MSREP: A Fast yet Light Sparse Matrix Framework for Multi-GPU Systems

Jieyang Chen
chenj3@ornl.gov
Oak Ridge National Laboratory

Chenhao Xie
chenhao.xie@pnnl.gov
Pacific Northwest National Laboratory

Jesun S Firoz
jesun.firoz@pnnl.gov
Pacific Northwest National Laboratory

Jiajia Li
jiajia.li@ncsu.edu
North Carolina State University

Shuaiwen Leon Song
shuaiwen.song@sydney.edu.au
University of Sydney

Kevin Barker
kevin.barker@pnnl.gov
Pacific Northwest National Laboratory

Mark Raugas
mark.raugas@pnnl.gov
Pacific Northwest National Laboratory

Ang Li
ang.li@pnnl.gov
Pacific Northwest National Laboratory

ABSTRACT

Sparse linear algebra kernels play a critical role in numerous applications, covering from exascale scientific simulation to large-scale data analytics. Offloading linear algebra kernels on one GPU will no longer be viable in these applications, simply because the rapidly growing data volume may exceed the memory capacity and computing power of a single GPU. Multi-GPU systems nowadays being ubiquitous in supercomputers and data-centers present great potentials in scaling up large sparse linear algebra kernels. In this work, we design a novel sparse matrix representation framework for multi-GPU systems called MSREP, to scale sparse linear algebra operations based on our augmented sparse matrix formats in a balanced pattern. Different from dense operations, sparsity significantly intensifies the difficulty of distributing the computation workload among multiple GPUs in a balanced manner. We enhance three mainstream sparse data formats – CSR, CSC, and COO, to enable fine-grained data distribution. We take sparse matrix-vector multiplication (SpMV) as an example to demonstrate the efficiency of our MSREP framework. In addition, MSREP can be easily extended to support other sparse linear algebra kernels based on the three fundamental formats (i.e., CSR, CSC and COO).

ACM Reference Format:
Jieyang Chen, Chenhao Xie, Jesun S Firoz, Jiajia Li, Shuaiwen Leon Song, Kevin Barker, Mark Raugas, and Ang Li. 2018. MSREP: A Fast yet Light Sparse Matrix Framework for Multi-GPU Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Graphics processing units (GPUs) have become the mainstream and powerful accelerators in the past decade and supported a wide spectrum of applications, such as applications based on direct and iterative solvers, machine learning algorithms, etc. Due to advances in interconnect technology, multiple-GPU systems nowadays have been widely adopted in from the world's fastest supercomputer Summit [4] to NVIDIA's Super-AI DGX systems [2], and even desktop workstations. With a condensed configuration, up to 16 GPUs can be installed in a single compute node with fast interconnect such as NVLink or NVSwitch connecting them. These high-speed interconnects offer much superior data-exchanging rate among GPUs than the conventional PCI-e solution, where data always has to be routed by CPUs. This fundamental change poses a unique opportunity for designing scalable multi-GPU algorithm for large-scale data processing.

Nevertheless, except for deep-learning tasks, few research has been conducted on optimizing performance for a multi-GPU system leveraging these new interconnect. Some work targeted multi-GPUs [5, 9, 24, 30, 39], but due to restricted bandwidth of the PCI-e, they tend to avoid communication whenever possible, undermining the capability of the fast interconnect; a general framework that can fill this gap can be highly beneficial to the community.

To facilitate multi-GPU programming and data sharing among the GPUs, the vendors like NVIDIA has introduced Unified Memory [1] and NVSHMEM technology [3]. Nonetheless, the designing of malleable multi-GPU data structures while achieving workload balance have been left over at the discretion of the programmers. In this work, we target the domain of sparse matrix operations. In particular, we introduce novel data structures for storing sparse data on multi-GPU systems. In addition, we propose a workload balancing technique for multi-GPU sparse kernels. Unlike dense linear algebra kernels, the sparsity feature makes it challenging to achieve balanced workload distribution.

To this end, we consider sparse matrix-vector multiplication (SpMV) kernel. SpMV is one of the most extensively utilized sparse matrix operations in big data analytics and scientific computations. This kernel has been widely studied in many research work [6–8, 12, 17, 21–28, 31–37, 40] in the context of data structure design, performance optimization, compiler implementation, and hardware architecture support, as well as on shared memory, distributed systems, and GPUs. However, to the best of our knowledge, few studies

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

have been carried out in designing SpMV kernel for multi-GPU systems to take the advantage of fast GPU-GPU interconnect. Existing works on SpMV either involve CPUs to orchestrate data transfers or split the whole workload into small, independent tasks to distribute on multiple GPUs without giving careful consideration to the workload imbalance [5, 18]. Moreover, applications with large linear systems make offloading an SpMV kernel on a single GPU infeasible due to its limited memory capacity. While possible solutions of deploying out-of-core or distributed SpMV exist, these solutions suffer from slow CPU-GPU transfer rate or network latency. In these cases, it is challenging to guarantee high performance, especially for memory-bound SpMV kernel. Hence, we take SpMV as the driving kernel to showcase and illustrate our sparse matrix representation framework's efficiency on multi-GPU systems.

In this work, we propose a general framework, namely MSREP, consisting of augmented sparse matrix formats and balanced distribution. We demonstrate that, with our framework, sparse linear algebra kernels, in particular, SpMV, can achieve scalable performance on multi-GPU systems. In addition, our framework can support the existing SpMV kernels based on these formats. Specifically, our contributions are as follows:

- We extended three popular sparse data formats, Compressed Sparse Row (CSR), Compressed Sparse Colunmn (CSC), and coordinate (COO) by storing a part of a sparse matrix with arbitrary start and end positions. The new formats, pCSR, pCSC, and pCOO, requires small additional memory to maintain the metadata and can be converted from the well-known formats swiftly.
- To demonstrate our framework's efficiency on multi-GPU systems, we develop an SpMV kernel for multi-GPU system called `mSPMV`. To achieve better scalability, `mSPMV` leverages our sparse matrix representations on multi-GPUs for efficient workload distribution.
- We evaluate our MSREP on two dense GPU systems: the Summit supercomputer at Oak Ridge National Laboratory and a NVIDIA V100-DGX-1 system. Experiments using matrices from the *SuitSparse Matrix Collection* [14] shows that `mSPMV` can achieve 5.5X speedup using six GPUs on Summit and 6.2X speedup using eight GPUs on the NVIDIA V100-DGX-1 system.

The rest of this paper is organized as follows. In Section 2 we provide background of this work with a discussion about three popular sparse data formats and use SpMV as an example to illustrate workload imbalance due to data distribution. In Section 3, we discuss in detail the design of our sparse matrix representation framework together with the design of the multi-GPU SpMV. We propose several implementation optimizations for SpMV on multi-GPU systems in Section 4. We report our experimental results in Section 5. We summarize our observations in Section 6. We give an overview of related works in Section 7 and draw our conclusion in Section 8.

Table 1: Notation in Algorithms and Formulations.

m	Number of rows in the input matrix.
n	Number of columns in the input matrix.
nnz	Number of non-zero elements in the input matrix.
np	Total number of partitions.

10	0	0	0	-2	0
3	9	0	0	0	3
0	7	8	7	0	0
3	0	8	7	5	0
0	8	0	9	9	13
0	4	0	0	2	-1

Figure 1: A sparse matrix where a large portion are zero elements.

2 BACKGROUND

We use the example matrix in **Fig. 1** to illustrate different storage formats and their SpMV algorithms. We also analyze the imbalance issue on multi-GPU systems for sparse matrix operations. **Table 1** shows the related notations for this paper, for a $m \times n$ matrix with nnz non-zero elements.

Val:	10	-2	3	9	3	7	8	7	3	8	7	5	8	9	9	13	4	2	-1
row_idx:	0	0	1	1	1	1	2	2	2	3	3	3	4	4	4	4	5	5	5
col_idx:	0	4	0	1	5	1	2	3	0	2	3	4	1	3	4	5	1	4	5

Figure 2: Coordinate (COO) sparse matrix format.

Val:	10	-2	3	9	3	7	8	7	3	8	7	5	8	9	9	13	4	2	-1
col_idx:	0	4	0	1	5	1	2	3	0	2	3	4	1	3	4	5	1	4	5
row_ptr:	0	2	5	8	12	16	19												

Figure 3: Compressed Sparse Row (CSR) sparse matrix format.

Val:	10	3	3	9	7	8	4	8	8	7	7	9	-2	5	9	2	3	13	-1
row_idx:	0	1	3	1	2	4	5	2	3	2	3	4	0	3	4	5	1	4	5
col_ptr:	0	2	5	8	12	16	19												

Figure 4: Compressed Sparse Column (CSC) sparse matrix format.

2.1 Mainstream Sparse Matrix Storage Formats

2.1.1 COO Format. Coordinate (COO) format is the most straightforward sparse matrix format by storing only non-zero elements along with their indices and values. **Fig. 2** shows the data structure of the COO format to store the example sparse matrix (**Fig. 1**). Three nnz-sized arrays are used. `val` stores the values of non-zero elements. `row_idx` and `col_idx` store the row and column indices corresponding to each non-zero element.

2.1.2 CSR Format. Compressed Sparse Row (CSR) format is a more compressed format compared to the COO format. **Fig. 3** gives the data structure of the CSR format using the same example sparse matrix (**Fig. 1**). The same with COO, `val` stores the non-zero values and `col_idx` stores their column indies. `row_ptr` compresses row indies by pointing the row starting position.

2.1.3 CSC Format. Compressed Sparse Column (CSC) is similar to the CSR format, which is also compressed but on columns. **Fig. 4** shows the CSC format where `val` stores the non-zero values, `row_idx` stores row indices, and `col_ptr` stores the pointers of

starting position of column indices. The CSC format of a matrix A is the same with the CSR format of its transposed matrix A^T.

2.2 Sparse Matrix-Vector Multiplication

Sparse Matrix-Vector Multiplication (SpMV) operation is the most popular operation of sparse linear algebra and has broad applications. In this work, we take SpMV as an illustration example to show the benefit of our framework MSREP.

Algorithm 1 calculates the SpMV using CSR format. This algorithm loops all rows of a sparse matrix; then for each row, loops all the non-zeros inside. According to the $col\_idx[j]$ index to locate the x value to do the product with non-zero value $val[j]$. All the partial products on row i will be summed and used to update $y[i]$ for the final output. SpMV algorithm based on CSC format is to switch the role of x and y. SpMV algorithm using COO format is not hard to image: only one loop for all the nnz non-zero elements while its corresponding column index to locate the corresponding x element then do the product and update its counterpart output y element by indexing with its row index.

Algorithm 1: CSR-based SpMV: $y = \alpha Ax + \beta y$

```
In     : spares matrix A (m × n): val, col_idx, and row_ptr
In     : dense vector x (n)
In     : scalar α and β
In/Out : dense vector y (m)
for i = 1 to m do
    for j = row_ptr[i] to row_ptr[i + 1] do
        y[i] = α * val[j] * x[col_idx[j]] + β * y[i]
    end
end
```

2.3 Imbalance issue on multi-GPU systems

To motivate our work we show how workload distribution strategies can impact the performance of a sparse matrix operation. Take SpMV as the example, it is generally a memory bound computation on mainstream architectures as its flops-to-bytes ratio is roughly $O(1)$. Thus, the cost of loading input data is a main dominant factor of its performance due to its relatively large memory footprint compared to the two vectors and the potential data reuse due to cache and memory hierarchy [11]. For the input matrix, each element is only used once during the entire computation, so the main cost of SpMV comes from accessing the nonzero elements. For other sparse matrix operations, sparse matrix-sparse vector and sparse matrix times multiple dense vectors have similar behavior with SpMV. Even for sparse matrix-dense/sparse matrix operation, due to the potential data reuse of the right matrix, the memory access of sparse matrix still playing an important role.

For the dense matrix multiplication, a commonly used strategy to distribute elements in the input matrix is simply dividing the matrix into row blocks and then assigning each of them to different GPUs for computation. Distributing rows evenly among GPUs would leads to good workload balancing and performance. However, when applied to sparse matrix, this kind of workload distribution will not work well without considering the sparsity of the matrix. Since calculations on zeros are unnecessary, they are usually omitted, which

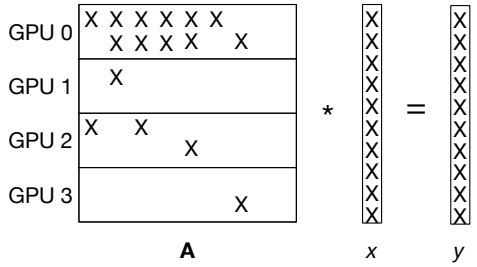


Figure 5: Naive workload distribution of SpMV with non-uniformly distributed input matrix elements.

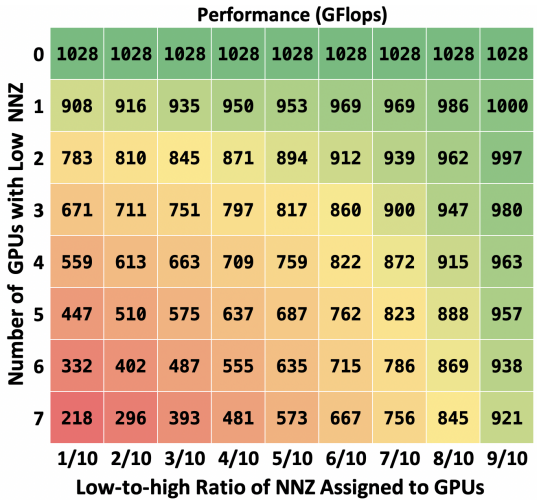


Figure 6: Imbalanced non-zero elements distribution among GPUs can cause performance degradation in SpMV. For example, if 4 of the total 8 GPUs are assigned with only 1/10 of the none-zeros assigned to the other 4 GPUs, the overall performance would reduce to about half (559/1028).

leads to workload imbalance as the number of zeros may varies in between row blocks as shown in **Fig. 5**. For SpMV, the workload of each GPU is proportional to the number of non-zero elements (nnz) rather than the number of rows (m). Workload imbalance in SpMV could greatly impact the overall SpMV performance. **Fig. 6** shows the benchmarking results of using the straightforward distributed strategy in SpMV. We generate input matrices with different kinds of non-zeros element distribution that lead to imbalanced nnz on each GPUs. To simplify, the distribution leads to two kinds of workload among GPUs. One kind of workload has higher number of nnz than the other ones. The ratio of nnz between low-to-high is shown in the x-axis. The test is conducted on a NVIDIA V100-DGX-1 system.

3 MSREP FRAMEWORK

We introduce our MSREP framework in this section with our proposed three enhanced formats, pCSR, pCSC, and pCOO and workload management.

3.1 Challenges

Distributing the input matrix according to nnz (shown in **Fig. 7**) is the most straightforward and efficient way to divide the entire workload in finer granularity to achieve better workload balance. As

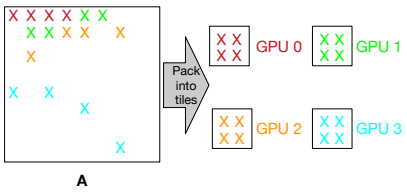


Figure 7: Ideal SpMV workload distribution based on none-zeros.

the focus of this work is exploiting parallelism across GPUs, we choose to leverage existing well-optimized works [6–8, 8, 12, 16, 17, 21, 22, 26–28, 32, 33, 37] to handle workload on each single GPU. Popular state-of-the-art works and libraries support at least one of the three mainstream sparse matrix storage formats: CSR, CSC, and COO. So, compatible with these three format can allow us not only leverage existing state-of-the-art kernels but also benefit from them in the future. However, making the workload distribution efficient and compatible is non-trivial.

3.2 Fine-Grain Workload Distribution

3.2.1 pCSR. We first propose a data format called partialCSR (pCSR). pCSR can easily represent a subset of non-zero elements in a sparse matrix while preserving all necessary element distribution information. It can be converted from CSR format efficiently. Once CSR is partitioned into pCSR, it can be used by CSR-based SpMV and other kernels without overhead.

Fig. 8 shows the data structure of pCSR. To enable efficient conversion to/from CSR format, the most straightforward way is to avoid data copy. So, to represent a subset of non-zero elements in a sparse matrix, we use two index values (i.e., `start_idx` and `end_idx`) to mark the starting and ending position in the non-zero array of CSR. The storage cost is $O(1)$. However, maintaining a local row pointer array is necessary for SpMV kernels to perform correctly. The storage cost is proportional to the number of rows in the partition, which varies depending on non-zero elements distribution. The total cost is no more than $O(m)$. This local row pointer array can be calculated efficiently and will be discussed later. Also, we use the two index values (i.e., `start_idx` and `end_idx`) to mark the starting and ending position in the column index array, so no additional cost is introduced. Since elements in the same row could be distributed into multiple pCSRs, we also maintain a flag (i.e., `start_flag`) to mark if the first row maintained by the current pCSR is partial or not. The last row does not need to be flagged as it can be inferred from the `start_flag` of the next pCSR. Finally, for merging multiple pCSR into one CSR it is necessary to maintain two indices that stores the start and end row index in the global view.

Algorithm 2 shows how to convert CSR to pCSR format. The main cost comes from searching the start and end indices using Binary Search (i.e., $O(log(m))$) and computing the local row pointers (i.e., $O(pA[i].end\_row - pA[i].start\_row)$). The total cost is $O(np * log(m) + m)$. The former one can be efficiently done on CPUs and the local row pointers can be computed using GPUs. We will show time cost comparison of the partitioning with and without GPUs. Finally, each individual partition can be generated independently so the partitioning process can be efficiently parallelized.

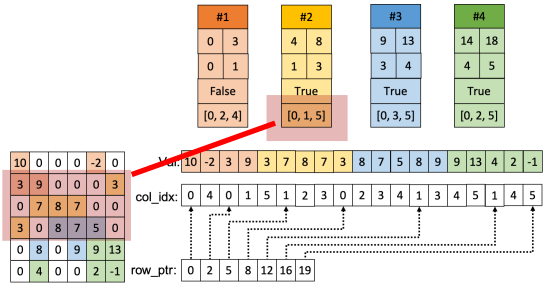


Figure 8: Example of partitioning a matrix into four parts using pCSR.

Algorithm 2: Converting from CSR to pCSR

```
In  : CSR matrix A
In  : Number of Non-zeros in A: nnz
In  : Number of pCSRs: np
Out : pCSR matrix pA[np]
1  for i = 1 to np do
2      pA[i].start_idx=⌊i * nnz/np⌋
3      pA[i].end_idx=⌊(i + 1) * nnz/np⌋ − 1
4      pA[i].start_row=BinarySearch(A.row_ptr,
         pA[i].start_idx)
5      pA[i].end_row=BinarySearch(A.row_ptr, pA[i].end_idx)
6      if pA[i].start_idx > A.row_ptr[pA[i].start_row] then
7          pA[i].start_flag = True
8      else
9          pA[i].start_flag = False
10     end
11     for j = 1 to pA[i].end_row - pA[i].start_row do
12         pA[i].row_ptr[j] = A.row_ptr[pA[i].start_row + j] -
             pA[i].start_idx
13     end
14 end
```

Algorithm 3 shows how to launch CSR-compatible SpMV kernel using pCSR format. We can see pCSR can be converted to CSR without overhead when invoking SpMV kernels. This ensures that all existing and future CSR-compatible SpMV kernel on single GPU can use our framework. **Line 9 - 17** shows how to correctly merge a series of partial results into a final result. We will further discuss how it can be efficiently done in section 4.

3.2.2 pCSC. Similar to pCSR, we also propose partialCSC (pCSC) for partitioning a sparse matrix that is stored in CSC format as shown in **Fig. 9**. Algorithm 4 shows how to efficiently convert CSC to pCSC format. It is easy to see that the overall cost of the conversion is similar to pCSR format. The total cost is $O(np * log(n) + n)$. The algorithm can also be efficiently parallelized. Algorithm 5 shows how to use pCSC in CSC-based SpMV kernels. Again, we will discuss how to do efficient result merging in section 4.

3.2.3 pCOO. Finally, we propose partialCOO (pCOO) format for partitioning COO format-based sparse matrices. To void the cost of element reordering, we choose to generate partitions by dividing the input into consecutive non-zero elements. Partitioning COO is slightly different than CSR and CSC since the elements can be sorted

Algorithm 3: Using pCSR on CSR-based SpMV kernels

```
In      : CSR matrix A
In      : dense vector x (n × 1)
In      : scalar α and β
In/Out  : dense vector y (m × 1)
In      : Number of pCSRs: np
In      : pCSR matrix pA[np]
Allocate space to hold partial results py[np]
/* Run in parallel on multi-GPU */
for i = 1 to np do
    val = A.csr[pA[i].start_idx]
    row_ptr = pA[i].row_ptr
    col_idx = A.col_idx[pA[i].start_idx]
    Launch: py[i]=<csrSpMVKernel>(val, row_ptr, col_idx)
end
for i = 1 to np do
    if pA[i].start_flag then
        tmp = y[pA[i].start_row]
    end
    memcpy: y[pA[i].start_row] ← py[i]
    if pA[i].start_flag then
        y[pA[i].start_row] -= tmp * β
    end
end
```

Algorithm 4: Converting CSC to pCSCs

```
In  : CSC matrix A
In  : Number of pCSCs: np
In  : pCSC matrix pA[np]
for i = 1 to np do
    pA[i].start_idx=⌊i ∗ nnz/np⌋
    pA[i].end_idx=⌊(i + 1) ∗ nnz/np⌋ − 1
    pA[i].start_col=BinarySearch(A.col_ptr, pA[i].start_idx)
    pA[i].end_col=BinarySearch(A.col_ptr, pA[i].end_idx)
    if pA[i].start_idx > A.col_ptr[pA[i].start_col] then
        pA[i].start_flag = True
    else
        pA[i].start_flag = False
    end
    for j = 1 to pA[i].end_col - pA[i].start_col do
        pA[i].col_ptr[j] = A.col_ptr[pA[i].start_col + j] -
          pA[i].start_idx
    end
end
```

or unsorted. The whether or not the COO sorted will matters to how much information we would be able to know about a partition. If the elements are sorted, for example sorted by rows, then it is possible to efficiently find the start and end row index correspond to the partition so that we can locate corresponding row of result vector it will calculate and facilitate the partial merging process. However, if the elements are unsorted, we can only assume that the elements

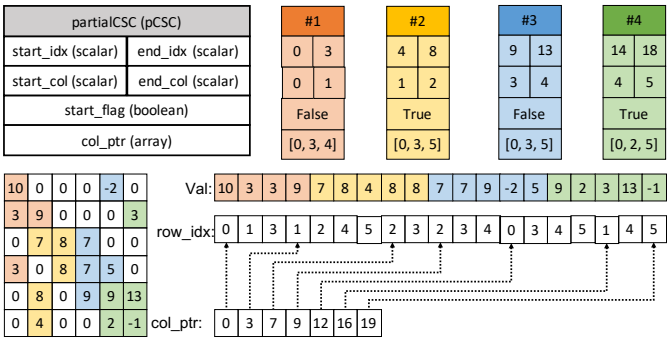


Figure 9: Example of partitioning a matrix into four parts using pCSC.

Algorithm 5: Launching CSC-based SpMV kernel using pCSC

```
In      : CSC matrix A
In      : dense vector x (n × 1)
In      : scalar α and β
In/Out  : dense vector y (m × 1)
In      : Number of pCSCs: np
In      : pCSC matrix pA[np]
Allocate space to hold partial results py[np]
/* Run in parallel on multi-GPU */
for i = 1 to np do
    val = A.csc[pA[i].start_idx]
    col_ptr = pA[i].col_ptr
    row_idx = A.row_idx[pA[i].start_idx]
    Launch: py[i]=<cscSpMVKernel>(val, col_ptr, row_idx)
end
for i = 1 to np do
    sum_y += py[i]
end
y =
```

in a particular partition can spread among the entire matrix without knowing row or column range. This could bring extra memory cost for storing partial results and time for merging partial results. For simplicity, we assume the elements are sorted by rows in this paper. Algorithm 6 shows how to efficiently convert COO to pCOO format. It is easy to see that the overall cost is $O(np * log(m))$ assuming the non-zeros are already sorted by the row index. If sorted by column index the cost is $O(np * log(n))$. The algorithm can be efficiently parallelized into np individual tasks. Algorithm 7 shows how to use pCOO in COO-based SpMV kernels. Again, we will discuss how to do efficient result merging in section 4.

3.3 Managing SpMV Workload on Multi-GPUs

To efficiently manage multiple GPUs at the same time, we use one dedicated CPU thread to manage one GPU. Each thread is responsible for generating the workload partition for the corresponding GPU. If NUMA optimization (will be discussed in section 4.2) is enabled, threads on each NUMA node will also elect one representative thread to handle workload partitioning among NUMA nodes. Partial results

Algorithm 6: Converting COO to pCOOs

```
In  : COO matrix A
In  : Number of pCOOs: np
In  : pCOO matrix pA[np]
for i = 1 to np do
    pA[i].start_idx=⌊i ∗ nnz/np⌋
    pA[i].end_idx=⌊(i + 1) ∗ nnz/np⌋ − 1
    pA[i].start_row=BinarySearch(A.row_ptr,
      pA[i].start_idx)
    pA[i].end_row=BinarySearch(A.row_ptr, pA[i].end_idx)
    if pA[i].start_idx > A.row_ptr[pA[i].start_row] then
        pA[i].start_flag = True
    else
        pA[i].start_flag = False
    end
end
```

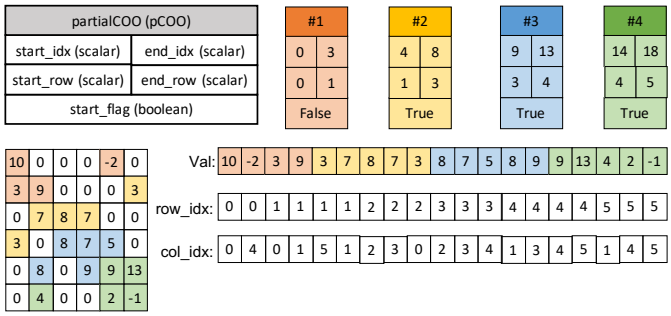


Figure 10: Example of partitioning a matrix into four parts using pCOO (sorted by row).

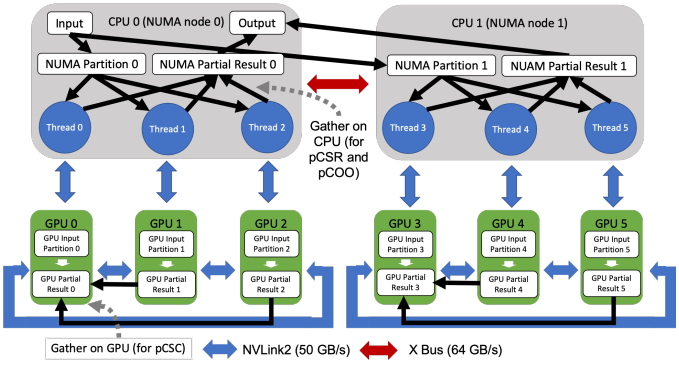


Figure 11: Managing SpMV workload on a compute node on Summit

will be gathered on CPUs for pCSR and pCOO or GPUs for pCSC. Details about partial result gathering will be discussed in section 4.3. **Fig. 11-12** show how the SpMV workload is managed among the 6 GPUs on the computing node on Summit and 8 GPUs on the NVIDIA V100-DGX-1 system.

4 IMPLEMENTATION OPTIMIZATIONS

In this section we cover several issues related to the implementations that are critical for achieving good performance and scalability.

Algorithm 7: Launching COO-based SpMV kernel using pCOO

```
In     : COO matrix A
In     : dense vector x (n × 1)
In     : scalar α and β
In/Out : dense vector y (m × 1)
In     : Number of pCOOs: np
In     : pCOO matrix pA[np]
Allocate space to hold partial results py[np]
/* Run in parallel on multi-GPU */
for i = 1 to np do
    val = A.coo[pA[i].start_idx]
    col_idx = A.col_idx[pA[i].start_idx]
    row_idx = A.row_idx[pA[i].start_idx]
    Launch: py[i]=<cooSpMVKernel>(val, col_idx, row_idx)
end
for i = 1 to np do
    if pA[i].start_flag then
        tmp = y[pA[i].start_row]
    end
    memcpy: y[pA[i].start_row] ← py[i]
    if pA[i].start_flag then
        y[pA[i].start_row] -= tmp * β
    end
end
```

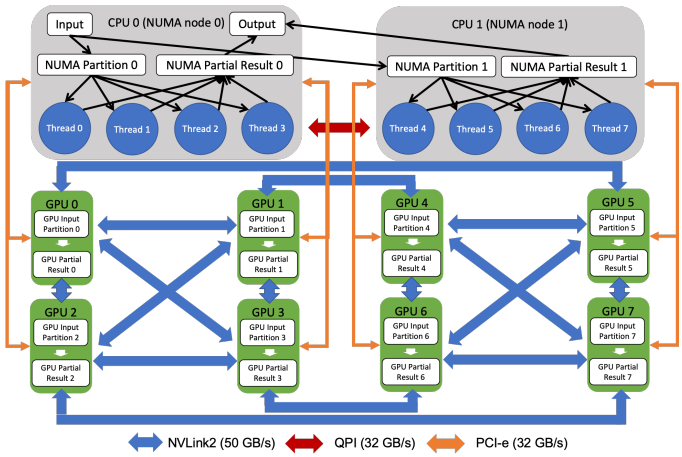


Figure 12: Managing SpMV workload on NVIDIA V100-DGX-1

4.1 Workload partition

As will be seen in the evaluation, the workload partition can introduce considerable overhead up to 85% on the tested matrices. Since each partition can be independently generated as discussed in Section 3.2, we parallelize the partition process through multi-threading – each thread is dedicated for a GPU. Additionally, we offload the most expensive workload to GPUs as specially designed kernels, e.g., the calculation of the local row pointers for CSR, column pointer for CSC, and row index array for COO. As data movement from CPU to GPUs is inevitable, this will not incur extra overhead.

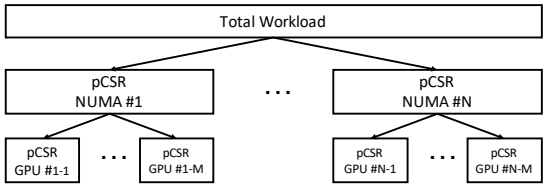


Figure 13: Two Level Workload Partition.

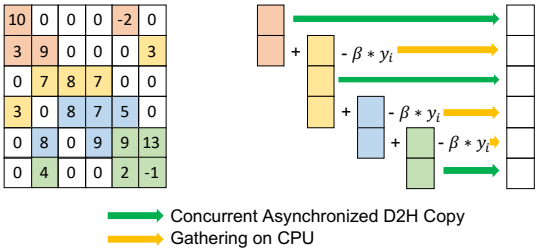


Figure 14: Result Merging for pCSR and pCOO (sorted by row).

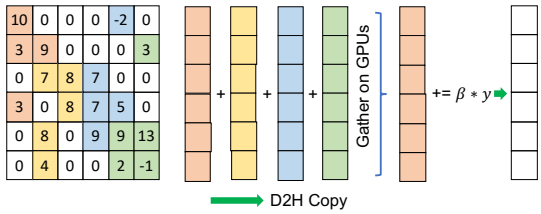


Figure 15: Result Merging for pCSC and pCOO (sorted by column).

4.2 Handling NUMA Effect

Since SpMV is a memory bound computation, the cost of data movement usually dominates the overall cost of the operation. So, efficient copy of the partitioned workload represented by pCSR, pCSC, or pCOO to each of the GPU memory from the CPU memory is specially important. Dense GPU nodes usually partition the GPUs among several NUMA nodes. For example, on Summit six GPUs are partitioned among two NUMA nodes on each computing node as shown in **Fig. 11**. If the workload partitions are placed naively (i.e., on one NUMA node), then it is difficult to scale SpMV beyond three GPUs. This is mainly limited by both the CPU memory throughput within on NUMA node and the inter-connection speed in between NUMA nodes since GPUs on a different NUMA node need to fetch data through the inter-connection (e.g., X Bus on Summit).

In this work, we design `mSpMV` to consider the NUMA effect to make sure pCSR, pCSC, or pCOO partitions are placed in among different NUMA nodes. The placement strategy is to place the number of workload partitions proportional to the number of GPUs on each NUMA node. This is done efficiently using our two level partitioning strategy. As shown in **Fig. 13**, the first level partitions the workload among NUMA nodes and second level partition among GPUs. The two level partitioning strategy enables the work of partitioning itself parallelizaible.

4.3 Merging Partial Results

Another issue that can impact the of performance of SpMV on multi-GPU system is how partial results are merged into the final result. The partitioning format used can impact how the partial results can be merged. Basically they can be classified into two categories: row-based partitioning and column-based partitioning.

Row-based partitioning such as pCSR and pCOO (sorted by row) assigns consecutive rows to a single partition. So, the result of each partition is a segment of the final result vector except for the element at each end, which may be partial result if the current partition share same rows with others. To optimize the merging process, we use GPU-CPU copy to directly copy the non-overlapping result to the final position on the CPU memory and let CPU handle overlapping elements. It brings relative low performance impact since (1) Memory copy can be done concurrently; (2) Since the overlapping issue only need to be handled np times.

Column-based partitioning such as pCSC and pCOO (sorted by column) assigns consecutive columns to a single partition. So, the result of each partition is a vector with the same dimension as the final result but each element in the vector is partial result. To optimize the merging process, we first let all GPUs gather their partial results to one GPU and then copy the result back to CPU.

5 EXPERIMENTAL EVALUATION

In this section, we report our experimental results and discuss our observations in detail.

5.1 Evaluation Platform

We evaluate our framework on two multi-GPU platforms: the Summit supercomputer at Oak Ridge National Laboratory and an NVIDIA V100-DGX-1 system. On Summit, one computing node is used for computation. Each node is equipped with 6 Nvidia Tesla V100 GPUs with 16 GB memory on each GPU and two IBM POWER9 CPUs with 512 GB memory. GPU-GPU and CPU-GPU are inter-connected via NVLinks. CPU-CPU are inter-connected via X-Bus. On the DGX-1 system, the whole system is used for computation. The DGX-1 system is equipped with 8 Nvidia Tesla V100 GPUs with 16 GB memory and two 20-Core Intel Xeon E5-2698 v4 with 512 GB memory. GPUs are inter-connected via NVLinks. CPU-GPU are inter-connected via PCIe. CPUs are inter-connected via QPI.

We compiled our framework with GCC 7.4.0 and CUDA 10.1.168. We employ OpenMP (version 4.5) for spawning tasks on the GPUs For single-GPU kernel execution, we utilize the CSR-based SpMV kernel in cuSparse. If inputs are in CSC format, the kernel is invoked with transpose on so as to avoid format conversion. For COO based inputs, a GPU-based COO-to-CSR conversion kernel is invoked first before the SpMV kernel. We include the execution time of both conversion step and computation step in our reported time.

5.2 Selected Matrices

Table 2 lists the set of sparse matrices we use in our evaluation, collected from the SuitSparse Matrix Collection [14]. The matrices are ordered by the number of non-zero elements. When selecting the matrices, we choose matrices with strong power-law characteristics (skewed degree distribution) [29]. Such pattern is commonly observed in social networks and web graphs. As reported by Yang et al. [39], the number of non-zeros in the columns of these matrices follow a power-law distribution. Hence, we apply this rule to locate these matrices. The distribution in the form: $P(k) \sim k^{-R}$ is considered to follow power law. Here, R is called the exponent of

Table 2: List of sparse matrices used for evaluation.

Matrix	row × col	nnz	R
mouse_gene	45K × 45K	28M	1.03
wb-edu	9M × 9M	57M	2.13
com-LiveJournal	3M × 3M	69M	2.40
hollywood-2009	1M × 1M	113M	1.92
com-Orkut	3M × 3M	234M	2.13
HV15R	2M × 2M	283M	3.09

the power law and is calculated from the distribution of non-zero elements, k denotes the number of non-zero elements per column. Usually, choosing a value for R within the interval $[1, 4]$ correlates to strong phenomenon of power law [25].

5.3 Evaluation Methodology

To evaluate our multi-GPU SpMV design, we implement the following versions:

- **Baseline** is a simple multi-GPU SpMV that we consider as the baseline for comparison purpose. In the baseline version, the input matrix is partitioned in either row blocks (for CSR and COO) or column blocks (for CSC) without considering the distribution of non-zero elements. Moreover, no multi-threading is involved for partitioning the workload or managing the GPUs. Both workload partitioning and partial result merging are done on the CPU without optimization. In addition, no NUMA-aware optimization has been applied.
- **p*** refers to the implementation that leverages our pCSR, pCSC, and pCOO datastructures to achieve workload balance. Multi-threading is used for partitioning the workload, merging the result, and managing the GPUs in parallel. However, no optimization is performed.
- **p*-opt** is based on p*. All implementation optimizations are applied to this implementation.

5.4 Partitioning Overhead

We report partitioning overhead as the percentage of total execution time spent in partitioning the input matrix and distributing the partitions to the GPUs. **Fig. 16** shows the overhead of workload partitioning on Summit and DGX-1. In the baseline implementation, the input matrix is partitioned into row/column blocks. With CSR and CSC format, the main cost comes from calculating the local row/column pointers. However, calculating row/column pointers takes only constant time ($O(m)$ or $O(n)$). If the size of the matrix is large, partitioning it on CPUs can be time consuming. For example, large matrix partitioning incurs 3.8% - 9% overhead on Summit and 0.5%-2.3% on DGX-1. For COO, the partitioning is done by searching the row pointer array and finding the start and end positions of each row block. The most expensive part for partitioning COO comes from calculating the right row index array. Compared to the CSR and CSC representation, this operation is more expensive ($O(nnz)$ compared to $O(m)$ or $O(n)$ for CSR or CSC). From Fig. 16, we observe 72% - 85% overhead on Summit and 38% - 62% on DGX-1 for COO partitioning.

Compared to the baseline partitioning, our pCSR, pCSC, and pCOO (shown as p*) do not incur any extra partition overhead. By parallelizing the partitioning process, we achieve good speedup with increasing number of GPUs. However, partitioning these data structures still encounter high overhead due to the involvement of the CPUs.

On the other hand, from **Fig. 16**, we can see that, p*-opt gains significant benefit by offloading time consuming part on to the GPUs as mentioned in section 4.1 and thus reducing the overhead. By applying this optimization, partitioning overhead was reduced to less than 2% for most cases.

5.5 Partial Results Merging Overhead

Fig. 22 reports the time taken to merge the partial results from each GPU. Since the baseline design partitions the matrix into row/column block, the cost for merging results based on CSR and COO involves copying non-overlapped segments of the result vector only. Hence, the overhead is relative low. For CSC representation, each partial result is a vector with the same dimensions as the final result vector, thus need to be added together. For this reason, in this case, the execution time increases linearly with the number of partitions (GPUs).

On the other hand, both pCSR and pCOO require handling overlapping rows, which results in higher cost for merging partial results. pCSC has the same overhead as the baseline design while merging partial results.

By using GPUs to accelerate the merge step, we are able to reduce the overhead to less than 3.8% for CSR, 9% for CSC, and 17% for COO. For some cases, optimized merging process brings higher overhead since total execution time is reduced.

5.6 Effect of NUMA Awareness

Fig. 20 shows the overall speedup of SpMV with and without applying the NUMA-aware design. All other optimizations are applied in both cases. The cost of copying data in between NUMA nodes are omitted in the results. From the figure we can see that, on Summit, the positive impact of NUMA awareness on the performance is clear. The design without NUMA awareness cannot scale well beyond 3 GPUs. However, on DGX-1 we do not consistently observe any NUMA effect.

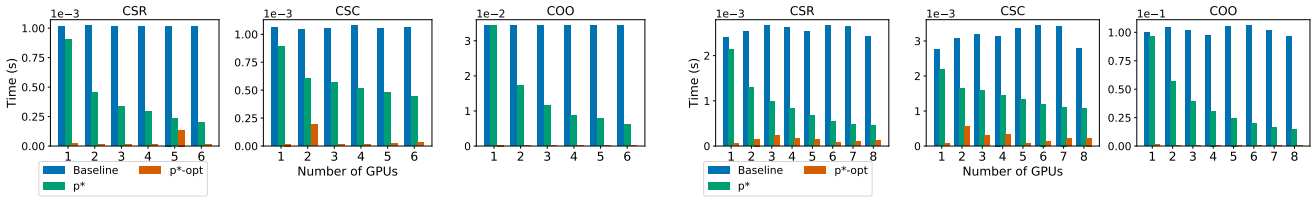
5.7 Overall Performance

We report overall speedup of our SpMV kernel on Summit and DGX-1 machine in **Fig. 21**. From the figure, we observe that the baseline design shows worst performance since it does not perform any optimization to balance the workload. By using our MSREP framework and not applying any optimizations, performance improves but still lacks scalability with larger numbers of GPUs. Finally, applying all optimizations discussed earlier, we achieve near linear speedup with increasing number of GPUs. **Fig. 23-22** show the speedups on different matrices after applying all optimization techniques.

6 DISCUSSION

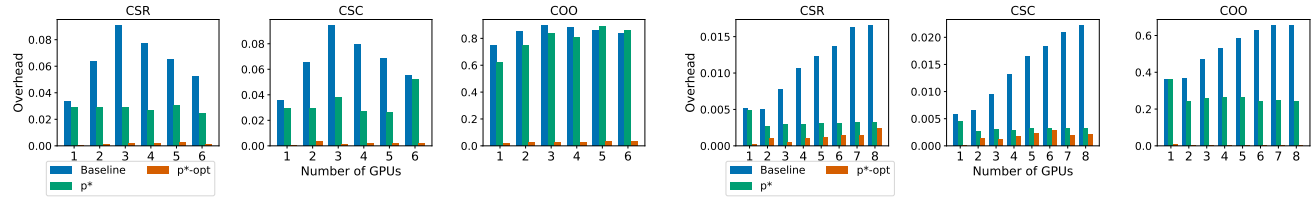
In this section we further discuss our design as well as the potential impact of this work.

Comparing with single GPU works: Our work lays in between single GPU works and distributed GPU works. Single GPU works that focus on sparse data need to consider both the efficiency of



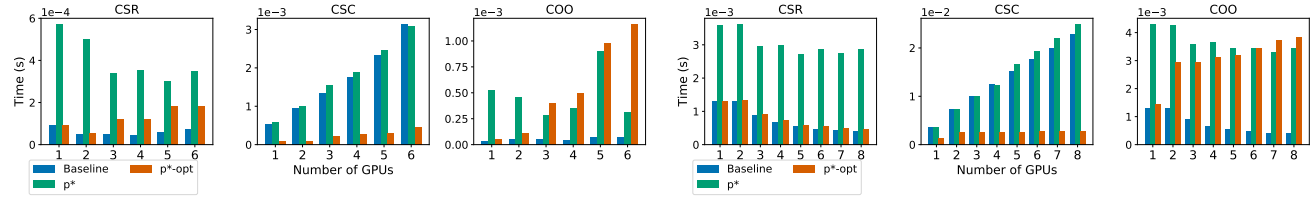
(a) ORNL Summit (b) NVIDIA V100-DGX-1

Figure 16: Time cost of workload partition (input matrix: HV15R)



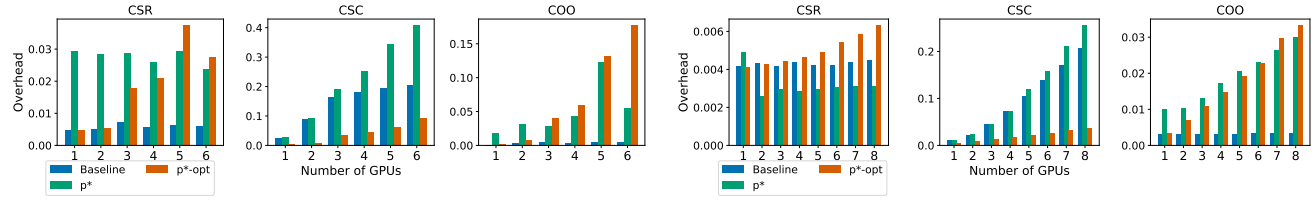
(a) ORNL Summit (b) NVIDIA V100-DGX-1

Figure 17: Overhead workload partition (input matrix: HV15R)



(a) ORNL Summit (b) NVIDIA V100-DGX-1

Figure 18: Time cost of merging partial results (input matrix: HV15R).



(a) ORNL Summit (b) NVIDIA V100-DGX-1

Figure 19: Overhead of merging partial results (input matrix: HV15R).

loading data to registers and fine grain parallelism in the thread level. So, it is common to develop a new storage format to facilitate the performance optimization. Our multi-GPU work aims to leverage existing single GPU works and make them scalable beyond one GPU. Thus, one major focus on this work is the compatibility with existing works and sparse data format is the main dominate factor for compatibility. Instead of proposing new formats, we aim to make existing sparse formats scalable across GPUs. Special attention must be made to data partitioning, result merging. Also, we need to explicitly handle memory copies in between CPUs and GPUs considering their inter-connect topology.

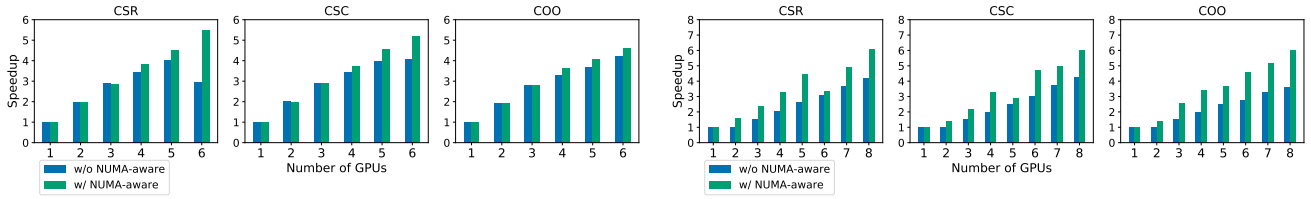
Impact on distributed GPU systems: Distributed GPU systems are similar to dense multi-GPU systems. So our framework can also support distributed GPU systems. However, since inter-connect bandwidth in between GPU computing nodes is usually limited, special care must be taken when choosing the workload type and data format. For example, SpMV with CSR or COO input brings less communication cost so it is more likely to give relative good scalability on distributed GPU systems.

Benefits to applications: Enabling scalable sparse operations on multi-GPU systems can potentially benefit many applications that rely on them such as compressed Convolution Neural Network (CNN) [20], Graph Convolutional Network [15, 19], Graph-based algorithms [10, 38] and many other exascale scientific applications [13].

7 RELATED WORKS

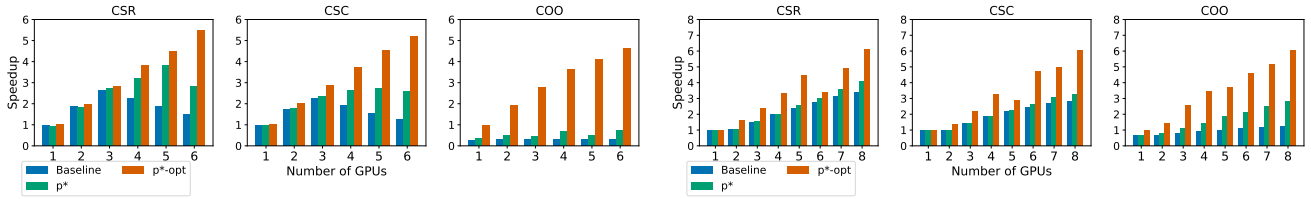
In this section, we summarize existing works regarding multi-GPU SpMV, single-GPU SpMV and graph algorithms that adopts can benefit from our sparse data format for multi-GPU execution.

SpMV on Multi-GPUs: Yang et al. [39] implemented SpMV on a multi-GPU cluster with each CPU node attaching to a single GPU. Each node keeps a local partition of the matrix, while at the end all



(a) ORNL Summit **(b) NVIDIA V100-DGX-1**

Figure 20: Comparing speedup with and without NUMA-aware (input matrix: HV15R)



(a) ORNL Summit **(b) NVIDIA V100-DGX-1**

Figure 21: Comparing total speedup (input matrix: HV15R).

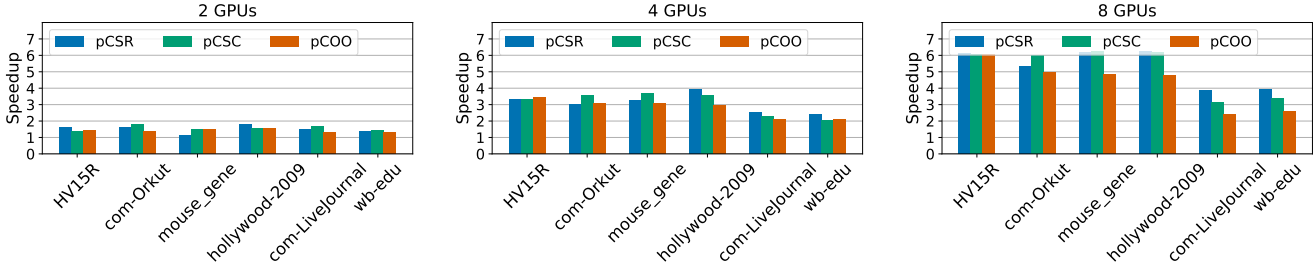


Figure 22: Speedup comparison using different matrices on NVIDIA V100-DGX-1.

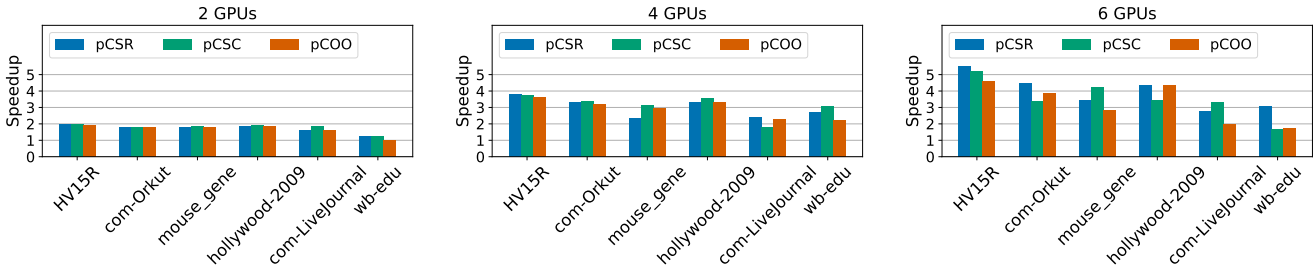


Figure 23: Speedup comparison using different matrices on ORNL Summit.

nodes broadcast their local results to the other nodes. This communication cost of broadcasting is the key factor limiting the scalability. Our work is distinguished from this work in three aspects: 1) Only partial results are merged with lower overhead; 2) Advanced workload distribution strategy is proposed to handle workload imbalance; 3) Rather than assuming a homogeneous node configuration (using MPI), we optimize our design by addressing NUMA effect. We also want to mention that our intra-node scale-up design is independent of their scale-out design, and thus can be integrated to enable the processing of extremely large matrices.

Kreutzer et al. [24] proposed an SpMV design based a new sparse matrix storage format called pJDS. Again, this work also targeted on distributed GPU clusters. Schubert et al. [5] proposed a multi-GPU SpMV design – KSPARSE SpMV, based on the Blocked-Sparse-Row format. Their implementation assigned independent tasks across GPUs, making it difficult to conserve workload balancing. Guo et al. [18] implemented SpMV on multi-GPU systems using CPU multi-threading and concurrent GPU streams, which cannot leverage the high-speed GPU interconnect such as NVLink.

SpMV on Single GPU: Plenty of research has been done to improve SpMV performance on a single GPU. Some of them proposed new data formats for facilitating the processing of SpMV, such as CSR5 [26], BCCOO [37], SELL-C [6], HYB [8], while others applying auto-tuning methodology for improving SpMV performance [12, 16, 22, 28, 33]. Many works also proposed architecture-specialized designs leveraging the unique opportunities from the GPU hardware [7, 8, 17, 21, 27, 32]. Our work can utilize these fast implementation for single GPU and leverage them in MSREP.

Graph Algorithms: Graph algorithms are an important class of applications that can directly benefit from the proposed pCSR and

pCSC data structures for multi-GPU execution. For example, existing multi-GPU graph analytics frameworks such as Gunrock [30] and Groute [9] essentially partition the graph data in CSR format across multiple GPUs. In addition, GraphBLAS [10] specification expresses graph algorithms as sparse matrix and vector operations on an extended semi-ring algebra, which can be accelerated by GPUs [38]. Our sparse matrix representation as well as the SpMV design can embark new optimization opportunities for GraphBLAS.

8 CONCLUSION

Sparse linear algebra kernels play a critical role in numerous applications. As the volume of data growth exponentially, it will soon become impractical for conducting sparse linear algebra operations in a single GPU due to limited memory capacity and computation performance. In this work, we propose a novel sparse matrix representation framework for multi-GPU systems – MSREP, which aims to help scale sparse linear algebra operations on multiple GPUs while leveraging existing works on a single GPU. We use SpMV to showcase the efficiency of our framework. Evaluation results on an NVIDIA V100-DGX-1 system and the Summit supercomputer show that our SpMV design based on MSREP can achieve linear speedup: 5.5× using 6 GPUs on Summit, and 6.2× using 8 GPUs on the NVIDIA V100-DGX-1 system.

ACKNOWLEDGEMENT

This research was supported by the S-BLAS project under PNNL's High Performance Data Analytics (HPDA) program. This research was supported by the U.S. DOE Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

REFERENCES

[1] 2020. CUDA Unified Memory. https://devblogs.nvidia.com/unified-memory-in-cuda-6/. (2020). [Online; accessed 2020].
[2] 2020. NVIDIA DGX Systems. https://www.nvidia.com/en-us/data-center/dgx-systems/. (2020). [Online; accessed 2020].
[3] 2020. NVIDIA NVSHMEM. https://github.com/NVIDIA/df-nvshmem-prototype. (2020). [Online; accessed 2020].
[4] 2020. Summit at Oak Ridge National Laboratory. https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/. (2020). [Online; accessed 2020].
[5] Ahmad Abdelfattah, Hatem Ltaief, and David Keyes. 2015. High performance multi-GPU SpMV for multi-component PDE-based applications. In *European Conference on Parallel Processing*. Springer, 601–612.
[6] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2014. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs. *University of Tennessee, Tech. Rep. ut-eecs-14-727* (2014).
[7] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 781–792.
[8] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.
[9] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.
[10] Aydin Buluc, Timothy Mattson, Scott McMillan, Jose Moreira, and Carl Yang. 2017. The GraphBLAS C API Specification. *GraphBLAS. org, Tech. Rep.* (2017).
[11] Jieyang Chen, Nan Xiong, Xin Liang, Dingwen Tao, Sihuan Li, Kaiming Ouyang, Kai Zhao, Nathan DeBardeleben, Qiang Guan, and Zizhong Chen. 2019. TSM2: optimizing tall-and-skinny matrix-matrix multiplication on GPUs. In *Proceedings of the ACM International Conference on Supercomputing*. 106–116.
[12] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM sigplan notices* 45, 5 (2010), 115–126.
[13] Jeanine Cook, Hal Finkel, Christoph Junghams, Peter McCorquodale, Robert Pavel, and David F Richards. 2017. *Proxy App Prospectus for ECP Application Development Projects*. Technical Report. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).
[14] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article Article 1 (Dec. 2011), 25 pages. DOI: http://dx.doi.org/10.1145/2049662.2049663
[15] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*. 3844–3852.
[16] Athena Elafrou, Georgios Goumas, and Nectarios Koziris. 2019. BASMAT: Bottleneck-Aware Sparse Matrix-Vector Multiplication Auto-Tuning on GPGPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 423–424. DOI: http://dx.doi.org/10.1145/3293883.3301490
[17] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.
[18] Ping Guo and Changjiang Zhang. 2016. Performance Optimization for SpMV on Multi-GPU Systems Using Threads and Multiple Streams. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. IEEE, 67–72.
[19] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.
[20] 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).
[21] 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. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 66–79.
[22] Kaixi Hou, Wu-chun Feng, and Shuai Che. 2017. Auto-tuning strategies for parallelizing sparse matrix-vector (spmv) multiplication on multi-and many-core processors. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 713–722.
[23] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. 2004. Sparsity: Optimization framework for sparse matrix kernels. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 135–158.
[24] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R Bishop. 2012. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, 1696–1702.
[25] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 117–126.
[26] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 339–350.
[27] Duane Merrill and Michael Garland. 2016. Merge-based parallel sparse matrix-vector multiplication. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 678–689.
[28] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. 2010. Automatically tuning sparse matrix-vector multiplication for GPU architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 111–125.
[29] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.
[30] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2017. Multi-GPU graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 479–490.
[31] Gerald Schubert, Holger Fehske, Georg Hager, and Gerhard Wellein. 2011. Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. *Parallel Processing Letters* 21, 03 (2011), 339–358.
[32] 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*. 1–11.
[33] Guangming Tan, Junhong Liu, and Jiajia Li. 2018. Design and implementation of adaptive spmv library for multicore and many-core architecture. *ACM*

Transactions on Mathematical Software (TOMS) 44, 4 (2018), 1–25.

[34] Richard Wilson Vuduc and James W Demmel. 2003. *Automatic performance tuning of sparse matrix kernels*. Vol. 1. University of California, Berkeley Berkeley, CA.

[35] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. 2007. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 1–12.

[36] Biwei Xie, Jianfeng Zhan, Xu Liu, Wanling Gao, Zhen Jia, Xiwen He, and Lixin Zhang. 2018. Cvr: Efficient vectorization of spmv on x86 processors. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 149–162.

[37] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: yet another SpMV framework on GPUs. *Acm Sigplan Notices* 49, 8 (2014), 107–118.

[38] Carl Yang, Aydin Buluc, and John D Owens. 2019. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. *arXiv preprint arXiv:1908.01407* (2019).

[39] Xintian Yang, Srinivasan Parthasarathy, and P. Sadayappan. 2011. Fast Sparse Matrix-Vector Multiplication on GPUs: Implications for Graph Mining. *Proc. VLDB Endow.* 4, 4 (Jan. 2011), 231–242. DOI:http://dx.doi.org/10.14778/1938545.1938548

[40] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 94–108.