RDMA-Based Algorithms for Sparse Matrix Multiplication on GPUs

Benjamin Brock*
Intel Corporation
Santa Clara, CA, USA
benjamin.brock@intel.com

Aydın Buluç
University of California, Berkeley
Berkeley, CA, USA
abuluc@lbl.gov

Katherine Yelick
University of California, Berkeley
Berkeley, CA, USA
yelick@cs.berkeley.edu

Abstract

Sparse matrix multiplication is an important kernel for large-scale graph processing and other data-intensive applications. In this paper, we implement various asynchronous, RDMA-based sparse times dense (SpMM) and sparse times sparse (SpGEMM) algorithms, evaluating their performance running in a distributed memory setting on GPUs. Our RDMA-based implementations use the NVSHMEM communication library for direct, asynchronous one-sided communication between GPUs. We compare our asynchronous implementations to state-of-the-art bulk synchronous GPU libraries as well as a CUDA-Aware MPI implementation of the SUMMA algorithm. We find that asynchronous RDMA-based implementations are able to offer favorable performance compared to bulk synchronous implementations, while also allowing for the straightforward implementation of novel work stealing algorithms.

1 Introduction

Sparse matrix multiplication is an important computational primitive that arises in simulation, data analysis, and machine learning applications. Typically limited by memory and network performance, these computations are especially challenging for unstructured matrices, such as those occurring in graph neural networks, genomics, graph analytics, and other data intensive problems. Sparse matrix primitives provide a convenient and important set of operations for performance tuning that will impact a wide array of applications, and there is a large body of prior work optimizing sparse matrix kernels for multicore [22], manycore [23], GPU [31], and distributed memory environments [1, 16, 18, 24, 26]. Two of the most important kernels are sparse times dense matrix multiplication (SpMM), where the dense matrix is tall and skinny, i.e., representing a set of vectors, and sparse times sparse matrix multiplication (SpGEMM). Sparse matrix-vector multiplication is also an important kernel in many applications, and has been studied even more extensively, although our focus here is on the matrix-matrix operations.
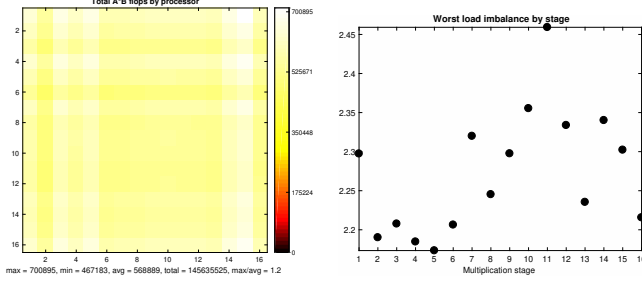
The large majority of prior work on distributed-memory sparse matrix kernels focuses on bulk synchronous implementations based on SUMMA [29], using collective operations, typically broadcast and/or reduce, in the inner loop

of the algorithm to move data to the processes that need it for their local matrix multiplications. While SUMMA-like algorithms have the advantages of being straightforward to implement and generally scaling well, for sparse problems, they have the disadvantage of forcing processes to proceed in lockstep in the inner loop of the algorithm. When there is load imbalance, either in the amount of computation that must be performed locally by each process, the amount of data that must be transferred to each process, or both, this can potentially result in decreased performance. This can occur even when the total amount of work performed by each process is the same, since processes may have differing amounts of work in each iteration. To illustrate, Figure 1 shows two kinds of load balance for squaring a randomized sparse matrix generated by the R-MAT model [12], with parameters $a = 0.6$, $b = c = d = 0.4/3$, edgefactor 8, and scale 17. The total (end-to-end) computation has only 20% load imbalance whereas the synchronization points in between stages amplify the load imbalance to $\approx 2.3\times$. The load imbalance is defined as the max/avg ratio, the ratio of maximum number of flops performed by any processor to the average number of flops per processor.

Load imbalance is often addressed by randomly permuting the rows and/or columns of the sparse matrices, but this has disadvantages, the foremost being that the required permutation of the inputs, followed by permutation of the outputs, can be expensive. Furthermore, permuting the matrix may possibly remove structural locality, resulting in decreased performance of local matrix multiply operations. For example, Slota et al. [25] showed that the loss of locality due to random permutations can hurt the performance of graph algorithms significantly where parallel PageRank performs $2 - 5\times$ slower on a randomly permuted graph compared to other orderings. Even if we ignore the loss of locality or try to recover it by local reordering after a global random permutation, there are theoretical limits on how much load balance random permutations can achieve. For matrices with dense rows or columns, the load balance can deteriorate with $\sqrt{p}$ for large p, where p is the number of processors in the worst case, per Theorem 5.4 of Azad et al. [4].

Asynchronous algorithms, which do not require synchronization or coordination between processes as in bulk synchronous algorithms, are an approach to dealing with this load imbalance in distributed sparse matrix multiplication.

(a) End-to-end load imbalance when processors do not synchronize across 16 stages. The heat plot has $256 = 16 \times 16$ boxes, which demonstrate the max/avg load imbalance is ≈ 1.2.

(b) Per stage max/avg load imbalance of the same algorithm. An implementation that synchronizes in between stages would consequently incur load imbalance of ≈ 2.3.

Figure 1. Total (end-to-end) vs. per-stage load balance multiplying a R-MAT model-generated sparse matrix with a sparse 2D algorithm. Simulated on a 16×16 process grid.

In this paper, we develop dense and sparse matrix data structures that use RDMA, meaning that a process can manipulate remote parts of the matrix using one-sided RDMA put and get operations, without requiring synchronization or coordination with the remote process. Using these dense and sparse matrix data structures, we then design and implement a number of different *RDMA-based, asynchronous* matrix multiply algorithms, which remove synchronization from the inner loop, allowing each process to proceed independently. In addition, we develop and implement an RDMA-based workstealing algorithm, which allows processes that have finished early to steal work from processes that are overburdened using a lightweight reservation scheme.

Furthermore, we examine the performance of these different sparse matrix multiply algorithms running across multiple GPUs in a distributed memory environment. We examine the challenges presented by the highly compute dense nodes of modern GPU-based supercomputers, as well as the benefits of direct GPU-to-GPU transfers offered by new network technologies like NVIDIA GPUDirect RDMA and NVLink.

The main contributions of this paper are:

- Design and implementation of RDMA-based distributed dense and sparse matrix data structures for GPUs.
- New asynchronous, RDMA-based algorithms for SpMM and SpGEMM, including workstealing algorithms.
- A roofline-based performance model, to characterize the performance of our RDMA-based implementations.
- An in-depth performance analysis of our RDMA-based implementations, along with a comparison to traditional bulk synchronous methods.

2 Background

Sparse times dense matrix multiply (SpMM) and sparse times sparse matrix multiply (SpGEMM) are two important sparse linear algebra primitives. SpMM is used in a variety of blocked iterative methods, graph algorithms, as well as graph neural networks (GNNs) [19, 20, 28]. In these contexts, SpMM typically involves multiplication of a sparse matrix with a tall and skinny dense matrix, where the number of columns of the sparse matrix varies between 32 and 1024. SpGEMM is also used in graph algorithms, including betweenness centrality, triangle counting, cycle detection, and Markov clustering, as well as in applications such as in genomics [3, 10, 11, 13, 14].

2.1 Distributed Matrix Algorithms

Distributed matrix algorithms allow multiple processes to work together to execute a single matrix multiply operation. Let's first consider a distributed matrix multiply computing the product $C = AB$. Here, each of the matrices C, A, and B is split up into tiles which live on different processes. The matrix tiles will be sparse or dense, depending on what kind of matrix multiplication is being performed. Let's assume that C is distributed among a grid of $M \times N$ tiles, A is distributed among a grid of $M \times K$ tiles, and B is distributed among a grid of $K \times N$ tiles. The algorithms we consider in this paper rely on a tiling scheme that is regular on the indices. In other words, each tile has the same dimensions, modulo differences due to the tile size not dividing the matrix size evenly. A different, more irregular tiling mechanism might be more appropriate for matrices from certain applications such as those involving linear-scaling quantum-chemical calculations [8], where the near-sightedness creates natural block-sparse structure. However, the vast majority of application targets (GNNs, graph algorithms, eigenvector computations, and tensor decompositions) do not have this natural block-sparse structure, so the added complexity of irregular tiling is not justified for us.

To compute a single tile of the C output matrix $C_{i,j}$, we must compute the result $C_{i,j} = C_{i,j} + \sum_{k=0}^{K} A_{i,k} B_{k,j}$, where the indices i, j, and k index into tiles of the matrices. The computation can be thought as a 3D cube, which can be assigned to processors in different ways, which result in 1D, 2D, and 3D structured algorithms. We are then left with the choice of how to assign pieces of the computation to processors, as well as how to communicate tiles of the matrices necessary to execute the matrix multiply. One of the most common, and usually easiest to implement, distribution methods is the *stationary C* method, in which the process that owns each block of C will be responsible for executing each of the local matrix multiplications necessary to compute the result for that block. This means that the A and B matrices will be communicated, while tiles of the C matrix will be available locally. For each tile of the matrix $C_{i,j}$, the process that owns

that tile will be responsible for calculating the output block $C_{i,j} = C_{i,j} + \sum_{k=0}^{K} A_{i,k} B_{k,j}$.

Similarly, we could organize the computation so that the owner of each tile of A will be responsible for executing each of the local matrix multiplication operations in which it is used. In this *stationary A* method, tiles of A will thus be available locally, while tiles of B will be fetched through some method, and output updates to C will be sent and accumulated at their final location. This means that for each tile of A $A_{i,k}$, the process that owns that tile will be responsible for executing $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$ for all j. Note that this means that the tile of B $B_{k,j}$ must be somehow retrieved from remote memory, while the result of each local matrix multiply $C_{i,j}$ must be sent and accumulated to the corresponding block of C. The *stationary B* method is similar, except that B remains stationary, while A must be communicated.

2.2 Bulk Synchronous SUMMA

In bulk-synchronous SUMMA, collective broadcast operations are used to communicate tiles of the matrix. In SUMMA, communicators are created for each tile row of the A matrix and each tile column of the B matrix. In each iteration of the algorithm, a block of the A matrix is broadcast in each row communicator and a block of the B matrix is broadcast in each column communicator. Each process will compute its corresponding local matrix multiplication, accumulating into its local block of C. Stationary A (or B) algorithms are also possible with SUMMA, by replacing one of the broadcasts with a collective reduction operation to accumulate each block of C into the correct place.

As discussed in Section 1, variation in the number of nonzeros, as well as the sparsity patterns, of individual tiles of sparse input matrices can create load imbalance in computation due to the differing numbers of flops that must be performed in each local matrix multiply, as well as load imbalance in communication, due to the different amounts of data that must be transferred by each process. Bulk synchronous SUMMA-like algorithms can suffer due to these load imbalances.

2.3 RDMA and Asynchrony

Remote Direct Memory Access (RDMA) is a hardware capability offered by the Network Interface Cards (NICs) in most modern supercomputers and datacenters. Using RDMA, a process can expose a region of its memory, often referred to as "shared memory," to be directly accessible by remote processes. Processes can then issue put, get, and atomic operations to remotely access and manipulate the shared memory regions of other processes. These RDMA operations are handled directly by the NIC on the remote node, hence no coordination is required in order to access the memory of a remote process. Some of the commonly used frameworks that support direct RDMA access include OpenSHMEM, GASNet-EX, and MPI's one-sided communication API. For distributed

programs that run on GPUs, systems with Infiniband can currently take advantage of GPUDirect RDMA, which allows direct transfers from GPU-to-GPU over the network. NVSHMEM, an extension of OpenSHMEM from NVIDIA, allows for direct transfers of data between GPUs, either within a node over PCIe or NVLink, or between nodes using GPUDirect RDMA over Infiniband. In this paper, we use NVSHMEM to directly copy between GPUs. A number of different libraries have been developed that use RDMA primitives to build data structures and algorithms, with some focusing particularly on irregular data structures and algorithms, where asynchronous RDMA primitives can be particularly beneficial, both for performance and software engineering reasons [6, 9, 15].

3 RDMA-Based Algorithms

3.1 Data Structures

Unlike bulk synchronous SUMMA algorithms, RDMA-based algorithms use direct RDMA put and get operations to access remote data. In order to implement RDMA-based algorithms, we first developed RDMA-based dense and sparse matrix data structures to support direct remote access to tiles. In our data structures, each process holds a directory of *global pointers*, which are objects that reference remote data that can be accessed over RDMA. In the dense case, each process holds a global pointer for each remote tile, and it can issue put or get operations to read or write to the remote matrix tile. In the sparse matrix data structure, the directory holds three global pointers for each tile, which point to the values, row pointer, and column indices arrays that constitute a Compressed Sparse Row (CSR) format sparse matrix.

3.1.1 Reading Tiles. To fetch a remote tile of the matrix is fairly straightforward. For a dense matrix, we issue a single RDMA get operation to copy the data from the remote node to the process's local memory. For a sparse matrix, we issue three get operations to retrieve each of the remote CSR arrays. Note that since we use RDMA, these operations are completely asynchronous, and do not require coordination with the remote process. We implement both blocking and non-blocking versions of the operation.

3.1.2 Modifying Remote Tiles. Modifying a local tile is straightforward, since we can use local pointers which point to a local tile of the matrix to directly modify the data. Modifying a remote tile, which is only required for some of the RDMA algorithms, is somewhat more complex, particularly when the remote tile happens to be sparse. To deal with this, we use a system of remote queues to issue asynchronous updates to the remote tiles. Each process has a globally visible queue, and other processes can push updates to this queue in order to send updates that the remote process needs to apply to tiles that it owns. During algorithms that require accumulations to remote tiles, each process will periodically check to see if there are elements waiting in the

queue, and if so, dequeue the elements and accumulate the tile. The element inserted into the queue is a lightweight pointer to the submatrix that needs to be accumulated, so the dequeueing process will issue get operations to retrieve the data before accumulating. Push and pop operations are performed atomically to ensure no updates are lost.

3.2 Algorithms

3.2.1 RDMA Stationary C Algorithm.
The most straightforward RDMA-based algorithm is the stationary C algorithm, in which each processor iterates through its tiles of the C matrix, and, for each tile, iterates through the corresponding row block of A and column block of B, retrieving each of the remote tiles via get_tile and multiplying them together using a local matrix multiply operation. Since the method get_tile does not require any coordination with remote processors, each process can execute its work independently, and does not need to wait on or synchronize with other processors. In order to achieve good performance, a few optimizations are required, including prefetching tiles for the next iteration, which allows overlap of computation and communication, and an iteration offset in the inner loop, which balances which tile is requested within each row and column and generally ensures the first RDMA get is to the local tile. These optimizations are discussed in detail in Section 3.3.

Algorithm 1 RDMA-based 2D A-Stationary SpMM. M, N, and K represent tile dimensions.

```
for i in 0..M-1:
  for k in 0..K-1:
    if A.owner(i, k) == rank():
      for j in 0..N-1:
        local_a = A.tile_ref(i, k)
        local_b = B.get_tile(k, j)
        local_c = local_a*local_b
        queue[C.owner(i, j)].push(get_ptr(local_c))

while local_c_ptr = queue[rank()].pop():
  C.my_tile() += get_tile(local_c_ptr)
barrier()
```

3.2.2 RDMA Stationary A and B Algorithms.
The RDMA-based stationary A algorithm is similar to the stationary C algorithm, except we assign work based on which processor owns which tile of A. Each process iterates through its tiles of A, and, for each tile, iterates through the corresponding tile row of B, pulling in each tile and performing a local matrix multiply. Each of these partial results must be accumulated into different tiles of C, each of which is potentially owned by a different process. In order to accumulate these partial results into their individual tiles of C, a global pointer to the local result is pushed to a queue on the remote process where it needs to be accumulated. At some point, the remote process will pop the pointer off of its queue, retrieve the

remote partial result tile, and accumulate it into the correct local tile. Pseudocode for the RDMA-based stationary A algorithm is shown in Alg 1. The stationary B algorithm is very similar, except work is assigned based on which processes own which tile of B.

3.3 Optimizations

Two important optimizations are necessary in order to achieve good performance for the RDMA-based algorithms described here. First, we use the non-blocking version of get_tile to asynchronously retrieve a remote tile of the matrix. This allows us to prefetch the tiles needed for the next iteration before entering into the local matrix multiply operation, creating overlap of communication with computation. Second, we apply an iteration offset to the inner loop, which ensures two things: (1) it spaces processes apart, so that not all processes will request the same tile in their row and column at the same time, and (2) it generally ensures that the first remote get issued is to a local tile, which helps jumpstart communication and ensures that almost all communication can be overlapped with computation. For the stationary C algorithm, we apply an iteration offset of $i + j$, while in the stationary A and stationary B algorithms, we apply an iteration offset of $i + k$ and $k + j$, respectively. Pseudocode which demonstrates these optimizations applied to a stationary C algorithm is shown in Alg 2.

Algorithm 2 Optimized RDMA-based 2D C-Stationary SpMM. M, N, and K represent tile dimensions.

```
for i in 0..M-1:
  for j in 0..N-1:
    if C.owner(i, j) == rank():
      k_offset = i + j
      buf_a = A.async_get_tile(i, k_offset % K)
      buf_b = B.async_get_tile(k_offset % K, j)
      for k_ in 0..K-1:
        k = (k_ + k_offset) % K
        local_a = buf_a.get()
        local_b = buf_b.get()
        local_c = C.tile_ref(i, j)

        if (k_ + 1 < K):
          buf_a = A.async_get_tile(i, (k+1) % K)
          buf_b = B.async_get_tile((k+1) % K, j)
        local_c += local_a*local_b
barrier()
```

3.4 Workstealing Algorithms

In addition to supporting generally asynchronous implementations of distributed matrix multiply, RDMA also allows for more straightforward implementations of workstealing algorithms, in which processes that have finished their work can steal work from processes that are overburdened. In our workstealing algorithms, we use lightweight 2D and 3D reservation schemes to allow processors to claim work. First,

processors attempt to perform their work as normal, iterating through each of the tiles that they own. However, before performing any work, they first issue a remote fetch-and-add operation, executed using RDMA, to reserve the work. If some of a process' work is stolen, it will move on to the next piece of work available. After it has completed all of its normal work, each process will iterate through a number of other processes, checking to see if they have work available to be stolen.

Random workstealing. The simplest workstealing strategy is random workstealing, where work is stolen randomly without regard to locality. This has the advantage that many blocks are available to be stolen, with the disadvantage that performing stolen work will usually be more expensive than performing regular work, since the tiles of A, B, and C must all be communicated, as opposed to just two. Random workstealing uses a 2D work grid corresponding to the tiles of the stationary matrix. Each element in the work grid contains an integer that is initialized to zero. To claim a piece of work involving a tile, processes perform a fetch-and-add operation on the corresponding element of the work grid. The integer value returned corresponds to the piece of work that has been claimed. Pseudocode for this workstealing algorithm is shown in Alg 3 using the stationary A method.

Locality-Aware workstealing. In locality-aware workstealing, instead of stealing randomly, each process will only steal pieces of work for which it owns one of the components, ensuring a lower cost of performing stolen work. For example, in a stationary C locality-aware work-stealing algorithm, processes will first attempt to perform matrix multiplies involving their tiles of C. After these are completed, each process will attempt to steal work involving their tiles of A and B. Locality-aware workstealing requires a 3D work grid, where element i, j, k corresponds to the component matrix multiply $C[i, j] + = A[i, k] * B[k, j]$. There is a slightly higher overhead associated with this 3D work grid, which requires a separate remote fetch-and-add for each piece of work performed, but this is largely offset by the improvement in locality.

4 Performance Model

In this section, we build a performance model to characterize and predict the performance of our RDMA-based matrix multiply algorithms, first computing the expected cost of communication before using this communication analysis to build an *inter-node* roofline model, which characterizes performance in terms of the ratio of network traffic to local work.

Assume a two-dimensional grid of matrix tiles, where the matrices A ($m \times k$), B ($k \times n$) B, and C ($m \times n$) are distributed amongst p processors laid out in $\sqrt{p} \times \sqrt{p}$ tile grids. Tile dimensions for A, B, and C are then $\frac{m}{\sqrt{p}} \times \frac{k}{\sqrt{p}}$, $\frac{k}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$,

Algorithm 3 Stationary A SpMM with random workstealing.

```
def attempt_work(i, k):
  # Remote atomic fetch-and-add to reserve work
  my_j = reserve_grid[i, k]++
  while my_j < N:
    local_a = A.get_tile(i, k)
    local_b = B.get_tile(k, j)
    local_c = local_a*local_b

    queue[C.owner(i, j)].push(get_ptr(local_c))
    my_j = reserve_grid[i, k]++

# Do work for my tiles
for i in 0..M-1:
  for k in 0..K-1:
    if A.owner(i, k) == rank():
      attempt_work(i, k)

# Attempt to steal work
for idx in 0..M-1:
  i = (rank()+idx) / M
  k = (rank()+idx) % M
  attempt_work(i, k)

while local_c_ptr = queue[rank()].pop():
  C.my_tile() += get_tile(local_c_ptr)
barrier()
```

and $\frac{m}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ respectively, with $\sqrt{p}$ tile columns and $\sqrt{p}$ tile rows in each matrix. For a dense matrix, the total number of elements in each tile is the product of its dimensions, while for a sparse matrix we approximate the number of nonzeros per tile as the matrix density d multiplied by the product of the tile dimensions.

For our stationary C RDMA-based algorithm for SpMM, in each iteration, each process will issue two RDMA get operations, one to retrieve a sparse tile of A, and one to retrieve a dense tile of B. Therefore, in each iteration, each process will retrieve exactly $\frac{kn}{p}$ elements in the dense tile of B, along with $\frac{dmk}{p}$ elements in a CSR data structure, for a total communication cost of $\frac{kn}{p} + 2\frac{dmk}{p} + \frac{m}{\sqrt{p}} + 1$ elements.

If we offset the order of iteration of the k loop by $i + j$, as discussed in Section 3.3, we can also ensure that communication will be balanced with each process sending exactly two tiles per iteration, one of the A matrix, and one of the B matrix. To prove this, assume tile (i, j) is stored on processor $i(N + 1) + j$, where $i \in [0, N)$, $j \in [0, N)$, and $N = \sqrt{p}$.

Proof. Towards a contradiction, assume that $i(N + 1) + j$ is not unique. Then, $i(N + 1) + j = i'(N + 1) + j'$. Assume $i' > i$. Then $j = (i' - i)(N + 1) + j'$. Since $i' - i \geq 1$, $j > N + 1$, which is a contradiction. □

With the offset, each process sends and receives exactly $\frac{kn}{p} + 2\frac{dmk}{p} + \frac{m}{\sqrt{p}} + 1$ elements over the network in each iteration. In a fully connected network, each of these transfers is independent and will not interfere with each other. Many

modern datacenter networks, such as that used by Summit, are indeed fully connected fat trees where this assumption holds completely. Many other network topologies, such as Dragonfly, approximate a fully connected network.

Next, we present an *inter-node* roofline model that predicts the performance of each iteration of the distributed matrix multiplication. The roofline model predicts the maximum possible performance of a computational kernel depending on its *arithmetic intensity*, or number of operations performed per byte that must be fetched from memory. Typically, the roofline model is used to characterize serial or multithreaded compute kernels in terms of how compute or bandwidth intensive they are, where the limit on "compute" is defined by the processor's arithmetic peak, and "bandwidth" is determined by memory bandwidth. Here, we develop an *inter-node* roofline model to characterize performance of our distributed memory, RDMA-based matrix multiply algorithms. In this instance, the bandwidth involved is network communication, and the roofline peak of local operations serves as our "arithmetic peak.". In our *inter-node* roofline model, we compute the inter-node arithmetic intensity of each iteration as the number of flops performed divided by the number of bytes that must be communicated over the network. The flat portion, providing the "roof" on performance of our inter-node roofline, is the *local* roofline peak for our local SpMM or SpGEMM calls.

We first build a regular *local* roofline model to characterize the performance of our local matrix multiplies. For local SpMM and SpGEMMs, precise roofline models can be difficult to compute, often requiring analysis of the actual matrices themselves, since they depend not only on the size of the matrices and the number of nonzeros, but also the sparsity patterns of the matrices involved. For SpMM, we can calculate an approximate upper bound on arithmetic intensity as the number of flops to be performed in the SpMM divided by the total size of the sparse and dense matrices in bytes. This upper bound assumes perfect cache performance for B and C, and depending on the nonzero pattern, values may have to be reloaded from memory, resulting in lower performance. For a local SpMM operation multiplying a tile of A times B, with the tile sizes derived above, and adding the variable w, which is the number of bytes per word, we compute the arithmetic intensity of our local SpMM operations.

$$local\ SpMM\ AI\ =\ \frac{2(\frac{dmk}{p})(\frac{n}{\sqrt{p}})}{w(2\frac{dmk}{p}+\frac{m}{\sqrt{p}}+1+\frac{mn}{p}+\frac{kn}{p})}$$

That is, the number of flops to be performed, divided by the total number of bytes in A, B, and C. To calculate the local roofline peak, which is the "roof" on our inter-node roofline model, we multiply this arithmetic intensity by the memory bandwidth of the local processor B, followed by a max operation with the arithmetic peak.
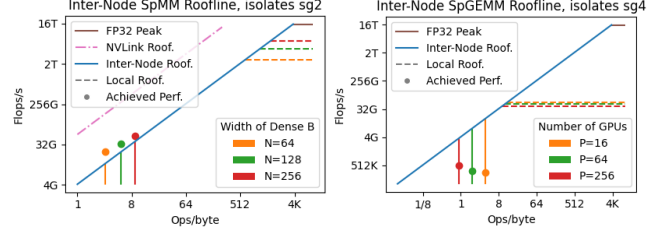

Figure 2. *Inter-node* roofline plots for SpMM and SpGEMM with a 2D distribution. SpMM plot models performance for different widths of the dense B matrix at a fixed scale (24 GPUs), while SpGEMM models performance at different scales. Dashed horizontal lines represent *local roofline peaks* for SpMM and SpGEMM operations, while vertical lines represent *inter-node* roofline peaks for particular problems.

To compute the arithmetic intensity of an iteration of our distributed SpMM algorithm using the *inter-node* roofline model, we divide the number of flops performed, which is the same expression from the numerator of the local SpMM arithmetic intensity, by the total number of bytes to be communicated over the network, equal to the total number of bytes in the tiles of A and B.

$$inter\text{-}node\ SpMM\ AI\ =\ \frac{2(\frac{dmk}{p})(\frac{n}{\sqrt{p}})}{w(2\frac{dmk}{p}+\frac{m}{\sqrt{p}}+1+\frac{kn}{p})}$$

We can then use this roofline to characterize the performance of SpMM on the Summit supercomputer. Figure 2 shows the roofline plot for our inter-node SpMM roofline model, using the isolates subgraph2 sparse matrix, and evaluating performance for various numbers of columns in the dense matrix. In the plot, the blue sloping portion represents the region that is bound by network communication, and as such the bandwidth-bound portion has a slope of 3.83 GB/s, which is each GPU's share of injection bandwidth on Summit. The brown roofline at the top is the 32-bit floating point arithmetic peak, 16 TFlops/s for an Nvidia Tesla V100 GPU. Each of the other dotted horizontal lines represents a *local* roofline peak, which in the inter-node roofline model replaces the arithmetic peak. The vertical solid lines represent the roofline peak for a particular problem size, while the dots indicate achieved performance of our RDMA algorithm. Since all three of the problem sizes plotted are well into the bandwidth-bound portion of the inter-node roofline plot, we expect them to be bound by network communication. The SpMM roofline model was across 24 GPUs. Note that the achieved performance slightly exceeds the roofline bound based on newtork bandwidth. This is because of the high-speed NVLink interconnect providing higher bandwidth for intra-node transfers.

While we can also construct an inter-node roofline model for SpGEMM, it is not possible to write a general formula for FLOPs performed for a particular matrix size and density. This is because in SpGEMM, the number of FLOPs performed

depends on the particular sparsity patterns of the matrices, and as such the number of FLOPs can vary significantly for matrices with the same dimensions and number of nonzeros. In light of this, we use the function FLOPS(A, B) to represent FLOPs performed, and in our roofline plot we use average FLOP values calculated experimentally. As such, we can represent the inter-node roofline model's arithmetic intensity as the number of flops performed divided by the size of A and B.

$$\text{inter-node SpGEMM AI} = \frac{\text{FLOPS}(A, B)}{w(\frac{2dmk}{p} + \frac{m}{\sqrt{p}} + 1 + \frac{2dkn}{p} + \frac{k}{\sqrt{p}} + 1)}$$

For our local roofline model, we use the bound on local SpGEMM arithmetic intensity by Gu, *et al.*, which is representative of modern SpGEMM algorithms and expresses arithmetic intensity in terms of the compression factor cf, the number of flops performed per nonzero output, and the number of bytes to express each nonzero b.

$$\text{local SpGEMM AI} = \frac{cf}{(3 + 2cf) * b}$$

Note that the roofline model is dependent on the sparsity structure, since cf depends on the particular sparse matrices being multiplied. Our inter-node SpGEMM roofline model is plotted in Figure 2. To obtain realistic values for cf and the number of flops performed *FLOPS(A, B)*, we performed each of the component local SpGEMM operations in the distributed SpGEMM operation for the isolates subgraph4 matrix, recording their values for cf and *FLOPS(A, B)* to compute average values for different numbers of GPUs P. In the SPGEMM plot, inter-node roofline peaks are much closer to their local roofline peaks than in the SpMM plot. Thus, our roofline model suggests SpGEMM is significantly less network communication-bound than SpMM, although still communication bound. We discuss more conclusions from our roofline model in Section 6.

5 Implementation

5.1 Communication Layer

As discussed in Section 2.3, the current generation of GPU-based supercomputers offers GPUDirect RDMA over Infiniband networks, which ensures that issuing one-sided operations will be efficiently executed in hardware. In GPUDirect RDMA, the CPU prepares an Infiniband request, which it sends to the local network interface card (NIC). When the NIC receives the request, it will copy the data directly over the network from GPU to GPU, without staging data through the CPU. When copies take place between GPUs within the same node, the high-bandwidth NVLink fabric is used.

All of our RDMA-based implementations use NVSHMEM for communicating data between GPUs. NVSHMEM uses GPUDirect RDMA over Infiniband to transfer data when the GPUs involved are on separate nodes, and NVLink to

transfer data when the GPUs involved are within the same node.

5.2 Data Structures

We implement sparse and dense distributed matrix data structures using BCL, an RDMA-based distributed data structures library in written in C++. Both dense and sparse matrix data structures split the distributed matrix up into evenly sized tiles, which are then assigned to processors using a processor grid, as in ScaLAPACK. As discussed in Section 3.1, each process has a copy of a directory of remote pointers, which point to the matrix tiles stored in pinned memory, directly accessible through RDMA operations using NVSHMEM. In the dense case, a single remote pointer points to a dense tile data structure, and in the sparse case three remote pointers point to the values, row pointer, and column index arrays of a compressed sparse row (CSR) data structure. All data is stored directly on GPUs, and can be copied directly from GPU to GPU over the network using BCL's NVSHMEM backend.

5.3 Data Access Primitives

In terms of data access primitives, all direct manipulation of data is performed using put and get RDMA operations. To retrieve a tile of the dense matrix, a process will issue an RDMA get operation to retrieve all or part of a remote tile. Similarly, to write to the tile, a process can issue a remote put operation. In the sparse case, reading from a tile involves issuing remote get operations to retrieve each of the value, row pointer, and column index arrays in a CSR data structure. Due to the complexities involved in modifying a CSR data structure in place, only the process that owns a sparse tile is permitted to modify it. This is done by calling a function `replace_tile()` to replace the old tile with a new one, followed by a collective function `renew_tiles()` that will make all sparse tile modifications visible to other processes. As discussed in Section 3.1.2, when a process needs to send an update to a remote tile, as in the A and B stationary algorithms, we use a distributed queue to send a pointer to the update to the process that owns a particular tile, who will then perform an accumulation at the destination tile. For the update queue, we use BCL's `CheckSumQueue`, which enqueues a pointer using a single fetch-and-add operation and an RDMA put, while allowing simultaneous enqueues and dequeues.

As discussed in Section 3.1.1, the primary primitive for accessing remote tiles of the matrix is `get_tile()`, which fetches a remote tile of the matrix, and has the same API for both dense and sparse matrices. In the asynchronous version, we return a future object, which will return the local object when the method `get()` is called. In order to ensure the best possible performance and simplify memory management, we allocate most of each GPU's memory as shared memory that can be addressed remotely by NVSHMEM, and use our own memory allocator to allocate memory when necessary.

This (1) ensures that the local destination buffers in remote get operations are allocated in the shared memory segment, which is a requirement for NVSHMEM transfers over Infiniband, and (2) reduces the overhead of memory allocation during the algorithm, since `cudaMalloc` tends to be much less efficient than using a custom memory allocator.

5.4 MPI SUMMA Implementations

For one comparison benchmark, we also implement the bulk synchronous SUMMA algorithm using CUDA-aware MPI. In the MPI implementations, we use `MPI_Bcast` to broadcast tiles of the matrix using row and column communicators. We use the CUDA-aware API in IBM Spectrum MPI to perform our broadcasts directly on data that sits on the GPU, and IBM Spectrum MPI is configured to take advantage of GPUDirect RDMA to copy data directly from GPU-to-GPU over the Infiniband network. Note that the MPI SUMMA implementation only runs on square processor grids, so we run this implementation on perfect square numbers of processors.

6 Evaluation

We evaluated our RDMA-based sparse matrix multiply implementations in both single-node and multi-node environments. Multi-node experiments are run on the Summit supercomputer at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory. A Summit node has 6 Nvidia Tesla V100 GPUs, each with 16 GB of memory, and the nodes are connected with dual-rail EDR InfiniBand with a link bandwidth of 23 GB/s. Within a node, the GPUs are connected with Nvidia's NVLink interconnect. Single-node experiments are run on a DGX-2 system that is part of the Bridges-2 system at the Pittsburgh Supercomputing Center. The DGX-2 has 16 Nvidia Tesla V100 GPUs with 32 GB of memory, which are fully connected by Nvidia's NVLink interconnect. Both systems use NVLink 3.0, which provides 50 GB/s of link bandwidth. We ran experiments for matrices whose single-GPU runtime took less than a second in the single-node DGX-2 environment, while we used the multi-node environment for matrices which took longer.

On Summit, all codes were compiled with GCC 8.1.1, CUDA 11.2.0, and IBM Spectrum MPI 10.3.1.2. On the single-node DGX-2 system, codes were compiled with GCC 10.2.0, CUDA 11.1.0, and Open MPI 4.0.5. Our RDMA-based implementations use NVSHMEM 2.0.2. All performance experiments use CuSPARSE for local matrix computation. In addition to our own SpMM and SpGEMM implementations, we also benchmark CombBLAS's GPU SpMM and PETSc's GPU SpGEMM [7, 10]. We corresponded with CombBLAS's authors on how to build and execute the gpu branch of CombBLAS on Summit, and with OLCF staff for how to properly build PETSc with GPU support.

The sparse matrices in our experiments are detailed in Table 1. Load imbalance is defined as $max(\text{nnz}(A_{i,j}))/ave(\text{nnz}(A_{i,j}))$

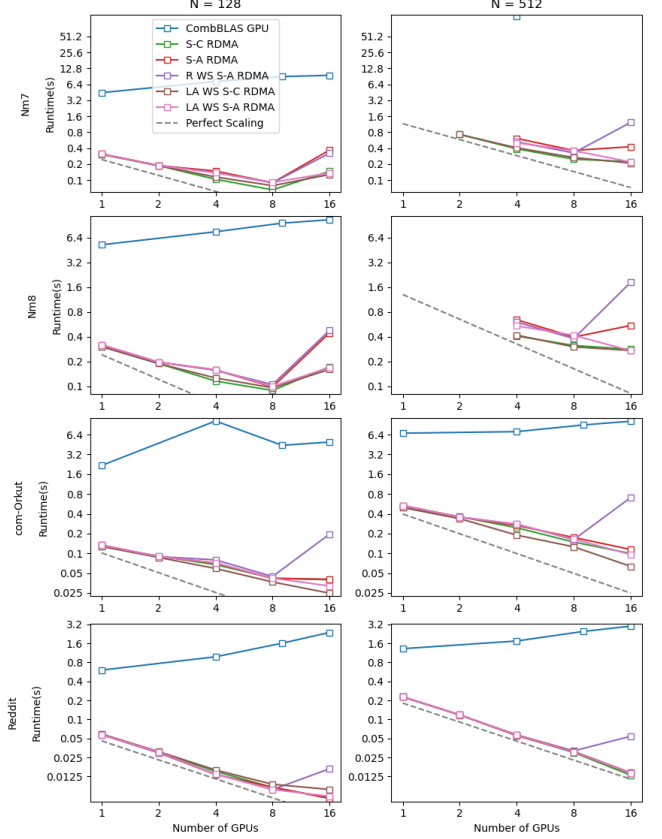**Figure 3.** Single-node runtimes for SpMM, with different numbers of columns N in the dense matrix B.

Table 1. Matrices used in our experiments. "load imb." lists the imbalance in number of nonzeros when split amongst 100 processes on a 10×10 2D process grid.

Sparse matrix (A)	kind	$m = k$	nnz(A)	load imb.
Mouse Gene	Biology	45.1K	29.0M	2.13
ldoor	Structural	952K	46.5M	8.23
reddit	GNN	233K	115M	1.08
nlpkkt160	NLP	8.3M	230M	9.46
Amazon Large	GNN	14.3M	230M	3.78
com-Orkut	NMF	3.1M	234M	8.15
Nm7	Eigen	5.0M	648M	6.38
Nm8	Eigen	7.6M	592M	6.48
Isolates, Subgraph4	Biology	4.4M	327M	1.00
Isolates, Subgraph2	Biology	17.5M	5.2B	1.00
Friendster	Graph	62.5M	3.4B	7.68

where $A_{i,j}$ is the matrix assigned to a single processor $P(i, j)$. We multiply these matrices against dense matrices with 128 and 512 columns. All matrices use 32-bit floating point values, and for indices, we use 32-bit integers except on "Isolates, Subgraph2" and "Friendster," which require 64-bit indices due to their size.

6.1 SpMM Experiments

To evaluate our RDMA-based algorithms for sparse times dense matrix multiplication (SpMM), we performed strong scaling experiments using a number of different sparse matrices multiplied by dense matrices of different widths.

Figures 3 and 4 show the results for a variety of different algorithms. The RDMA-based algorithms include a stationary C algorithm ("S-C RDMA"), a stationary A algorithm ("S-A RDMA"), stationary A algorithm with random workstealing ("R WS S-A RDMA"), and locality-aware workstealing with stationary A and C distributions ("LA WS S-C RDMA" and "LA WS S-A RDMA"). Bulk synchronous benchmarks include our own CUDA-aware MPI SUMMA implementation ("BS SUMMA MPI"), and CombBLAS's GPU SpGEMM kernel ("CombBLAS GPU"). We use matrix sizes representing typical workloads used in a range of real-world applications, such as Graph Neural Networks (GNNs), iterative methods, and graph algorithms[19, 20, 28].

The implementations generally scale well up to at least 8 GPUs on the single-node DGX-2 system in Figure 3, where each GPU has 50 GB/s of bandwidth. Some versions suffer a slowdown at 16 GPUs, especially on the smaller Nm7 and Nm8 matrices, where at 128 columns of B there is insufficient work to keep the GPUs fully occupied and still amortize data movement. This is particularly true for the stationary A algorithms because of an increase in the number of accumulation operations that become necessary as the number of tiles, and thus the number of local matrix multiply operations, increases.

For the larger matrices run on distributed memory in Figure 4, the shared links give each GPU only 3.8 GB/s on average. This significantly limits scalability, especially for the communication-bound case with a smaller dense B matrix. However the RDMA-based implementations outperform the bulk synchronous implementations for most of these configurations, and that benefit is greatest when the number of columns in the dense matrix B is lowest. This is likely because these problems have less computation and are therefore more sensitive to communication cost, both message latency and load imbalance from communicating unequal size blocks of A. A larger B matrix increases the both computation time and time for well-balanced communication of B. The stationary A algorithms have the same issue of increased accumulation cost as on the single node system, but due to the communication-bound nature of the multi-node problems, the overall effect is not as pronounced.

The multi-node scaling results are also influenced by the fact that some transfers happen over the local NVLink fabric, which is much faster than Infiniband. As the number of GPUs increases, the number of transfers that occur over local NVLink fabric decreases, and the lower bandwidth between nodes plays a larger role.
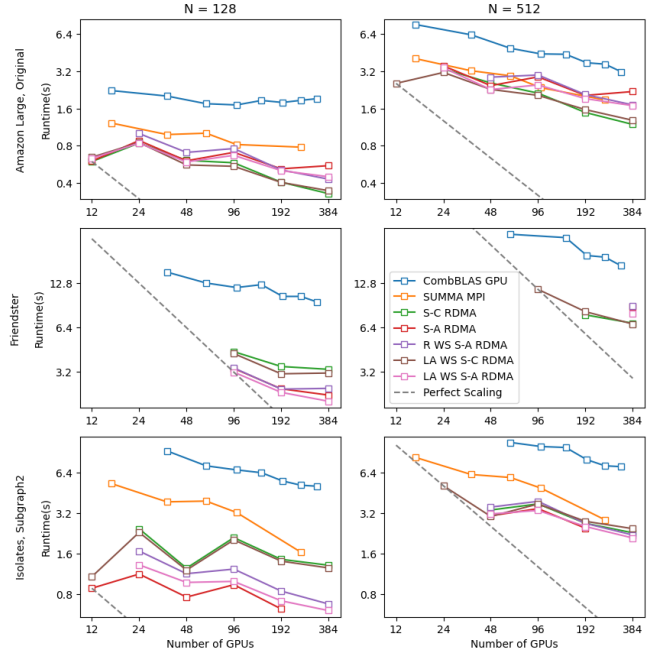

Figure 4. Multi-node runtimes for SpMM, with different numbers of columns N in the dense matrix B.

Scaling performance is best on larger matrices, such as isolates, and scaling is also better with wider B matrices. The reasons for this are likely twofold: (1) there is simply more work to be done with a larger dense matrix, and so scaling performance is better, and (2), as demonstrated by our inter-node roofline performance model in Figure 2, the wider the B matrix, the more arithmetically intense the operation becomes, and the less bound by network communication.

The relative performance of the RDMA SpMM algorithms generally depends on the relative sizes of the A, B, and C matrices. If one of the matrices is particularly large, then it is most efficient to keep that matrix stationary while communicating the two other, smaller matrices. When the sparse matrix is not too large in relation to the sizes of the dense matrices, stationary C performs best, which follows from the fact that it performs accumulations locally, removing the need for remote queues, which add some additional overhead. For problems where the sparse matrix is sufficiently larger than the dense matrices, the stationary A algorithm performs best. This trend can be seen in the multi-node plots for the matrix friendster, where stationary A performs better for $N = 128$ (meaning the dense B matrix is small), but that is reversed for $N = 512$ (meaning the dense B becomes bigger). As with most real-world graph applications, all of our sparse matrices are square, which means that the B and C matrices are the same size. This means there is no benefit in terms of communication volume to leaving B stationary instead of C, even while stationary B adds additional overhead due to remote queues. As such, we do not explore a stationary B implementation for SpMM.

Our RDMA-based workstealing algorithms are able to achieve performance improvements for some of the less load balanced matrices, such as com-Orkut and friendster, particularly at higher node counts, where the smaller amount of work per node is more likely to lead to some nodes being left with uneven amounts of work. The locality-aware workstealing algorithms tend to perform much better than the random workstealing algorithm, which often incurs a high cost when performing workstealing, since it is not guaranteed that any of the involved matrix tiles will be local. In Table 2a, we can see that the locality-aware workstealing algorithm is able to reduce the amount of time lost due to load imbalance compared to the other RDMA implementations. It should be noted that load imbalance numbers for the RDMA implementations cannot be directly compared to the bulk synchronous MPI implementation, since time lost to load imbalance is tied up in MPI's broadcast operations, which perform communication, but also enforce synchronization.

6.2 SpGEMM Experiments

To evaluate our RDMA-based SpGEMM algorithm, we perform the computation $C = AA$ for a number of different sparse matrices, representative of a number of different graph algorithms, such as Markov clustering [30], triangle counting [3], and transitive reduction [17]. We again benchmark RDMA-based stationary C and A algorithms, along with a work-stealing version, a SUMMA implementation in MPI, and PETSc's SpGEMM using CuSPARSE.

Strong scaling performance results are shown in Figure 5. Unlike SpMM, both the single- and multi-node environments are more likely to be compute bound. As a result, all the implementations have similar performance, because they are all performing the same computations. The only exception is PETSc, which is significantly slower, probably because it is not utilizing GPUDirect RDMA. On the large isolates matrix, our bulk synchronous CUDA-aware MPI implementation matches that of the RDMA implementation. On mouse gene, which is somewhat smaller and less load-balanced, we observe good scaling until very high concurrencies, but the RDMA-based implementations achieve higher overall performance as well as somewhat better scaling. On nlpkkt160, which has an uneven distribution of nonzeros, we observe poor performance in the multi-node experiments due to imbalance in both communication and computation. However, the RDMA implementations still perform better than the bulk synchronous implementations. Unlike SpMM, we do not meet our SpGEMM model's roofline bound on Summit, as shown in Figure 2. This is because the *local* cuSPARSE operations are not meeting the local roofline bound, evidenced by SpGEMM being compute bound in Table 2b. This likely indicates that there are opportunities for optimization in the local cuSPARSE SpGEMM operations.

For single-node experiments, we achieve better scaling on the highly load imbalanced matrices nlpkkt160 and ldoor,

Table 2. Component breakdown for selected matrices.

(a) Component breakdown for SpMM, N = 256 columns.

Env.	Matrix	Alg.	#GPUs	Comp.	Comm.	Acc.	Load Imb.
Summit	Amazon	S-C	24	0.02	1.3	-	0.4
			192	~0	0.5	-	0.2
		S-A	24	0.02	1.1	0.4	0.3
			192	~0	0.6	0.2	0.2
		S-C LW	24	0.01	1.2	0.03	0.3
			192	~0	0.6	0.01	0.1
		MPI	16	0.02	2.1	-	-
			256	0	1.0	-	-
DGX-2	Nm-7	S-C	4	0.44	0.01	0.02	0.14
		S-C	16	0.19	~0	0.09	0.10
		S-A	4	0.42	~0	0.07	0.12
		S-A	16	0.09	0.00	0.02	0.10
		MPI	16	0.64	0.51	0.01	0.15
			256	0.08	0.15	0.01	0.08

(b) Component breakdown for SpGEMM.

Env.	Matrix	Alg.	#GPUs	Comp.	Comm.	Acc.	Load Imb.
Summit	Mouse Gene	S-C	24	0.02	1.3	-	0.4
			192	~0	0.5	-	0.2
		S-A	24	0.02	1.1	0.4	0.3
			192	~0	0.6	0.2	0.2
		S-C LW	24	0.01	1.2	0.03	0.3
			192	~0	0.6	0.01	0.1
		MPI	16	0.02	2.1	-	-
			256	0	1.0	-	-
DGX-2	Mouse Gene	S-C	4	2.14	~0	0.02	0.72
			16	0.55	~0	0.01	0.21
		S-A	4	2.14	~0	0.38	0.49
			16	0.55	~0	0.11	0.13

likely because communication imbalance is eliminated due to by the significant increase in bandwidth, along with the fact that at lower concurrencies each GPU is left with a larger portion of the matrix, reducing the likelihood of load imbalance.

7 Related Work and Conclusions

The C++ PGAS library DASH, which uses RDMA through MPI's one-sided communication API, includes dense matrix and vector data structures as well as an implementation of stationary C dense matrix multiplication [15, 21]. However, DASH does not include any support for sparse matrices or GPUs.

The Combinatorial BLAS library (CombBLAS) [5] is an MPI-based library that provides distributed data structures for dense and sparse matrices, also implementing a number of distributed sparse matrix multiply algorithms. While focusing more on tensor computations, Cyclops Tensor Framework (CTF) [27] also supports several distributed sparse matrix multiplication algorithms.

Both CombBLAS and CTF implement also include so-called communication-avoiding algorithms for sparse matrices, which replicate some of the matrices to reduce communication costs. These ideas have also been applied to SpGEMM [2] and SpMM within the context of graph neural
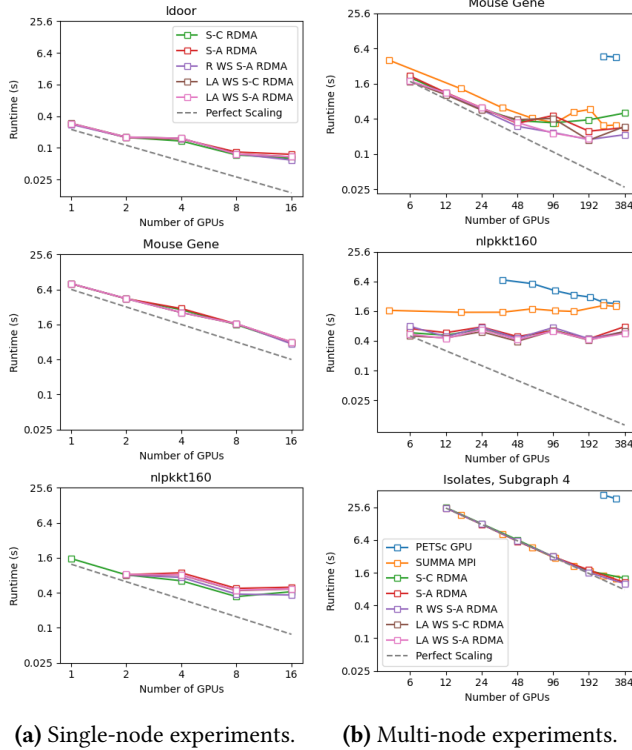
(a) Single-node experiments.   **(b)** Multi-node experiments.

Figure 5. SpGEMM strong scaling experiments.

network training [28]. Our work on asynchronous, RDMA-based implementations is orthogonal to these techniques, and could be combined.

This work explores a large space of possible communication and load balancing optimization for two sparse matrix operations, which are key to many simulation, analysis, and learning applications. We demonstrate that RDMA based communication can have a significant advantage over global collectives for some problems, and that dynamic work stealing can also contribute to improved scaling. Our extensive set of experiments on multiple matrices and two different machine configurations also highlight the importance of having high speed networking hardware that is balanced with the computational capabilities for this class of algorithms. Our inter-node roofline model provides a useful tool for evaluating performance and identifying opportunities for optimization.

Acknowledgments

References

[1] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. 2016. Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems. *Parallel Comput.* 59 (2016), 71–96.

[2] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. 2016. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing* 38, 6 (2016), C624–C651.

[3] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *IPDPSW*. IEEE, 804–811.

[4] Ariful Azad, Aydin Buluç, Xiaoye S Li, Xinliang Wang, and Johannes Langguth. 2020. A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs. *SIAM Journal on Scientific Computing* 42, 4 (2020), C143–C168.

[5] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John Gilbert, and Aydın Buluç. 2021. Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[6] John Bachan, Dan Bonachea, Paul H Hargrove, Steve Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Scott B Baden. 2017. The UPC++ PGAS library for Exascale Computing. In *Proceedings of the Second Annual PGAS Applications Workshop*. ACM, 7.

[7] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Dmitry Karpeyev, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Karl Rupp, Patrick Sanan, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2021. PETSc Web page. https://www.mcs.anl.gov/petsc. https://www.mcs.anl.gov/petsc

[8] Urban Borštnik, Joost VandeVondele, Valéry Weber, and Jürg Hutter. 2014. Sparse matrix multiplication: The distributed block-compressed sparse row library. *Parallel Comput.* 40, 5-6 (2014), 47–58.

[9] Benjamin Brock, Aydın Buluç, and Katherine Yelick. 2019. BCL: A cross-platform distributed data structures library. In *ICPP*.

[10] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, implementation, and applications. *The Intl. Journal of High Performance Comp. Applications* 25, 4 (2011), 496–509.

[11] Alhadi Bustamam, Kevin Burrage, and Nicholas A Hamilton. 2012. Fast parallel Markov clustering in bioinformatics using massively parallel computing on GPU with CUDA and ELLPACK-R sparse format. *IEEE/ACM TCBB* 9, 3 (2012), 679–692.

[12] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SDM*. SIAM, 442–446.

[13] Timothy A Davis. 2018. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In *HPEC*. IEEE, 1–6.

[14] S van Dongen. 2000. Graph clustering by flow simulation. *PhD thesis, University of Utrecht* (2000).

[15] Karl Fürlinger, Tobias Fuchs, and Roger Kowalewski. 2016. DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms. In *HPCC*. Sydney, Australia, 983–990. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140

[16] Zhixiang Gu, Jose Moreira, David Edelsohn, and Ariful Azad. 2020. Bandwidth Optimized Parallel Algorithms for Sparse Matrix-Matrix Multiplication using Propagation Blocking. In *SPAA*. 293–303.

[17] Giulia Guidi, Oguz Selvitopi, Marquita Ellis, Leonid Oliker, Katherine Yelick, and Aydin Buluc. 2021. Parallel String Graph Construction and Transitive Reduction for De Novo Genome Assembly. In *IPDPS*. IEEE.

[18] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *PPOPP*. 300–314.

[19] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *SC'20*.

[20] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC'20*.

[21] Felix Mössbauer, Roger Kowalewski, Tobias Fuchs, and Karl Fürlinger. 2018. A Portable Multidimensional Coarray for C++. In *PDP*. Cambridge, UK.

[22] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. 2015. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *ISC*. Springer, 48–57.

[23] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. 2013. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In *PPAM*. Springer, 559–570.

[24] 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.

[25] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2017. Order or shuffle: Empirically evaluating vertex order impact on parallel graph computations. In *IPDPSW*. IEEE, 588–597.

[26] Edgar Solomonik and Torsten Hoefler. 2015. Sparse tensor algebra as a parallel programming model. *arXiv preprint arXiv:1512.00066* (2015).

[27] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. 2013. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *IPDPS*. IEEE, 813–824.

[28] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *SC'20*. 1–17.

[29] Robert A Van De Geijn and Jerrell Watts. 1997. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience* 9, 4 (1997), 255–274.

[30] Stijn van Dongen. 2000. *Graph Clustering by Flow Simulation*. Ph. D. Dissertation. University of Utrecht.

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