Hierarchical Computation and Communication for Distributed Sparse Operations on Supercomputers with Multi-GPU Nodes

Mert Hidayetoğlu

January 2021

1 Introduction

Sparse operations are the main computational workload for various scientific, AI, and graph analytics applications. Most of the time, the computational cost of sparse operations constitutes an overall performance bottleneck. This thesis investigates this bottleneck at two specific points: memory accesses and communications.

The first investigation, on sparse memory accesses, targets accelerating single-device (GPU in this thesis) execution of distributed sparse computations. To that extent, this thesis proposes a novel tiling technique for sparse matrix - dense matrix multiplication (SpMM). The larger shared-memory capacity and new-generation high-bandwidth memory (HBM2) enables these techniques. Although the techniques described herein can be applicable for more general computations, the Tiled SpMM fits well to many physics-based and also data-driven applications that are relevant today. Examples include image reconstruction [1, 2], sparse neural networks [3], graph neural networks [4], and so on.

The second investigation, on sparse communications, targets alleviating the communication bottleneck of large-scale sparse applications, especially when the sparse matrix is very large. To that extent, this thesis a hierarchical communication strategy for SpMM-based distributed applications. The hierarchical communication strategy exploits the hierarchical architecture of the network interconnect in all levels, i.e., socket level, node level, rack level and so on.

For end-to-end acceleration of large-scale applications involving sparse computation and communication, I propose to make two main contributions:

1. A novel sparse matrix tiling for accelerating SpMM on GPUs.
2. Hierarchical communication strategy for the multi-GPU node architecture.

Accordingly, this proposal is structured into two main sections: Section 2 explains the Tiled SpMM in details and Section 3 explains the hierarchical communication strategy with a demonstration. These sections present the work completed to date along with open problems and proposed solutions to be studied in order to complete the thesis.

The performance demonstrations are provided from two applications that these techniques are already successfully applied: MemXCT [2] and SpDNN [3]. The MemXCT application is developed by me and my collaborators based on the methods presented in the thesis at Argonne National Laboratory to tackle extremely-large X-ray image reconstruction at syncrothron light source. The MemXCT application and its evaluation on the Summit supercomputer won the Best Paper Award at SC20[1]. The SpDNN application is developed by me and my collaborators at the IBM-Illinois

[1] Link Here: https://tinyurl.com/372tt35e

Center for Cognitive Computing and Systems Research to tackle performance bottleneck of sparse neural network inference. The SpDNN application was named the champion at 2020 MIT/Amazon/IEEE Graph Challenge[2]. The proposed techniques herein demonstrates unprecedented scale and performance for the MemXCT and SpDNN applications, and this thesis presents experimental characterization of these techniques on various architectures and applications.

2 Accelerating Sparse Computation

This section explains the acceleration of SpMM-based operations on single GPUs through the proposed Tiled SpMM technique.

2.1 Definitions

how affordable? (target sizes?)

An SpMM can be mathematically written as $\boldsymbol{A} \times \boldsymbol{B} = \boldsymbol{C}$, where $\boldsymbol{A}$ is an $M \times N$ sparse matrix, and $\boldsymbol{B}$ and $\boldsymbol{C}$ are dense matrices of size $N \times K$ and $M \times K$, respectively. To save memory, the sparse matrix is often stored and multiplied in a compressed form, where non-zero entries are not stored.

Traditional storage for the sparse matrix $\boldsymbol{A}$ include coordinate list (COO), compressed sparse row (CSR), and compressed sparse column (CSC) formats. Another storage format is ELL, which is originally used by the ELLPACK system for solving ellyptic boundary value problems [5]. ELL format employs zero-padding for an optimized memory access on GPUs with the expense of an additional zero-padding overhead [6]. This thesis considers CSR format as the baseline [7], which has $\mathcal{O}(KM + KN_{\text{nz}})$ computational complexity, where N_{nz} is the number of nonzeroes in $\boldsymbol{A}$. The proposed Tiled SpMM kernel employs a specialized ELL format described in Section 2.2. This thesis assumes dense matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ are stored in the column-major layout since it fits better than the row-major layout for the target applications.

2.2 CSR and ELL Storage Formats

A naive SpMM is based on the CSR storage format involving `displ`, `index`, and `weight` data structures. The first data structure, `displ`, stores the index pointing the first nonzero element of the corresponding row, and has a length of $M + 1$. The latter data structures stores the values and the corresponding column indices of nonzeroes in each row. In the naive CSR-based SpMM, each CUDA thread processes a single row of $\boldsymbol{A}$. Since each row has a different number of nonzeroes, the CSR-based SpMM yields not only thread divergence but also uncoalesced access to the `index` and `weight` data. As a remedy, ELL storage the rows are sliced and the memory layout is transposed and `index` and `weight` structures are zero-padded with corresponding granularity. This is sometimes considered as *sliced* [8] or *blocked* [9] ELL, but this thesis refers it as ELL. In the ELL storage, `displ` is smaller since it only points to the first element of each block, determined by the slice width.

The zero padding increases the memory footprint and consumes extra memory bandwidth during SpMM. To minimize this cost, it is a common practice to sort the rows by their number of nonzeroes. However, this thesis does not apply sorting since it changes the data access pattern, which is crucial for the optimized SpMM. For GPU implementation, it is logical to choose slice size equal to the warp size (32 in this case) to minimize the zero-padding overhead.

Figure 1(a) shows an example of an 13×14 sparse matrix with 57 nonzeroes (68.7% sparsity), and corresponding `displ` and `index` data structures in CSL and ELL storage formats. In this

[2]Link Here: https://tinyurl.com/1xldtlij

expl. relationship to lit

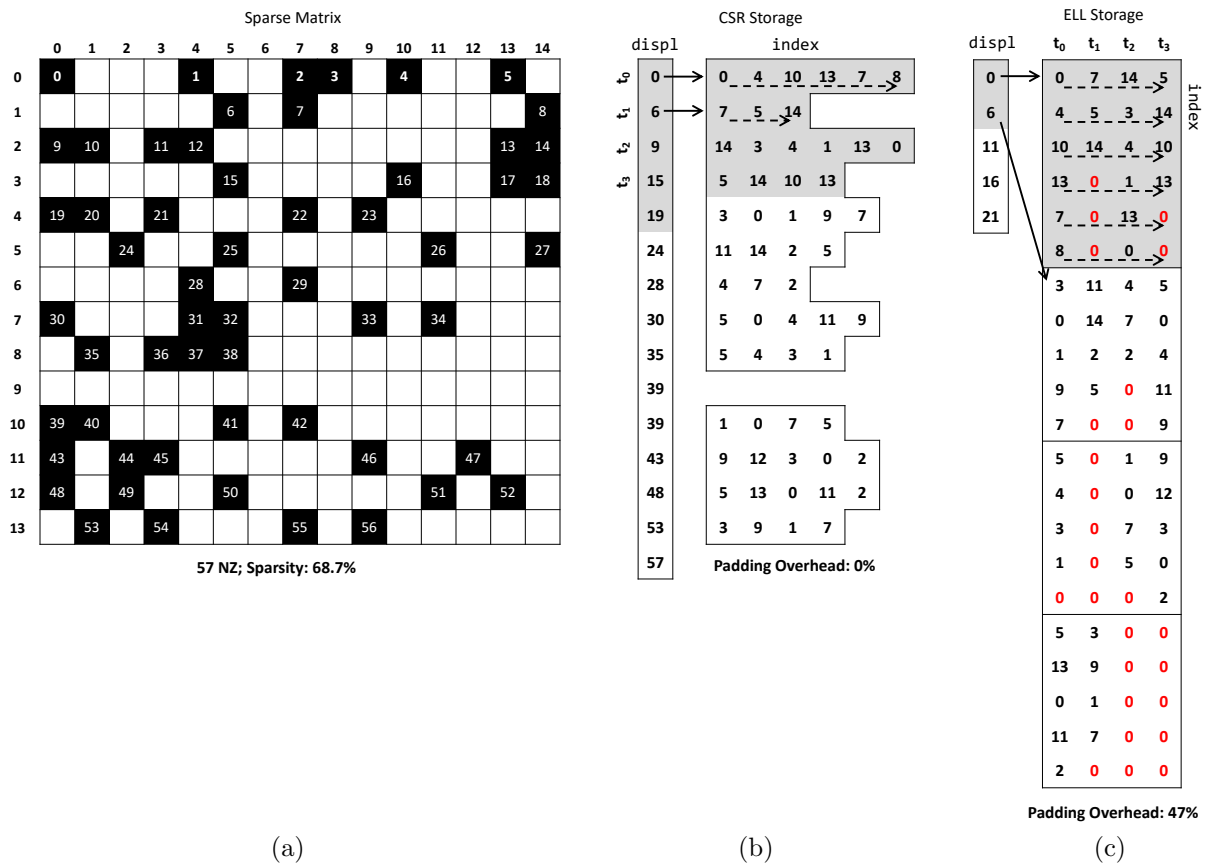


Figure 1: (a) $13 \times 14 \times 4$ sparse matrix and its (b) CSR and (c) ELL storage layouts.

example, each thread warp is assumed to be of size four, where the accessed data by a single thread warp is highlighted. In CSR storage, each thread in the warp accesses a different number of `index` values, resulting in a thread divergence. Moreover, since threads do not access to consecutive elements, cache-line utilization is low due to uncoalesced nature of the access. ELL storage format solves these problems with an expense of 47% zero-padding overhead.

Note that `index` is not sorted within each row in this example because the matrix is generated as such in many applications, e.g., XCT and graph analytics.

2.3 Tiled SpMM

This section explains accelerating single-GPU throughput by Tiled SpMM–one of the two main contributions of this thesis.

2.3.1 Overview

The CSR data structures are stored in GPU memory (HBM2 for Nvidia P100, V100, and A100 GPUs), where `displ` is stored in `long` format to represent large matrices. Larger data structures `index` and `value` are also stored in `int` and in desired precision (`float` in this case), respectively. Similarly, $\boldsymbol{B}$ and $\boldsymbol{C}$ are stored in GPU memory in column-major format with desired precision.

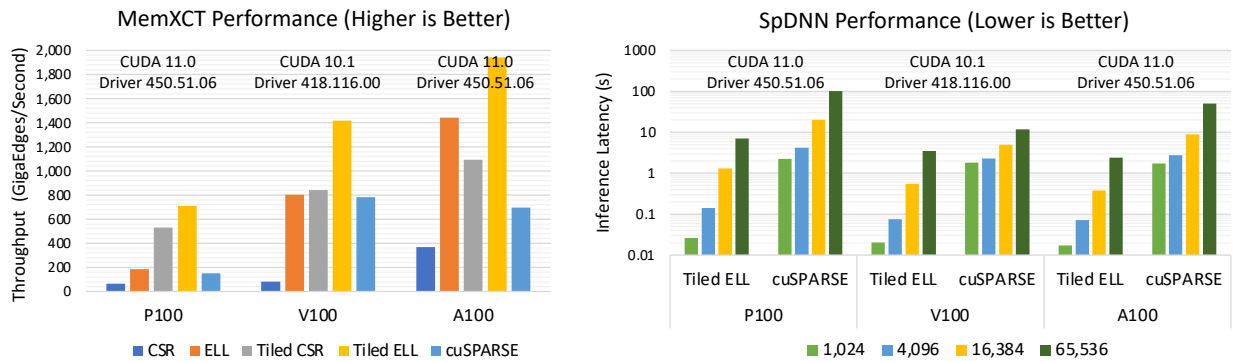
Figure 2 shows the overall performance of the proposed Tiled SpMM for MemXCT and SpDNN applications. Figure 2(a) Shows the GigaEdges/Second performance of the proposed Tiled SpMM

explain the app?

abs. perf numbers?

with CSR and ELL storage formats and Nvidia cuSPARSE library on three generations of P100, V100, and A100 GPUs. This demonstrates that Tiled SpMM with ELL format obtains the highest performance, with 4.69×, 1.81×, and 2.79× speedup over cuSPARSE, respectively[3]. Tiled SpMM's higher performance due to newer generations of GPUs is the increased HBM2 bandwidth and larger shared-memory capacity. Figure 2(b) Shows latency of the single-GPU inference of SpDNN application on logarithmic scale. The network models are taken from the sparse neural networks provided by the Graph Challenge. There are four networks with 120 layers and 1,024, 4,096, 36,384, and 65,536 neurons per layer. In this case, the sparse layer computation is based on SpMM fused with the activation function. This cannot be done in cuSPARSE directly, and therefore the SpMM and activation are performed separately. Results show that the proposed Tiled SpMM obtains 3.39× up to two orders of magnitude faster inference on the given networks[4].

how much due to fusion?



(a) (b)

Figure 2: (a) SpMM throughput in 3D image reconstruction. (b) Inference throughput in spare neural network.

All results in Figure 2 are tuned as explained in Section 2.4. For fair comparison, cuSPARSE is also tuned for the maximum performance. One observation is that cuSPARSE performs better on V100 GPU than on A100 GPU with the noted CUDA and driver versions on the figure. Further investigation will be conducted on the cuSPARSE versions and their performance.

2.3.2 Shared-Memory Tiling

The main bottleneck of naive SpMM is the irregular global memory access, which yields high latency accesses and consumes unnecessary bandwidth due to the underutilization of cache lines. The main idea of the shared-memory tiling is buffering data in shared memory for staging irregular accesses. That is, in the preprocessing step, by inspection of the sparsity pattern of the matrix, input data accesses by each thread block is found. Then a few extra data structures that maps global memory to shared memory are constructed. Using these data structures, for each Tiled SpMM, threads loads data from global memory and stores them in shared memory. After a thread synchronization, threads accesses shared memory irregularly with less latency and with finer granularity. Since the data is accessed from shared memory, the `index` data structure has to be updated with the new shared-memory indices. This brings another opportunity to store `index` as two-byte `short`, which saves 33% storage and global-memory bandwidth. For large problems, however, the

[3]Results can be reproduced using this repository: https://github.com/merthidayetoglu/MemXCT-GPU
[4]Results can be reproduced using this repository: https://github.com/merthidayetoglu/SpDNN_Challenge2020

explain

pseudocode !

why?

memory access footprint of each block is so large that the shared-memory tiling cannot be implemented with a single staging. As a remedy, this thesis develops a multi-staging strategy for the shared-memory tiling.

Figure 3 explains the multi-staging for the proposed Tiled SpMM kernel, whose code is provided in Listing 1. In this example, block size is four and thread size is two, therefore there are four thread blocks and eight warps. The register tiling factor (`REGISTER`) is two, meaning that each thread computes two outputs, and the shared-memory buffer can store 12 elements. Figure demonstrates operations of only the first block's, where those of the first threads are highlighted. The first stage loads global-memory locations {0,1 3,4,5,7} and stores them in shared memory locations {0,1,2,3,4,5}, according with the `map` data structure. Since each thread block may perform different numbers of stagings, `mapdispl` data structure is necessary to point the beginning of each staging. After the synchronization, the first thread (t_0) accesses data with the indices {0,3,5}, with original indices {0,4,7}. Once all threads are done with the fused-multiply-add (FMA) operations, the second staging is performed with additional synchronizations. ?

Unclear

Even though the baseline CSR storage is efficient in terms of memory footprint, the access is not coalesced among threads in each warp. To address this, we store the weight matrix in the ELL storage format with zero-padding in warp granularity. Figure 3(c) shows the `displ` and `index` data structures. The first two columns of windex (in orange) in Figure 3(b) shows the layout of the elements for the first block. The top two sections separated by the dashed line each contains the `index` elements accessed by a warp of the block during the first stage of execution. The bottom two sections are accessed by the same two warps during their second stage of execution. The `displ` elements marks the positions of the dash lines and solid lines for every warp and every thread block. The padded zeros are highlighted with red font color. The dashed lines represents the boundary between warps (each block has two warps in this example) and solid lines represent the boundaries between buffer stages, i.e., all blocks except block 2 involve two stagings and block 3 involves three stagings. In this example, the zero padding overhead is 27.5% for warp granularity, however, it would be 80% and 100% with zero padding in tile and layer granularity, respectively. Warp-level padding introduces a small number of zeros while maintaining coalesced (efficient) memory access.

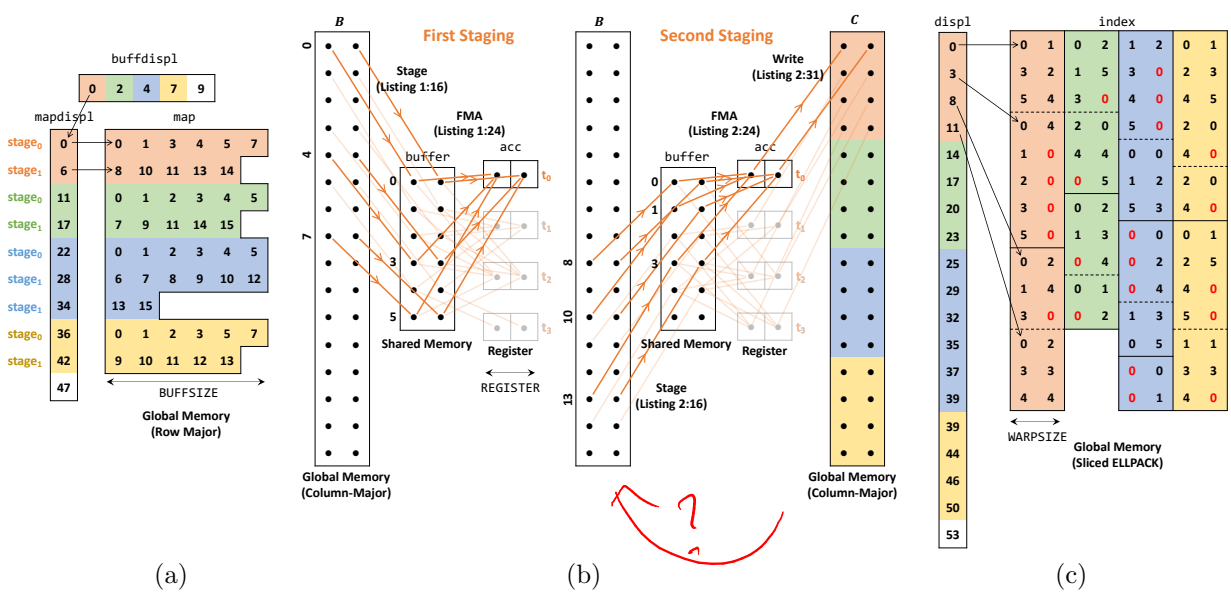


Figure 3: (a) Auxiliary data structures for the shared-memory tiling. (b) Staged shared-memory tiling and corresponding loads and stores. (c) Modified ELL data structure for the Tiled SpMM.

Listing 1: Tiled SpMM Kernel

```
1  __global__ void SpMM_Tiled __launch_bounds__(BLOCKSIZE,NUMBLOCK) (float *C,
2                                                                   float *B,
3                                                                   int *displ,
4                                                                   int *index,
5                                                                   float *value,
6                                                                   int numrow,
7                                                                   int numcol,
8                                                                   int *buffdispl,
9                                                                   int *mapdispl,
10                                                                  int *map,
11                                                                  int buffsize){
12   extern __shared__ float shared[];
13   float acc[REGISTER] = 0;
14   B += blockIdx.y*REGISTER*numcol;
15   C += blockIdx.y*REGISTER*numrow;
16   int row = blockIdx.x*blockDim.x+threadIdx.x;
17   int wind = row%WARPSIZE;
18   for(int buff = buffdispl[blockIdx.x]; buff < buffdispl[blockIdx.x+1]; buff++){
19     int mapnz = mapdispl[buff+1]-mapdispl[buff];
20     for(int n = threadIdx.x; n < mapnz; n += blockDim.x){
21       int ind = map[mapdispl[buff]+n];
22       #pragma unroll
23       for(int k = 0; k < REGISTER; k++)
24         shared[k*buffsize+ind] = B[reg*numcol+ind];
25     }
26     __syncthreads();
27     int warp = (buff*blockDim.x+threadIdx.x)/WARPSIZE;
28     for(int m = displ[warp]; m < displ[warp+1]; m++){
29       float val = value[m*(long)WARPSIZE+wind];
30       int ind = index[m*(long)WARPSIZE+wind];
31       #pragma unroll
32       for(int k = 0; k < REGISTER; k++)
33         acc[k] += shared[k*buffsize+index]*value;
34     }
35     __syncthreads();
36   }
37   if(row < numrow)
38     #pragma unroll
39     for(int k = 0; k < REGISTER; k++)
40       C[k*numrow+row] = acc[k];
41 }
```

2.3.3 Register Tiling

Rgister tiling technique is an output buffering technique, that the `index` and `value` elements are stored in register and reused for computation of many columns. That is each thread is responsible to compute multiple output (column) element, where it accumulates the result in the `acc` with `REGISTER` elements, i.e., the register-tiling factor (Listing 1, Line 13).

Register tiling increases the arithmetic intensity asymptotically-linear with the register-tiling factor, because the matrix elements are reused from registers `REGISTER`. On the other hand, register-tiling increases register pressure. Especially, register tiling unrolls the shared-memory staging, FMA, and global-memory write loops in the lines 23, 32, and 39 in Listing 1. Although it consumes significant register resource, the loop unrolling resolves data dependency between iterations and fills the memory access pipelines more efficiently, providing significant speedup. Nevertheless, the register pressure degrades kernel performance, and therefore the register-tiling factor has to be tuned as discussed in Section 2.4.

2.4 Tuning Parameters

There are three tuning parameters of the proposed Tiled SpMM: 1) Block size, 2) Shared-memory size, and 3) Register tiling factor. An auxiliary compiler optimization parameter is the minimum number of blocks (`NUMBLOCKS` in Listing 1) to be scheduled per SM.

GPU obtains high throughput by hiding memory and computation latency by context switching between warps. Therefore it is desirable for each thread block should use only limited resources so that SM occupancy is 100%. The Tiled SpMM, however, memory latencies are minimized to a great extent that providing larger resources per block is more desirable than maintaining SM occupancy. The explanation is the following: 1) In order to maximize the reuse of $\boldsymbol{X}$ from shared memory, block size needs to be large. 2) To minimize the sychronization in each shared-memory staging, shared-memory per block should be large. 3) For maximizing the reuse of $\boldsymbol{A}$ from register, the tiling factor should be large.

Considering the above, with the GPU generations considered in this thesis, performance is maximized by setting block size to 1024 and setting maximum available shared memory to each block–48 KB for P100, 96 KB for V100, and 163 KB for A100 GPUs. This schedules a single thread block with 32 warps on each SM with 50% occupancy. Knowing that, a single block can employ the whole (64 KB) register resource in SM, and for register opimization the compiler hint `__launch_bounds__` parameters `BLOCKSIZE` and `NUMBLOCKS` should be set to 1024 and 1, respecively.

Register tiling yields register pressure, which is handled by the `nvcc` compiler. With the given hint, the compiler allocates and optimize registers to prevent register leak as much as possible by sacrificing from performance. As a result, register tiling factor should be tuned in order to obtain maximum computational performance.

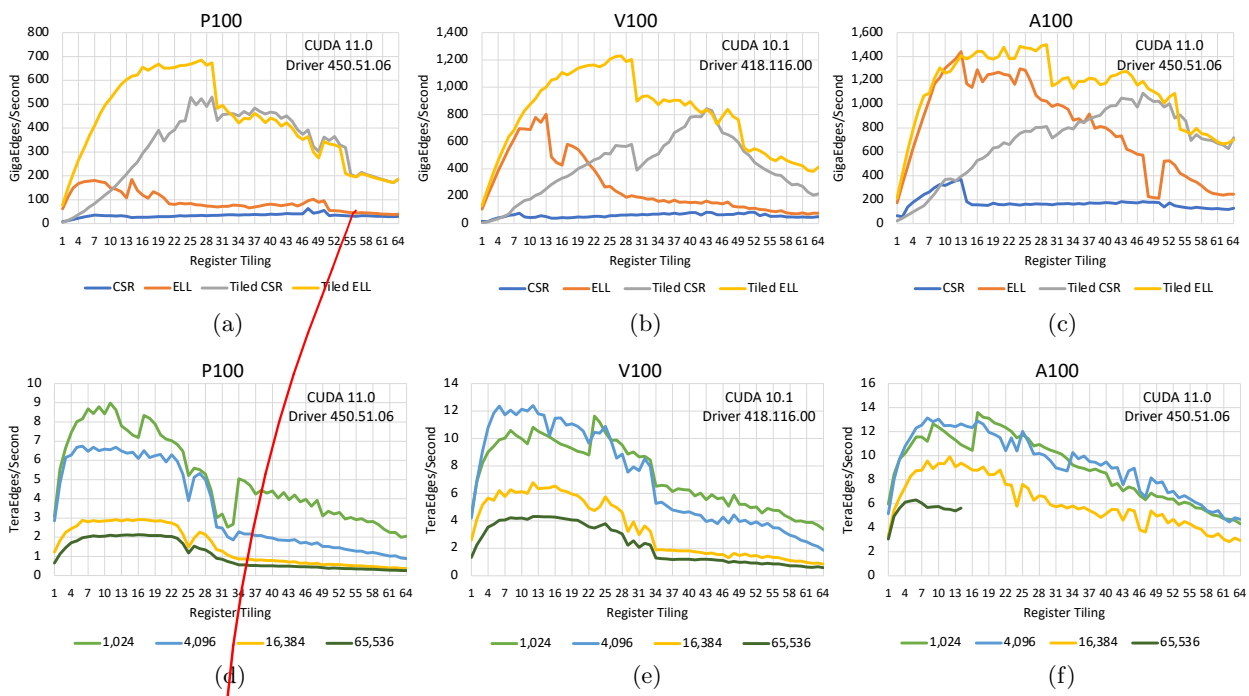


Figure 4: Computational throughput with respect to register tiling factor on P100, V100, and A100 GPUs. Upper row (a)–(c) compares Tiled SpMM performance with naive SpMM for MemXCT application. Lower row (d)–(f) compares inference performance (based on the proposed Tiled SpMM) for the SpDNN application with 120 layers with various number of neurons per layer.

Cut high reg!

why does tiled CSR catch up?

what is tiled CSR

Figure 4(a)–(c) shows sustained computational throughput for the MemXCT application with various register tiling factors. The results are collected by dividing the total SpMM kernel time by the number of nonzeroes processed in projection operation during iterative image reconstruction. The proposed Tiled SpMM with ELL storage format outperforms sub-optimal SpMM versions. Register tiling initially improves the SpMM performance thanks to the data reuse from register. On the other hand, the register pressure on each SM limits the performance. Especially, performance drops sharply when the register tiling factor is larger than 29. It is most probable that `nvcc` compiler employs has a different register allocation strategy by sacrificing from performance to avoid register spill, and it is cumbersome to disseminate the register allocation further because of the closed-source nature of the compiler. possible to look deeper!

Another performance bottleneck arises (especially in A100) is the shared-memory bank conflicts due to irregular memory accesses to the shared memory. That is, the shared-memory latency is not exposed in the P100 and due to its lower HBM2 bandwidth compared to V100 and A100, but it is not the case with the higher HBM2 bandwidth. This thesis proposes resolving the shared-memory bank conflict (Section 2.5.3) in order to improve Tiled SpMM throughput by 15% and 30% for V100 and P100 GPUs, respectively.

Figure 4(d)–(f) shows sustained computational throughput for the SpDNN application [3] with various register tiling factors. The results results are collected by dividing total fused SpMM+ReLU kernel time for sparse layers during inference. In this case, neurons per layer, i.e., matrix size, effects the performance, and an optimal register tiling factor is different for different matrix size. Since register tiling factor is a compile-time parameter, it is up to the user to find an optimal value by inspection.

2.5 Open Problems and Proposed Solutions

This section lists a few problems with the proposed Tiled SpMM algorithm. Upon acceptance of this proposal, additional thesis research will be conducted and results will be presented in the dissertation.

2.5.1 Enhancing Tiled SpMM by Matrix Reordering

Performance of the proposed Tiled SpMM depends on the sparsity pattern of the matrix. That is, the preprocessing step inspects the sparsity pattern to find any opportunity to provide data reuse from shared memory. To improve the efficiency of shared-memory tiling, this thesis investigates matrix reordering algorithms.

Figure 5(a) shows the sparsity pattern of a matrix from the MemXCT application. In this toy example, the matrix is 256×256 of size and 93.59% sparsity. Tiled SpMM assigns each thread block to process 32 rows, where each thread processes a single row. The objective in row reordering is to reassign rows to threads such that rows with similar sparsity patterns are assigned to the same block. In the MemXCT case, data is reordered using Hilbert space-filling curve [1, 2]. This reassignment increases data reuse from shared memory from 2.52 to 4.44, i.e., once stored in shared memory, each data is reused 4.44 times on average. In large-scale deployment of the MemXCT application, row reordering with Hilbert curve improves data reuse up to 30 on KNL and GPU architectures, and provides approximately 5× speedup.

Figure 5(c) and (d) shows column reordering and full (both row & column) reordering of the original matrix shown in (a), respectively. Column reordering does not have any effect on the tiling mechanism, however, it localizes the communications when the matrix is partitioned among many GPUs. The localization of the communications is essential for the hierarchical communication

(roofline) model?

strategy, explained in Section 3.

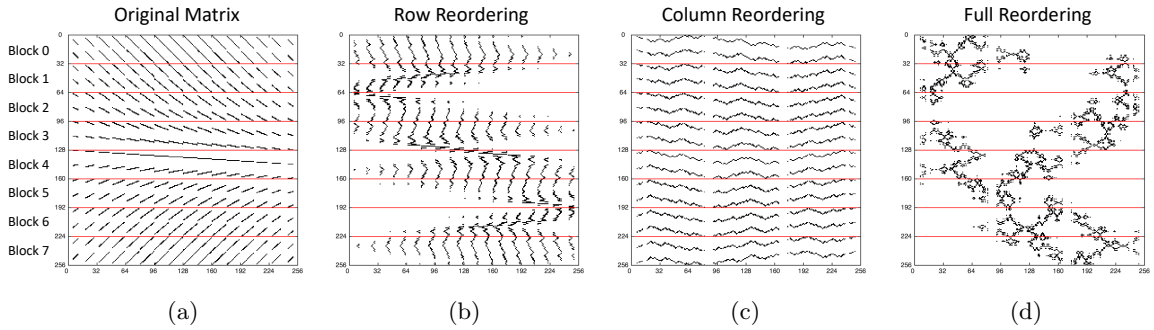


Figure 5: Sparsity patterns of (a) original matrix from MemXCT application, (b) after row reordering, column reordering, and (c) full reordering.

objectives? non-empirical findings?

Locality-preserving space filling curves like Hilbert ordering efficiently maps higher-dimensional regular data to the single dimensional memory address space. Therefore they are useful to store 2D or 3D images or volumes in scientific workloads, such as in the MemXCT application. However, there may not be any solid reordering rule for matrices involved in AI or graph analytics. As a remedy, this thesis will investigate algorithms for row and column reordering of large-scale matrices. This problem can be formulated as a partitioning problem, where similar rows and columns should be partitioned into the same thread block. We can consider sparsity pattern of each row (or column) as a point in the M-dimensional (or N-dimensional) space. These points can be clustered with a variation of k-means clustering. Because k-means clustering does not provide equal-size clusters, but it depends on the position of the initial centroids and distribution of data points.

The k-means clustering is a good candidate for reordering large-scale matrices because of its inherently-parallel nature and low computational complexity $\mathcal{O}(kNM)$ for each iteration, where k is the number of clusters (thread blocks in this case). This can be reduced further by making use of sparsity of the matrix. Alternatively, the problem can be formulated as a combinatorial optimization problem, however, its NP-hard nature is not viable for large-scale matrices. As a result, a heuristic approach may be employed to come up with a sub-optimal solution. One other approach is to employ a greedy algorithm.

2.5.2 On-the-fly CSR-to-ELL Conversion

Results on MemXCT and SpDNN applications show that ELL storage format works well, i.e., better than CSR, for the proposed Tiled SpMM technique. Even though the ELL zero-padding is not a significant overhead for traditional SpMM algorithms, it can be problematic with Tiled SpMM, especially with a large register tiling factor. That is, with register tiling, the shared-memory staging for each column becomes smaller. This increases the zero-padding overhead significantly, but still performs better than Tiled SpMM with CSR storage with no zero padding. The zero padding for ELL has two drawbacks: 1) increases the memory footprint of the application and 2) consumes unnecessary memory bandwidth. To overcome these problems, an on-the-fly conversion will be investigated via warp-level instructions. what if not?

2.5.3 Resolving Shared-Memory Bank Conflicts

Irregular access to the shared memory yields shared-memory bank conflicts. More than often, the number of two-way bank conflicts are the highest, there is less number of three-way conflicts

and even less number of four-way conflicts, and so on. Bank conflicts yield latency and consumes extra shared-memory bandwidth. Practically, this is not a big issue when there is a more serious bottleneck such as global-memory bandwidth. For example, initial experiments on P100, V100, and A100 GPUs show that overall perfomance increases 3.86%, 15.3%, and 29.6%, respectively, when bank conflicts are resolved manually.

This thesis proposes investigating pre-processing mechanisms for solving shared-memory bank conflicts. As in the matrix reordering problem, resolving bank conflicts can be formulated as a combinatorial optimization problem, which is NP-hard to find the optimal solution. Further investigation for faster, sub-optimal solution will be made upon acceptance of the proposal.

3 Accelerating Sparse Communication

This section explains matrix partitioning for SpMM-based operations distributed on many GPUs.

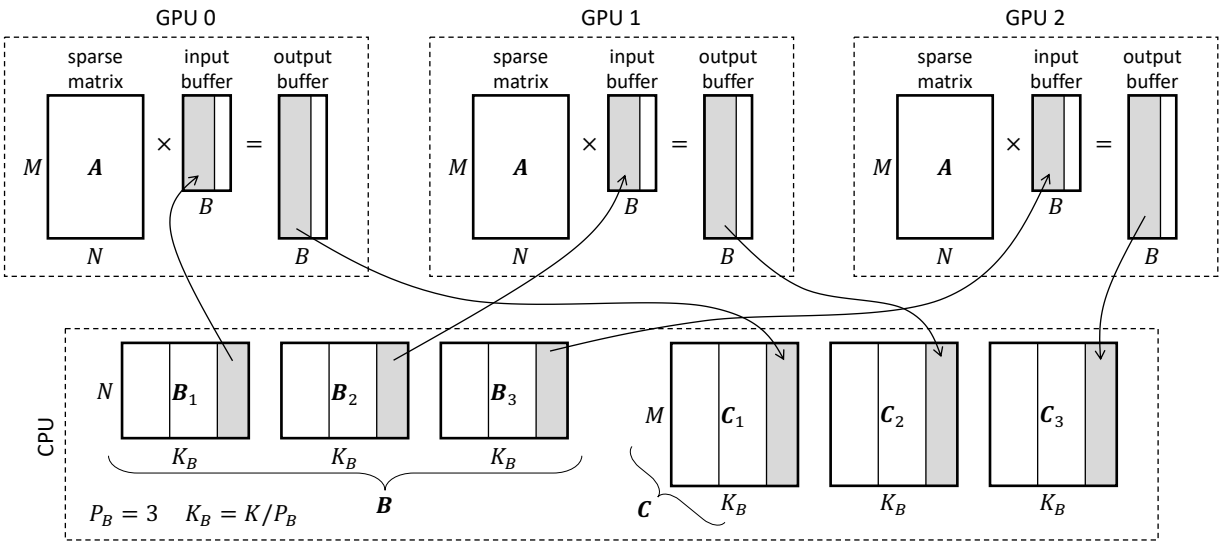


Figure 6: Batch parallelization of SpMM and partitioning of $\boldsymbol{B}$ and $\boldsymbol{C}$ among three GPUs. Streaming of batches with the size of B between CPU and GPU memories.

3.1 Batch Parallelization & Streaming

Assuming $\boldsymbol{A}$ fits into the GPU memory and K–the number of right hand sides–is large, SpMM can be parallelized embarrassingly by partitioning columns of $\boldsymbol{B}$ and $\boldsymbol{C}$ equally among P_B processes, where P_B is the number of *batch processes*. In this case $\boldsymbol{A}$ is duplicated among the batch processes and each process are assigned with approximately $K_B = K/P_B$ columns of $\boldsymbol{B}$ and $\boldsymbol{C}$. This thesis assumes each process is assigned to a single processor–GPU.

Another scenario is that K is so large that $\boldsymbol{B}$ and $\boldsymbol{C}$ does not fit into the single GPU memory, even though they are partitioned. In that case, each batch processor performs multiplications with batches of B columns through input and output buffers of size $N \times B$ and $M \times B$, respectively.

As an example, Figure 6 shows parallelization of $\boldsymbol{A} \times [\boldsymbol{B}_1|\boldsymbol{B}_2|\boldsymbol{B}_3] = [\boldsymbol{C}_1|\boldsymbol{C}_2|\boldsymbol{C}_3]$ among three batch processors, i.e., $P_B = 3$. Assuming $K = 1024$, GPU 0, GPU 1, and GPU 2 are responsible to multiply $K_B = \{342, 341, 341\}$ columns of $\boldsymbol{B}$ and computes the corresponding columns of $\boldsymbol{C}$. As a result, $\boldsymbol{A} \times \boldsymbol{B}_1 = \boldsymbol{C}_1$, $\boldsymbol{A} \times \boldsymbol{B}_2 = \boldsymbol{C}_2$, and $\boldsymbol{A} \times \boldsymbol{B}_3 = \boldsymbol{C}_3$ are performed embarrassingly parallel.

modeling?

proof?, / cite?

In this example, each GPU performs corresponding multiplication through batches of $B = 128$ columns, streaming through $N \times B$ input and $M \times B$ output buffers. In this case, GPU 0 reads and multiplies 128, 128, and 86 columns of $\boldsymbol{B}_1$, and computes the corresponding columns of $\boldsymbol{C}_1$ in each step.

Batch streaming has a minimal overhead as long as SpMM kernel and data transfers are sufficiently overlapped through double-buffering. As a result, the GPU memory footprint of dense matrices are controllable, and the significant memory bottleneck becomes storing the sparse matrix $\boldsymbol{A}$.

?

This strategy is well-applied to the iterative matrix inversion problem, where GPUs store a copy of sparse matrix $\boldsymbol{A}$ and its transpose $\boldsymbol{A}^T$, and solves the unknown matrix $\boldsymbol{B}$ by batch parallelization and streaming through input and output buffers [2]. In the matrix inversion case, $\boldsymbol{C}$ is the input and $\boldsymbol{B}$ is the output. To solve large problems, an additional layer of parallel disk I/O is added for streaming from filesytem.

? a whole different thing?

3.2 Data Parallelism: Partitioning Sparse Matrix

Large-scale applications yield so large sparse matrices (even with very high sparsity) that does not fit into a single GPU memory. In that case, data parallelism has to be employed where the sparse matrix is partitioned among many GPUs until each partition fits into a single GPU. However, data parallelisation comes with a communication overhead, which constitutes bottleneck for many applications.

This thesis proposes employing data and batch parallelism simultaneously to optimize communication. That is, data parallelism is employed only until each partition fits into the GPU memory. Rest of the GPUs are employed by batch parallelism in an embarrassingly parallel fashion, as explained in Section 3.1. Figure 7(a) depicts partitioning of sparse and dense matrices among 4×3 processor grid, where the sparse matrix is partitioned among every four processors, and each partition is duplicated among every three processors. This thesis refers these processes as *data* and *batch* processes, where they partition rows and columns of dense matrices, respectively. For topology-aware rank assignment, these partitions are placed such that data processes are located in the same node, as seen in Figure 7(b). This minimizes communication overhead of data parallelism by utilizing high-bandwidth interconnect, e.g., Nvlink, between GPUs in the same node.

eval of this strategy

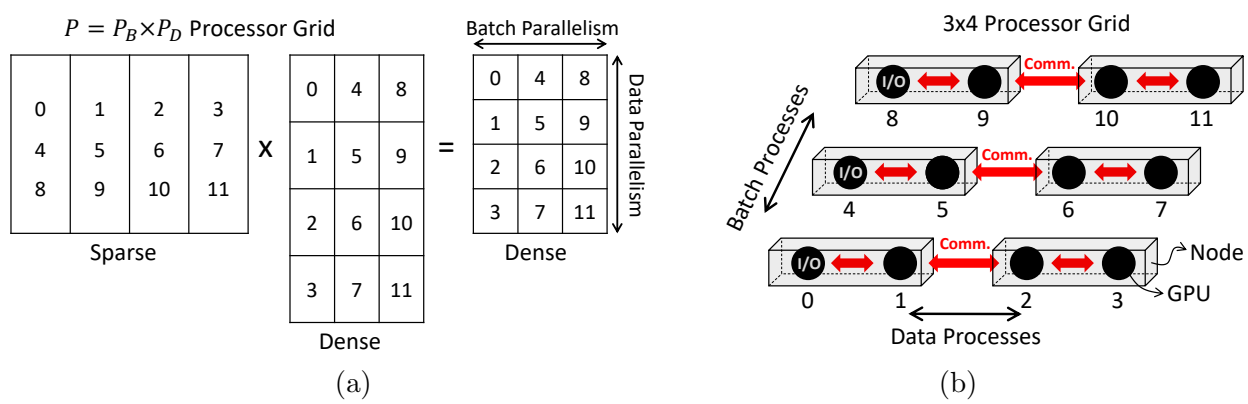


Figure 7: (a) Partitioning of sparse and dense matrices among 4×3 processor grid. (b) Topology-aware rank assignment among multi-GPU nodes.

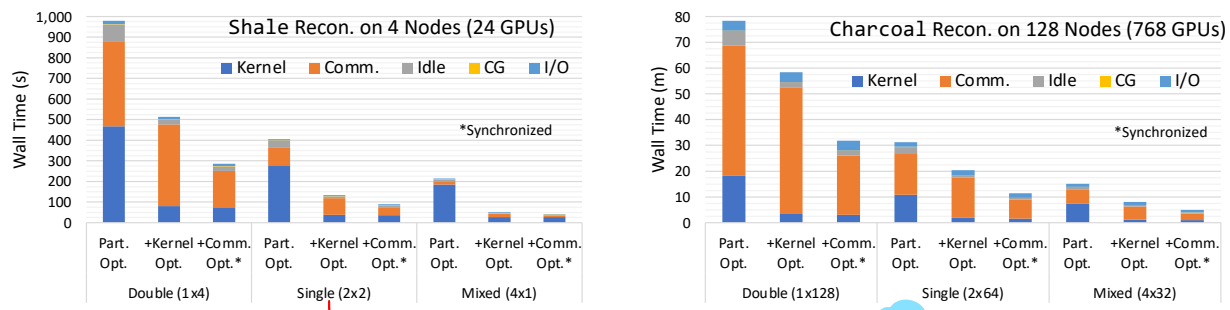
3.3 Hierarchical Communication

This section explains hierarchical communications–one of the two maincontributions of this thesis, originally developed for the MemXCT application [2].

3.3.1 Overview

Data must be partitioned when solving large problems in order to fit per-process memory footprint within available GPU memory. However, data partitioning requires communication and reduction of partial data. Consequently, the overall time is dominated by communication overhead for large problems. As a solution, this thesis propose a novel hierarchical communication involving partial data reduction in a local level. This strategy is designed for multi-GPU node architectures where high-bandwidth connections between peer GPUs are exploited.

Figure 8 shows breakdown of end-to-end image reconstruction with the MemXCT application with two datasets. The first dataset is small so that it fits into four nodes (with 24 V100 GPUs). The second dataset is large so it fits into 128 nodes (768 V100 GPUs). The same reconstruction is performed with double-, single-, and mixed-precision implementations, and therefore yields different memory and communication footprints. At this point, the partitioning strategy of the SpMM is altered according to the batch and data parallelism. That is, execution with double precision fits into four nodes and single precision fits into two nodes. Then data parallelism is reduced and batch parallelism is employed for the single-precision execution.



(a) Why mixed < single? (b)

Figure 8: Breakdown of end-to-end image reconstruction time in the MemXCT application with (a) small shale dataset on four nodes and (b) large charcoal dataset on 128 nodes. Each node has six V100 GPUs. Hierarchical communications reduces the communication time by aprroximately 60%.

The figures shows the ratio of communication and computation (kernel) times. By optimization of the GPU performance with the proposed Tiled SpMM, the communication bottleneck is exposed in both datasets. Hierarchical communications alleviates the communication bottleneck by approximately 60%, providing approximately 50% end-to-end speedup.

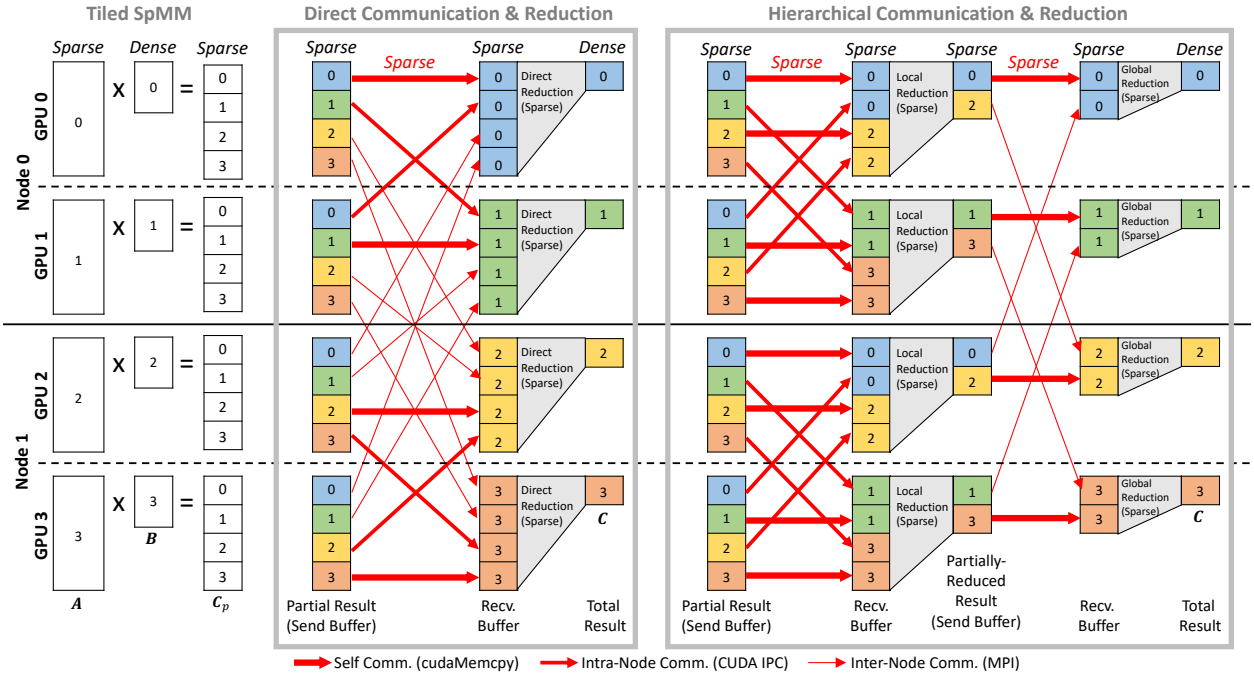
3.3.2 Direct Communications

The partitioning of the sparse matrix $\boldsymbol{A}$ follows a sender-multiplies approach. That is, sender GPUs multiplies its own portion, resulting a partial result in each GPU, i.e., $\boldsymbol{A}_p$. Then the partial results are required to be added together (reduced) for the total result at the receiver GPUs. In distributed SpMM, both partial results and communications are sparse, governed by the sparsity pattern of the sparse matrix $\boldsymbol{A}$, whereas the end result $\boldsymbol{C}$ is dense. Figure 9 (a) shows the partial

SpMM, where each GPU (GPUs 0–3) multiplies their own portion, and stores the partial results in a local buffer. This buffer stores the partial data to be sent to the receiver GPUs 0–3. Note that, each GPU is both sender and receiver.

Figure 9 (b) depicts the all-to-all communication between sender and receiver GPUs. Since this communication is sparse, it can be implemented via `MPI_alltoallv`, either via CUDA-Aware MPI, CPU-staged MPI, or by other means. For most efficient implementation, this thesis performs communications with CUDA inter-process communications (IPC) whenever two GPUs are located in the same node. When they are located in different nodes, CPU-staged MPI is employed to allow the pipelining strategy explained in Section 3.4. When all partial data are communicated, the receiver GPUs reduce the partial data into the total result. The reduction is sparse and its data structures are constructed in the preprocessing stage. As an example, Figure 10 (a) shows the direct communication matrix between 24 GPUs.

For large SpMMs, the direct communication and reduction of partial data yields a significant bottleneck because most of the data travels through slow infiniband interconnect between nodes. To overcome this problem, the proposed hierarchical communication reduces the partial data locally as much as possible, and communicates the reduced partial data between nodes.



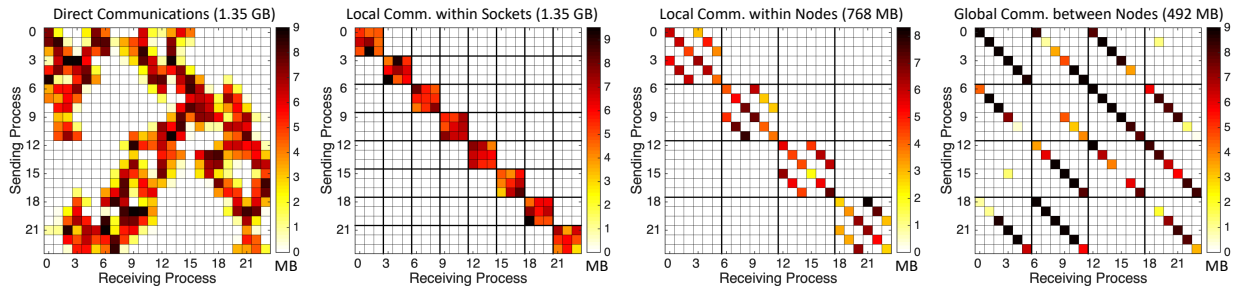
(a) (b) (c)

Figure 9: (a) Each partition of the SpMM results in a partial results. (b) Direct communication and reduction of partial data. (c) Hierarchical communication and reduction of partial data.

3.3.3 Local Reduction of Partial Data

The idea of hierarchical communications is to reduce the partial data locally within nodes, by an extra set of communications, and communicate the reduced data between nodes. Even though there is an extra communication step, this overhead pays well because local communications through Nvlink are faster than the limiting global communications through infiniband.

Figure 9 (c) depicts this idea. First, GPUs in the same node communicates data through



(a) (b) (c) (d)

Figure 10: (a) Direct communication matrix. Hierarchical communication matrices for (b) Socket-level, (c) node-level, and (d) global data reduction.

intermediate buffers, and then, partial data is reduced locally. In this example there are two GPUs in each node. Then, a second set of *global* communications are performed between nodes. Note that each GPU is responsible for reducing partial data going to the corresponding GPU at the receiving node, e.g., GPU 0 sends GPU 0 (self) and GPU 2 (first GPU in the receiving node. Finally, the receiving GPUs reduce the incoming data for the second time to compute the total result.

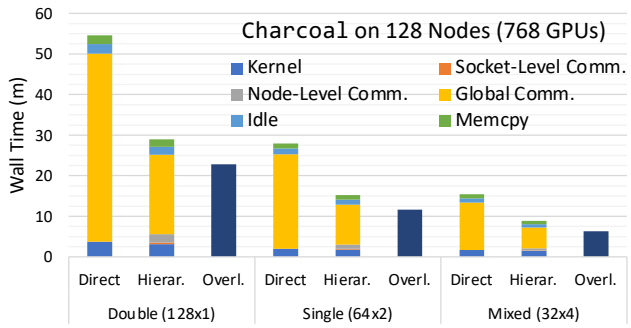
3.3.4 The Three-Level Hierarchy

This sections considers multi-GPU node architecture of Summit supercomputer, where each node has two CPU sockets. Each socket is connected to three GPUs that are densely-connected with high-bandwidth Nvlink interconnect. The CPU sockets within a node are connected with a slower data bus. Each node is connected to an even slower infiniband network. Although, we explain our hierarchical communication and reduction in the context of the Summit architecture, the method is general and applicable to other node architectures with different number of sockets and GPUs. Since the data bus between sockets is slow, we do not perform local reduction directly among six GPUs within a node. Instead, we first perform a socket-level communication and reduction among three GPUs within a socket. Then, an additional node-level communication and reduction among six GPUs within a node. Finally, a global communication and reduction among all GPUs between nodes is performed. Figure 10(b) presents the socket-level communication matrix as a block-diagonal structure because each GPU talks only to three GPUs (including itself) in the same socket. The socketlevel reduction reduces the original 1.35 GB partial data down to 768 MB (43% reduction). Then, the reduced partial data is further reduced within nodes among six GPUs. Figure 10(c) shows the intra-node local communication matrix. After the nodelevel communication, the partial data is further reduced to 492 MB (36% additional reduction). Finally, GPUs send reduced partial data only among nodes as shown in Figure 10(d). As a result, the total data communicated among slowly-connected nodes is reduced by 64% compared to direct communication.

In order to demonstrate the advantage of hierarchical communication strategy on Summit supercomputer, Figure 11 (a) shows a further breakdown of communication time for the MemXCT application. Socket-level, node-level, and global communication times as well as Memcpy time for the CPU staging are noted. Double, single, and mixed precision implementations are also compared, where lower-precision yields less communication volume. Figure 8 (b) shows the corresponding communication volume and bandwidth for a single SpMM (There are 61 SpMM performed in the iterative image reconstruction). In the figure, local and global reduction times are included in the Kernel time since they are performed on GPUs. The hierarchical communications reduces the

model predictions?

inter-node communication volume and the corresponding time by approximately 60%. However, it adds a local communication overhead approximately 4.1% and 0.95% for socket-level and node-level portions, respectively. As a result, end-to-end reconstruction time is reduced approximately by 45%.



(a)

Communication Volume and Effective Bandwidth

		Socket-Level Comm.		Node-Level Comm.		Global Comm.		Memcpy
	Prec.	Data	B/W	Data	B/W	Data	B/W	B/W
Direct	Double	N/A		N/A		36.6 TB	1.61 TB/s	35.2 TB/s
	Single					18.3 TB	1.61 TB/s	34.9 TB/s
	Mixed					9.16 TB	1.59 TB/s	34.6 TB/s
Hierar.	Double	36.6 TB	174 TB/s	21.4 TB	21.3 TB/s	15.2 TB	1.58 TB/s	34.9 TB/s
	Single	18.3 TB	170 TB/s	10.7 TB	22.8 TB/s	7.58 TB	1.55 TB/s	34.5 TB/s
	Mixed	9.16 TB	164 TB/s	5.35 TB	23.5 TB/s	3.79 TB	1.49 TB/s	33.6 TB/s

(b)

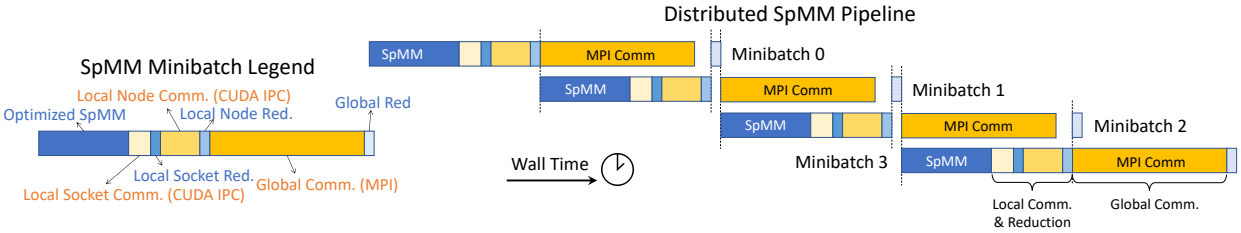
Figure 11: (a) End-to-end reconstruction time with the MemXCT application on 128 nodes of Summit (768 V100 GPUs). (b) Communication volume and effective bandwidth measured during the reconstruction.

Figure 8 (a) also shows the overlapped reconstruction time with the pipelining strategy explained in the next section.

3.4 Pipelining Distributed SpMM

The hierarchical communications for each column can be performed independently. However, communicating small amount of data poorly utilize interconnect bandwidth. As a result, the overall communications becomes latency-dominated. As a remedy, this thesis aggregates communication of many columns, which is called minibatching. Moreover, since the global communications through CPU-staged MPI can be overlapped with the local computations and communications within nodes, they can be overlapped by the proposed pipelinging strategy. One limitation is that the minibatch size should be consistent with the register tiling factor explained in Section 2.3.3.

Figure 12(a) shows the breakdown of end-to-end time of a single minibatch processing: it consists SpMM kernel time as-well-as local communication times in a realistic scale. The pipelining strategy overlaps local computations and communications with global communications and computations. Results for the MemXCT application shows that pipelining the minibatch processing saves approximately 30% time for the end-to-end image reconstrucion.



(a) (b)

Figure 12: (a) Legend for the breakdown of processing of a single minibatch. (b) Minibatch processing pipeling for overlapping local computations and communications.

3.5 Open Problems and Proposed Solutions

This section lists a few problems with the proposed hierarchical communications strategy. Upon acceptance of this proposal, additional thesis research will be conducted and results will be presented in the dissertation. As observed in Figure 5 (d), this can be achieved by matrix reordering.

3.5.1 Enhancing Hierarchical Communication by Matrix Reordering

In distributed SpMM yield sparse communications and its pattern depends on the sparsity pattern of matrix. Therefore, a matrix reordering algorithm can be applied in order to localize communications among GPUs. This can be seen by inspection in Figure 5. Detailed discussion on this problem given in Section 2.5.1.

3.5.2 Extending Communication Hierarchy for Inter-Node Interconnect

The hierarchical communications are originally proposed for the MemXCT application considering the Summit architecture with 6 GPUs per node (IBM AC922). A preliminary work has been done on ThetaGPU located at Argonne National Laboratoy, where there are 8 GPUs per node (Nvidia DGX-A100). This shows, to a great extent, this algorithm is highly portable on supercomputers multi-GPU node architecture, thanks to the high-bandwidth interconnect among GPUs in the node.

This section proposes extending the hierarchical communication idea to the inter-node interconnect. Summit supercomputer has a fat-tree network topology [10], and therefore there is no variation in bisection bandwidth between any two nodes. However, as some pre-exascale computers employs, several exascale computers are highly likely to have more hierarchical communication topologies, such as dragonfly [11]. The hierarchical communication architecture, where nodes inside the same rack will communicate with a higher bandwidth, brings us an opportunity to extend the levels of the hierarchical communication beyond node level. For example, the additional local reduction levels might be added, e.g., among rack and groups or racks.

4 Conclusion

Large-scale sparse applications suffer significantly from inefficient memory accesses and communication bottleneck. This performance degradation yields underutilization of the next-wave of exascale supercomputers. In the meantime, recently developed multi-GPU nodes with high connectivity and bandwidth between the peer GPUs brings optimization opportunities. To take these opportunities, this paper proposes two novel approaches to exploit recent multi-GPU node architecture. The first main contribution is the Tiled SpMM technique to accelerate sparse computations on a single GPU. Tiled SpMM employs novel shared-memory and register tiling techniques for sparse matrices and provides up to 4.7× speedup over Nvidia's cuSPARSE library. The second main contribution is the hierarchical communication strategy to alleviate the communication bottleneck between GPUs in different nodes. The three-level hierarchy reduces the inter-node communication volume by approximately 60% with a few extra communications in socket and node levels. This paper explains these techniques and lists a few open problems along with proposed solutions, further investigation will be conducted upon acceptance. Moreover, this thesis will involve a detailed performance characterization of the proposed techniques on various applications and computer architectures.

References

[1] M. Hidayetoglu, T. Bicer, S. G. D. Gonzalo, B. Ren, D. Gursoy, R. Kettimuthu, I. T. Foster, and W.-M. W. Hwu, "Memxct," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[2] M. Hidayetoglu, T. Bicer, S. G. de Gonzalo, B. Ren, V. D. Andrade, D. Gursoy, R. Kettimuthu, I. T. Foster, and W. mei W. Hwu, "Petascale XCT: 3D image reconstruction with hierarchical communications on multi-GPU nodes," 2020.

[3] M. Hidayetoglu, C. Pearson, V. S. Mailthody, E. Ebrahimi, J. Xiong, R. Nagi, and W.-M. Hwu, "At-scale sparse deep neural network inference with efficient gpu implementation," *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020.

[4] G. Huang, G. Dai, Y. Wang, and H. Yang, "Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20. IEEE Press, 2020.

[5] [Online]. Available: https://www.cs.purdue.edu/ellpack/

[6] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Dec 2008. [Online]. Available: https://research.nvidia.com/publication/efficient-sparse-matrix-vector-multiplication-cuda

[7] M. Steinberger, A. Derlert, R. Zayer, and H.-P. Seidel, "How naive is naive SpMV on the GPU?" *IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2016.

[8] J. Zhang and L. Zhang, "Efficient cuda polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems," *Mathematical Problems in Engineering*, vol. 2013, p. 1–12, 2013.

[9] "cusparse." [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html

[10] C. E. Leiserson, "Fat-trees: Universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. C-34, no. 10, p. 892–901, 1985.

[11] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, no. 3, p. 77–88, Jun. 2008. [Online]. Available: https://doi.org/10.1145/1394608.1382129