0] will be 1000_b. If there are multiple minimum coordinates (*e.g.*, $COOR_0$ and $COOR_2$), the output bit vector will point all the locations ($COOR_Z = COOR_0 = COOR_2$ and min[3:0] = 0101_b).

So far, we have described the detailed method to build a comparator unit, one of the major components in our conversion unit. We build each individual component of the conversion unit and estimate its area and power consumption in Section 5.3. While we build a fixed-purpose hardware accelerator for the conversion units, there could be many possible implementations based on the demand from applications, which could be extended from our study.

5 EXPERIMENTAL EVALUATION

5.1 Methodology

Dataset: We use real sparse matrices from SuiteSparse Matrix Collection [9]. We filter out matrices with large dimension (> 44k rows) to fit dense B and C matrix of the same dimension in the GPU main memory. We also ignore the matrices with small dimension (< 4k rows) to let every SM get at least a single subproblem. Note this does not limit the generality; we test more than 3,500 different matrices without any biases, and the tested matrix set have divergent non-zero distribution and density. We further discuss techniques to apply our idea to larger matrices that do not fit in the GPU's main memory in Section 6. The small matrices may be better processed by CPUs because they cannot fully utilize GPU resources and also

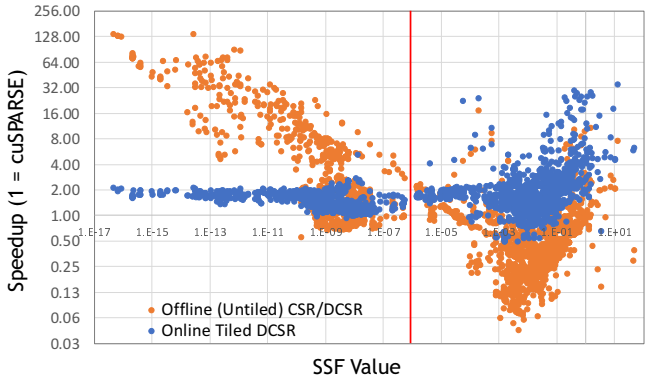

Figure 16: Heuristic vs. Speedup over cuSPARSE [23].

the dataset can possibly fit in CPU caches. We use 32-bit floating point datatype for multiplication and assign random values if a matrix does not have values (*e.g.* matrices which describe connectivity of nodes).

HW/SW Baseline: We use a server-class NVIDIA GV100 GPU. GV100 has 5,120 FP32 CUDA cores which operate at 1,530 MHz, up to 96 KB shared memory / SM, 6,144 KB L2 cache, and 16 GB HBM2 main memory with 4,096-bit bus providing 870 GB/s bandwidth. GV100 has die size of 815 mm^2. We compile our CUDA software using nvcc in CUDAToolkit v9.0 with -O3 compile option. We use B tile dimension of 64×64 to fully utilize the shared memory of an SM. We compare the results and performance with cuSPARSE v9.0 [23]. We verify our implementation can produce the same output as cuSPARSE.

Bandwidth Simulation: Our proposed method enables a small CSC memory footprint with the computation efficiency of DCSR. We simulate the CSC bandwidth using our Tiled DCSR kernel on the GPU by fetching random values from L2 cache using inline assembly for the difference of the metadata footprint. By reading from L2, we simulate the memory footprint of CSC and the Xbar bandwidth requirement of tiled DCSR or untiled DCSR.

5.2 Performance

Figure 16 shows speedup over cuSPARSE and its correlation with the heuristic value. In this evaluation, we assume a realistic situation where the input sparse matrices are provided either in (untiled)CSR/DCSR or CSC format. We include untiled DCSR because converting from CSR to (untiled) DCSR is straightforward and its cost is not overwhelming, while it provides significant performance improvement for highly sparse matrices (lower SSF value). The orange dots plot speedup for the CSR/DCSR baseline (C-stationary). We plot the better results from CSR and DCSR to show its upper-bound for each matrix. We observe all of the matrices with lower SSF value than SSF_th prefer DSCR. The blue dots plot speed up based on our online conversion approach from CSC to (tiled) DCSR (B-stationary). We observe it generally improves performance for higher SSF value as expected in the analytical model (Section 3). Moreover, it can provide performance benefit even for the matrices with lower SSF value, although the speedup is not as high as untiled DCSR (C-stationary). Thus, under a circumstance where we cannot characterize the input beforehand, we can blindly use CSC format as the input and apply our method. This all-tiling approach will provide 1.63× performance benefit compared to the baseline.

On the other hand, under a situation where we can select the input format from CSR or CSC and the algorithm from C- or B-stationary based on profiled SSF value, we can synergistically benefit from both approaches (orange dots on the left and blue dots on the right of Figure 16), achieving the best result (2.26×). This is very close to the optimal speedup of 2.30×, where we assume we can perfectly classify matrices. Online tiling reduces the effect of DCSR metadata (which is not included in our heuristic) and makes it closer to the analytical model in Section 3, providing more than 96% categorizing accuracy. We observe around 95% of input matrices see performance improvement under our scheme.

We also evaluate the performance of tiled DCSR converted offline. By introducing offline tiled-DCSR (B-stationary) in combination with offline DCSR (C-stationary), we observe 2.03× speedup on average, using the same SSF. This does not consider the offline tiling cost of DCSR, thus providing optimistic results. This offline conversion cost is not trivial and it often takes more time than the main SpMM kernel. Our approach brings the tiling benefits without incurring the offline tiling costs in terms of both processing time and storage.

We notice that some matrices do not achieve performance benefits from our proposal. There can be multiple causes that interact to produce inefficiency. For example, there could be imbalances of non-zero distribution across rows, which causes longer critical latency for a group of threads in a warp. This row-level non-zero skew can be addressed by merge based approach [21]. Also, too many non-zero elements per row could be balanced by changing how to allocate warps and threads per row (*e.g.* partial warps [22]). These approaches are orthogonal to our proposal and can be applied to both B- and C-stationary.

5.3 Area and Energy Consumption

We evaluate the additional area and energy consumption to integrate our proposed mechanism described in Section 4.2.2, which generates a *64* column wide DCSR matrix from a CSC matrix. We build detailed circuit models of the comparator, buffer, and control logic by using the standard cell library of TSMC 16 nm process technology [35]. The energy overhead is estimated based on the worst case input output combination, which requires the highest energy. We use CACTI [13] to estimate the performance and overhead for the internal buffer memory. We place our data transformation unit every HBM2 pseudo channels (64 pseudo channels in total).

Throughput demand. We first estimate the requested throughput of our mechanism. Our goal is providing higher throughput than data traffic from DRAM, thereby always providing better performance than the baseline. The smallest throughput that our mechanism can provide is the case of generating only *one* element per one DCSR row, which requires *i)* 8-byte input data (one index and one single-precision floating-point value) or *ii)* 12-byte input data (one index and one double-precision floating-point value). One pseudo channel of HBM2 supports 13.6 GB/s throughput, which can deliver 8-byte data every 0.588 ns and 12-byte data every 0.882 ns. To match these throughput, we build multiple stages of pipeline, the largest latency of which is smaller than the cycle time of 0.588 ns. Our

estimation shows that the longest latency in our pipeline is 0.339 ns, which we observed from a stage of coordinate comparator.

Internal buffer demand for hiding the latency for supplying per-column data. With integrating our multi-stage pipeline, the throughput of our proposed logic can fit in the target throughput. To enable the maximum throughput of conversion engine, all 64 column entries need to be supplied on time, which takes *i)* 3.3 ns for figuring out which column entries are consumed and needs to be delivered from memory (❸ and ❹ in Figure 14), and *ii)* 15 ns for accessing DRAM (CL: Column address strobe latency, the time delay between issuing DRAM command and delivering data from DRAM). This high latency for supplying per-column data significantly reduces overall throughput. To avoid this, we add a prefetch buffer that is fast (less than 0.588 ns target latency) and large enough to compensate for the long latency. To determine the capacity of the prefetch buffer, we conduct a case study for the highest per-column data demand, in which we consume one 8-byte element (12 bytes in double-precision) from the same column entry every 0.588 ns cycle time (0.882 ns in double-precision). Based on these, we determine the size of our internal memory structure as 256 bytes per column (16 Kbyte in 64 column wide DCSR) to be able to hide 18.8 ns in both single-precision and double-precision cases. This internal buffer is enough to hide the latency for supplying per-column data demand, even in the case of rare 100% DRAM channel utilization.

Area and energy consumption. In our estimation, the area for one transformation unit is 0.077 mm^2. In GV100 with 64 HBM2 pseudo channels, the total area for our transformation units is 4.9 mm^2, which is 0.6% of the overall chip (815 mm^2). We evaluate the highest power consumption, which is the case of always generating a single element DCSR row. Generating one single element DCSR row with 8-byte data (4-byte index and 4-byte value) consumes 6.29 pJ every 0.588 ns (7.09 pJ every 0.882 ns for 8-byte data value), leading to 0.68 W (0.51 W for 8-byte value) with a fully loaded memory system (870 GB/s). Considering that our evaluated system consumes maximum 250 W, the energy overhead of our mechanism is trivial even in the worst case. Indeed, the peak power of our engine is 0.27% of the TDP (and 2.96% of the idle power). Clearly, our average speedup (2.26x) more than amortizes for the added power and energy. Moreover, the processing time of the engine is smaller than the kernel processing time of each SM, thus it can mostly be hidden. The head and tail effects are negligible compared to the overall processing time. Furthermore, it can be clock-gated when not in use. Thus, no energy cost is added to the normal GPU operation.

The cost of the transform engine is proportional to the memory bandwidth. Thus a smaller system will also require fewer transformation engines. If we consider NVIDIA's 284 mm^2 TU116 chip with 288 GB/s of aggregate bandwidth across 24 16-bit wide 12 GB/s GDDR6 channels, adding 24 transform engines would cost 1.85 mm^2. This is 0.65% of the overall area (similar to the 0.6% overhead for GV100).

6 DISCUSSION

6.1 Data Layout and Load Balancing

A naive data layout of the sparse CSC input may cause memory traffic congestion issue at FB partitions. Figure 17 (left) illustrates

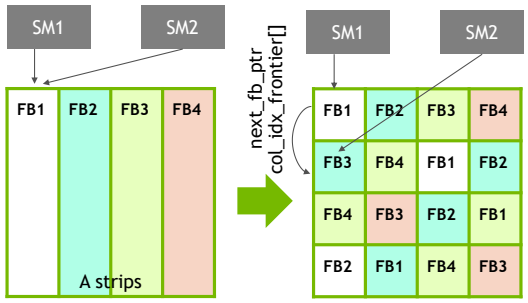

Figure 17: Load balancing issue and tile separation approach.

that each of the strips of sparse A matrix is allocated to one of the four FB partitions. As in the figure, allocating an entire strip of A in one FB partition causes a camping problem where multiple SMs pound on the same FB partition. Ideally, a shared unified memory would enable SMs to share a strip across SMs, however, in practice, SMs only have a private shared memory, thus with such data layout load balancing issue will inevitably arise.

To address the load balancing problem, we horizontally split the strips into tiles and store the tiles across multiple FB partitions. This allows SMs to access different parts of a strip stored in different FB partition, as illustrated in Figure 17 (right). When an SM advances to process the next tile in a strip, current FB partition returns `next_fb_ptr` and current `col_idx_frontier`. We do not have to store column index offset of a tile in the memory because tiles in a strip are sequentially accessed. However, `next_fb_ptr` and `col_idx_frontier` will cause small increase in the memory footprint. This overhead is small because we do not break up a strip into thousands of segments unlike the stateful CSR-to-DCSR conversion (Section 4).

We analyze the performance impact of switching FB partition by simulating its overhead of the additional memory bandwidth by inserting L2 load instructions. We use synthetic matrices with uniform random distribution and randomly selected matrices from [9], and analyze the execution time normalized to that of cuSPARSE assuming FB partition switching happens for every x non-zero rows, varying x. We observe that the overhead of the additional memory bandwidth adds negligible performance impacts if the number of non-zero tile rows stored in an FB partition is not less than 64. Therefore, we can just split the strip as many as the number of FB partitions, and since we are not required to do random access within a strip, there will be no non-trivial metadata that consumes bandwidth.

Alternatively, we can put the conversion unit in SMs. This also solves the load balancing problem, although it incurs 2× area cost to allow all SMs to have the conversion unit and to increase the buffer size to cover the increased latency for the Xbar traversal etc.

6.2 Towards Large Scale SpMM

SpMM for large scale matrices requires huge storage for dense B and C. For example, 2M × 2M dense matrix is as large as 17 TB, and the entire matrix cannot fit in the GPU main memory. For those extremely large matrices, we can divide the work and process each submatrix of the dense matrices by streaming the working data

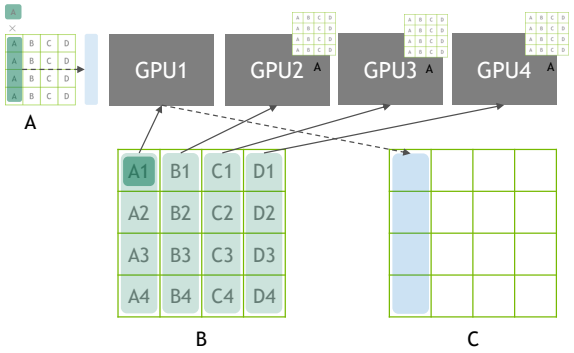

Figure 18: Large scale SpMM in a multi-GPU system.

set from the system memory to GPU's main memory. This can be enabled by using multiple CUDA streams or paging in and out of Unified Virtual Memory (UVM). Further, we can have multiple GPUs working on individual subproblems individually.

Our proposed method can naturally fit in this picture. In a multi-GPU system, the input sparse matrix is shared by all the participating GPUs by making its copies because A is the most space efficient data among the three. Since remote communication for fetching the dense matrices will be the bottleneck, a GPU node will fetch tiles in vertical B strips and calculate complete vertical C strips to minimize the communication. The space efficient CSR/CSC format is beneficial in this context by allowing larger memory space for B and C strips to overlap computation and data streams. Also, by dynamically generating DCSR tiles and its metadata within FB partitions of each GPU, we can provide the aforementioned benefits of CSC's bandwidth efficiency and tiled-DCSR's computation efficiency.

7 RELATED WORK

To the best of our knowledge, this is the first work that demonstrates the feasibility of near-memory data format transformation for SpMM. Below we discuss some of the closely related works.

Sparse Linear Algebra. Since sparse linear algebra is a fundamental operation in many kernels, there has been a large body of works which propose efficient parallel algorithms for heterogeneous architectures. A number of works have investigated efficient sparse data structure and algorithm for SpMV for GPUs [1, 4, 8, 17, 31, 38]. Recently Merrill *et al.* [21] proposed merge-based approach for parallel SpMV. Merge-based approach strictly balances the workload by performing balanced decomposition of loads for non-zero data and row offsets by finding a *merge-path*. Merge-based approach is also employed in [39] for SpMM. These are orthogonal to our approach and can be integrated to provide better performance especially when a skewed non-zero distribution is observed across rows.

Hong *et al.* [12] propose to extract highly clustered row segments from the sparse matrix. The heavy clusters are formatted into Densified CSR (DCSR) and multiplied by submatrices of the dense matrix cached in the GPU shared memory to achieve high data reuse. The remainder of A forms a light and sparse matrix, which is stored in CSR format and processed in output stationary fashion. However, this hybrid approach can suffer from bandwidth to read B matrix multiple times (especially when the B elements accessed by heavy rows segments and light matrix overlap) and format conversion (preprocessing) cost as discussed in Section 3.3, and in [39]. Our technique avoids both these problems.

Compressed Sparse Fiber (CSF) [7, 19, 30] is a sparse data structure for sparse tensors. It is originally designed to efficiently solve Matricized Tensor Times Khatri-Rao Product (MTTKRP), the core kernel of Canonical Polyadic Decomposition (CPD) with Alternating Least Squares. MTTKRP iteratively accesses *modes* (or dimension) of the sparse tensor in different orders, which requires efficient access to *slices* of the tensor in each mode. CSF uses a tree-based structure to make slices accessible in an efficient manner. While CSF is an interesting idea that provides flexibility in access to multi-mode tensors, it is overkill for SpMM of which (slice) access pattern is deterministic and not as complicated.

Sparse General Matrix-Matrix Multiplication (SpGEMM) is another class of sparse workloads, which multiplies two sparse matrices. Nagasaka *et al.* [22] proposes fast SpGEMM algorithm with low memory usage for GPUs. Their algorithm utilizes GPU's shared memory for accumulating partial contributions, combining it with several thread assignment method.

Near-Memory Data Manipulation. While many Processing-In-Memory (PIM) in the past explored single-die integration of logic and DRAM, manufacturing cost and unconventional programming models have imposed limitations on them and made them less practical. Near Data Processing (NDP) moves compute near memory, and reduces the expensive memory communication cost by performing computation locally near memory. TOP-PIM [40] studied the impact of NDP in the logic layer of 3D-stacked memory for heterogeneous computing of CPU and GPU. Kim *et al.* [15] proposes partitioned execution mechanism that enables NDP for data distributed across multiple standardized stacked memory with NDP layer. On contrary to those NDP approaches which perform computation locally to exploit internal memory bandwidth and reduce external bandwidth, we still use GPU cores to execute the main SpMM kernel. Since our approach adds metadata for efficiency, applying those NDP approaches is not desired because of its negative impact on the external memory bandwidth.

Design space for smart memory controller embodies another class of NDP. Impulse [6] adds another layer of address translation in the memory controller to remap memory access to distinct cache lines in physical pages (*e.g.* elements in a diagonal of a dense matrix) into a dense cache line. By gathering and densifying information conveyed by a cache line, it reduces wasted bus bandwidth and the effect of false sharing problem. Data Reorganization Engine [10] adopts similar gather/scatter engine on a 3D stacked memory. Our approach is different in that it augments the information for better performance by performing data structure and algorithm specific data manipulation. We also exploit large Xbar bandwidth available internally in GPU die, which does not form a bottleneck for applications with large memory footprint and irregular access patterns.

8 CONCLUSION

We propose a near-memory data transformation technique for efficient SpMM. We conduct a detailed analysis of SpMM workload using latest GPU, focusing on data locality, computation order, data

format, and computation / storage efficiency. We observe a combination use of C-stationary with DCSR and B-stationary with tiled DCSR provides best computation efficiency for matrices with a variety of density and non-zero distribution. We also develop an analytical model and a heuristic function to select the algorithms. While the tiling techniques offer opportunities for better job distribution and locality exploitation utilizing shared memory in SMs, storing pre-processed tiled data causes non-trivial increase in metadata storage overhead of sparse matrices which are most frequently accessed from DRAM as well as preprocessing cost. Our proposal converts the sparse data format from a storage friendly CSC to computationally friendly tiled DCSR in a hardware assisted manner at FP partition in GPU. We also propose techniques for better memory load balancing and larger problem scale targeting multi-GPU systems. Our approach achieves 2.26× better performance compared to cuSPARSE.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Akrem Benatia, Weixing Ji, Yizhuo Wang, and Feng Shi. 2016. Sparse matrix format selection with multiclass SVM for SpMV on GPU. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 496–505.

[5] Carmen Campos and Jose E Roman. 2012. Strategies for spectrum slicing based on restarted Lanczos methods. *Numerical Algorithms* 60, 2 (2012), 279–295.

[6] J. Carter, W. Hsieh, L. Stoller, M. Swanson, , E. Brunvand, A. Davis, , R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. 1999. Impulse: building a smarter memory controller. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*. 70–79. https://doi.org/10.1109/HPCA.1999.744334

[7] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30. https://doi.org/10.1145/3276493

[8] Mayank Daga and Joseph L Greathouse. 2015. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *2015 IEEE 22nd International conference on high performance computing (HiPC)*. IEEE, 64–74.

[9] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1.

[10] Maya Gokhale, Scott Lloyd, and Chris Hajas. 2015. Near Memory Data Structure Rearrangement. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 283–290. https://doi.org/10.1145/2818950.2818986

[11] 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).

[12] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. 2018. Efficient Sparse-Matrix Multi-Vector Product on GPUs. 18 (2018). https://doi.org/10.1145/3208040.3208062

[13] HP. [n. d.]. CACTI, An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. https://www.hpl.hp.com/research/cacti.

[14] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2016. A high-performance parallel algorithm for nonnegative matrix factorization. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 9.

[15] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. 2017. Toward Standardized Near-data Processing with Unrestricted Data Placement for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*. ACM, New York, NY, USA, Article 24, 12 pages. https://doi.org/10.1145/3126908.3126965

[16] 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.

[17] Kornilios Kourtis, Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. 2011. CSX: an extended compression format for spmv on shared memory systems. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 247–256.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[19] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: hierarchical storage of sparse tensors. In *SC '18 Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*.

[20] 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.

[21] Duane Merrill and Michael Garland. 2017. Merge-Based Parallel Sparse Matrix-Vector Multiplication. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE, 678–689. https://doi.org/10.1109/SC.2016.57

[22] Y. Nagasaka, A. Nukada, and S. Matsuoka. 2017. High-Performance and Memory-Saving Sparse General Matrix-Matrix Multiplication for NVIDIA Pascal GPU. In *2017 46th International Conference on Parallel Processing (ICPP)*. 101–110. https://doi.org/10.1109/ICPP.2017.19

[23] NVIDIA. 2018. cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html.

[24] National Institute of Standards and Technology. 2013. MatrixMarket: File Formats. https://math.nist.gov/MatrixMarket/formats.html.

[25] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford InfoLab.

[26] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. 2016. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409* (2016).

[27] Xavier Pinel and Marc Montagnac. 2013. Block Krylov methods to solve adjoint problems in aerodynamic design optimization. *AIAA journal* 51, 9 (2013), 2183–2191.

[28] Ahmet Erdem Sarıyüce, Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2015. Regularizing graph centrality computations. *J. Parallel and Distrib. Comput.* 76 (2015), 106–119.

[29] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

[30] Shaden Smith and George Karypis. 2015. Tensor-matrix Products with a Compressed Sparse Tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms (IA3 '15)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2833179.2833183

[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*. ACM, 13.

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

[33] Alexandre Tiskin. 2001. All-pairs shortest paths computation in the BSP model. In *International Colloquium on Automata, Languages, and Programming*. Springer, 178–189.

[34] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 11.

[35] S. Y. Wu, C. Y. Lin, M. C. Chiang, J. J. Liaw, J. Y. Cheng, S. H. Yang, S. Z. Chang, M. Liang, T. Miyashita, C. H. Tsai, C. H. Chang, V. S. Chang, Y. K. Wu, J. H. Chen, H. F. Chen, S. Y. Chang, K. H. Pan, R. F. Tsui, C. H. Yao, K. C. Ting, T. Yamamoto, H. T. Huang, T. L. Lee, C. H. Lee, W. Chang, H. M. Lee, C. C. Chen, T. Chang, R. Chen, Y. H. Chiu, M. H. Tsai, S. M. Jang, K. S. Chen, and Y. Ku. 2014. An enhanced 16nm CMOS technology featuring 2nd generation FinFET transistors and advanced Cu/low-k interconnect for low power and high performance applications. In *IEDM*.

[36] 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.

[37] 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, 6-7 (2013), 280–290.

[38] 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.

[39] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*. Springer, 672–687.

[40] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM, New York, NY, USA, 85–98. https://doi.org/10.1145/2600212.2600213