Distributed-Memory Sparse Deep Neural Network Inference Using Global Arrays

Bruce Palmer, Sayan Ghosh, Andrés Márquez
Pacific Northwest National Laboratory, Richland, WA, USA firstname.lastname@pnnl.gov

Abstract—**Partitioned Global Address Space (PGAS) models exhibit tremendous promise in developing efficient and productive distributed-memory parallel applications. They have been used extensively in scientific computations due to conveniently offering a "shared-memory"-like programming model and interfaces that separate communication with synchronization. Traditionally, PGAS communication models have been applied to dense/contiguously distributed data, but most modern applications contain varied levels of sparsity. Existing PGAS models require certain adaptations to support distributed sparse computations, since associated computations often require matrix arithmetic, in addition to data movement.**

The Global Arrays toolkit from Pacific Northwest National Laboratory (PNNL) is one of the earliest PGAS models to combine one-sided data communication and distributed matrix operations, and is still used in the popular NWChem quantum chemistry suite. Recently, we have expanded the Global Arrays toolkit to support common sparse operations, like sparse matrix-dense matrix multiplies (SpMM) and sparse matrix-sparse matrix multiplication (SpGEMM). As it turns out, these operations are the backbone of sparse Deep Learning (DL); sparse deep neural networks have gained increasing attention recently in achieving speedups on inference with reduced memory footprints. Unlike scientific applications in High Performance Computing (HPC), modern (distributed-memory capable) DL toolkits often rely on non-standardized and closed-source vendor software optimizations, creating challenges in software-hardware co-design at scale.

Our goal is to support a variety of sparse matrix operations and helper functions in the newly created Sparse Global Arrays (SGA), such that it is possible to build portable and productive Machine Learning scenarios for algorithm/ software and hardware codesign purposes. We demonstrate the usefulness of SGA by building Sparse Deep Neural Network (SpDNN) challenge scenarios as a case study. The current SGA implementation is built on top of MPI and uses CPUs to maximize the portability across platforms.

Index Terms—**Sparse Deep Neural Network, Sparse Matrix Computations, Deep Learning, Machine Learning, Inference, Global Arrays, Distributed-Memory**

I. Introduction

PGAS (Partitioned Global Address Space) models such as SHMEM [6], UPC [4] and Global Arrays [26] predate the popular Message Passing Interface (MPI), and influenced the design of contemporary one-sided programming models and compiler extensions [3], such as Chapel [5], Coarray Fortran [27], X10 [7] and XScalableMP [21]. Although these PGAS interfaces provide rich local and global view data abstractions for efficient data movement, extreme-scale modern science and machine learning applications often need features such as distributed-memory sparse matrix operations, Sparse-Dense Matrix Multiplication (SpMM), Sparse Matrix-Vector Multiplication (SpMV), Sparse-Sparse Matrix Multiplication (SpGEMM) and their derivatives. Traditionally, PGAS approaches were prevalent in computational scenarios such as asynchronous update of fix-sized large distributed matrices with structured sparsity (e.g. the Density Functional Theory Hartree-Fock update in the NWChem computational chemistry suite [34]). As modern Deep Neural Network (DNN) workloads exhibit increasingly computationally demanding patterns, sparsifying neural networks might offer respite from higher memory consumption and data movement volumes [17]. Unlike sparsity in many existing science codes, which demonstrate structured data-blocks distributed uniformly (allowing locality-based optimizations), sparsity in neural networks can demonstrate variable patterns of structuredness [22] and accuracy/costs trade-offs [12], making it challenging for the underlying numerical libraries to devise standard solutions with sustainable performance. This leads to an even severe problem in developing distributed-memory solutions for ML workloads (i.e., data/pipeline parallelism), using traditional PGAS models which were primarily designed for scientific computation. On one hand, using existing (standardized and community adopted) PGAS models can lead to enhanced portability and augment software/hardware codesign efforts, but on the other hand, contemporary PGAS models must identify areas of improvement as we navigate novel DNN workloads. Standardization efforts such as GraphBLAS [19] to formally define the functions that could be used as building blocks for a wide range of distributed-memory sparse and combinatorial methods is a step in the right direction.

Originally, the Global Arrays library (GA) [25] provided "shared-memory"-like global view data abstractions, supporting dense array data structures, a set of one-sided communication primitives that facilitates arbitrary subarray access patterns using global arithmetic coordinates, dynamic load-balancing, and an interface to parallel dense linear algebra capability from ScaLAPACK [8]. The NWChem quantum chemistry package [33] is one of the widely used applications that relies heavily upon one-sided communication, having employed GA from the outset. Global Arrays has gone through a number of revisions, especially in its low-level communication substrate, from variants of ARMCI [13], [24] to the intermediate ComEx [10] runtime, which is, in turn, based on different MPI implementations. The current MPI-PR [15] (over MPI point-to-point communication) with *progress ranks/*

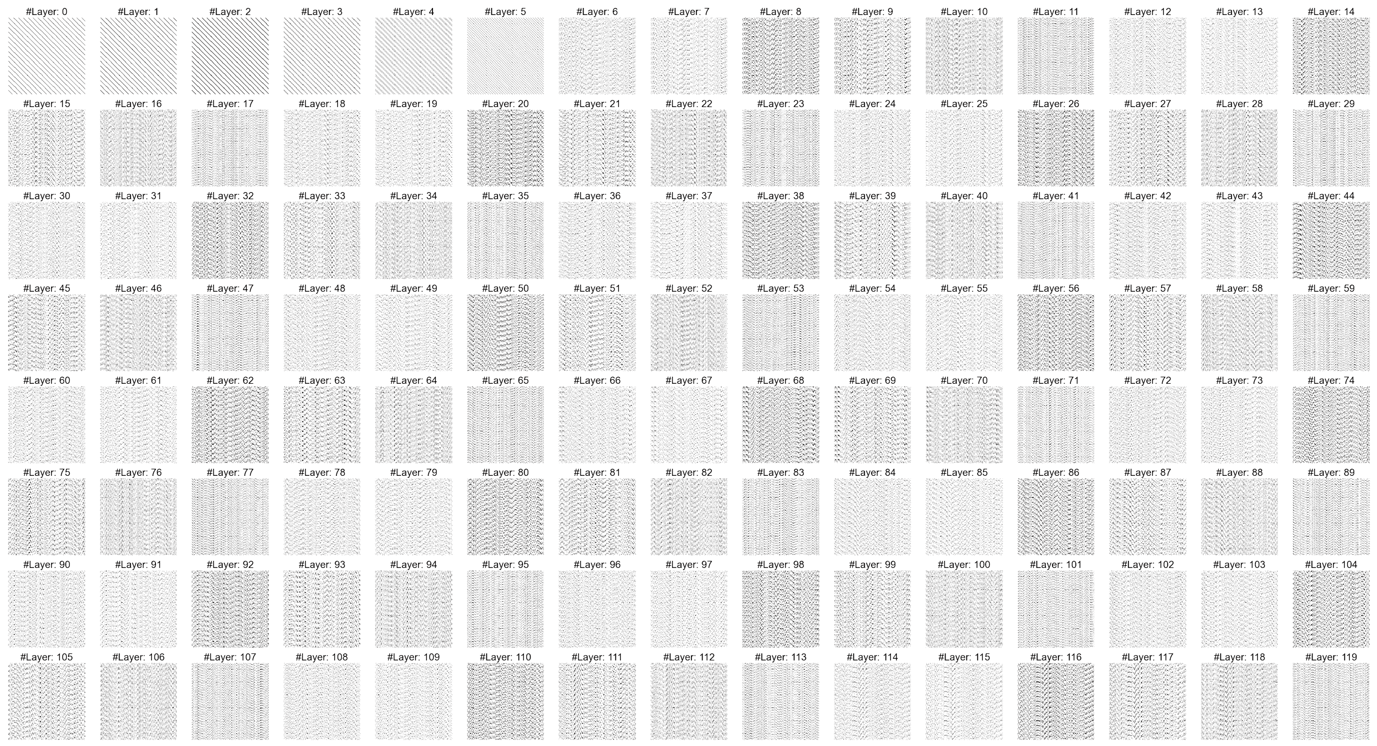


Fig. 1: Pattern of SpDNN weights for 1024 neurons on 120 layers (each 1024×1024 matrix has 32768 nonzeroes, 97% sparsity).

PEs on each node to evokes asynchronous communication. Process-based progress [31] can alleviate the drawbacks of thread-based or hardware interrupt-based communication progress, promoting scalability in MPI applications [30]. Most recently, the Global Arrays library has been overhauled to support sparse matrices (considering a two-level distribution, blocks of Compressed Sparse Row to hold the nonzero elements), and has native implementations of a number of sparse matrix operations. Recent efforts on MIT/IEEE/Amazon Sparse DNN Challenge has improved the state-of-the-art of SpDNN kernels [16], [32], [35] through various low-level optimizations, showing significant speedups on latest GPU accelerators. However, existing works on distributed-memory adaptations of SpDNN are relatively rare, a previous year's paper [23] alludes to the high synchronization costs of data and model parallelism. Instead of replicating the weights, we adopt *Tensor Parallelism*, which is a variant of Model Parallelism that distributes the weights and features across the PEs (each PE processes a different subset of features), invoking compulsory synchronizations at the end of a layer [29]. It is apparent that distributed-memory parallelism is necessary to cater to the largest models. Existing performant SpDNN implementations have either developed optimized low-level sparse-matrix operations [16], [32], or extensively relied on third-party vendor BLAS [14] libraries such as LAPACK [1] and NVIDIA's cuBLAS [28] to reach peak system efficiency (compiler-based solutions can also compete with vendor-optimized implementations [2], [9]). In contrast, we investigate the general effects of distributed-memory data movement and synchronization considering SpMM and SpGEMM based SpDNN variants. Our SGA based implementations are yet to be fully optimized for arbitrary hardware platforms, but reveals key design considerations and trade-offs that existing distributed-memory programming models must contend with while building the core functionality of machine learning applications at scale. Our study is geared towards facilitating codesign between sparse-matrix algorithms, programming models and runtime, and, hardware architecture (network interconnect topology and processors), in driving the efficiency of distributed-memory machine learning workloads.

II. SpDNN implementation using SGA

We followed the structure of the baseline implementation of the Sparse DNN Challenge [20], which performs the following sparse DNN computation on L layers: $Y_{l+1} = ReLu(W_l \times Y_l + b_l)$, where l is the current layer, Y is the feature matrix (M features of length N equal to the #neurons, $N \times M$), W is the matrix of activation weights ($N \times N$), b is the bias of size $N \times M$ and *ReLu* is the activation function with a maximum cutoff. We use the Compressed Sparse Row format (CSR) to store a block of nonzero elements. We perform the sparse layer computations SpDNN based on sparse-matrix-dense- matrix multiplication (SpMM) and also sparse-matrix-sparse-matrix multiplication (SpGEMM), considering the feature matrix Y as sparse or dense accordingly (applying bias activations separately), with distributed-memory support based on the recently SGA variant of GA.

A. Sparsity pattern

The sparsity pattern of neural networks is distinct from the structured sparsity observed in scientific computations, as evident from Fig. 1, which shows the nonzero patterns of the $N \times N$ weight matrices per layer, W_l. In contrast, the output feature matrices per layer l (i.e., $neurons \times features$, $Y_l\{N, M\}$), after application of bias activation demonstrates significant reductions in the sparsity across the layers, as shown in Table I. Therefore, although SpMM is the standard option

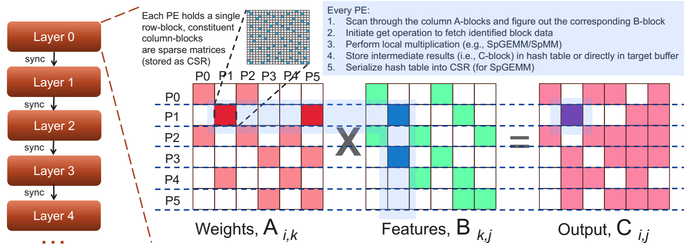
TABLE I: Sparsity patterns on output ($N \times M$) feature matrix Y after activation for the first ten layers.

N	M	*nnz*	sparsity (%)
1024	54687	17406000	68.92
1024	40498	8562176	79.35
1024	23042	5503360	76.68
1024	12010	4215712	65.72
1024	6537	3774752	43.61
1024	4087	2618048	37.44
1024	2774	2425600	14.61
1024	2128	2118688	2.77
1024	1939	1975216	0.52
1024	1874	1917600	0.07

for constructing SpDNN, SpGEMM could be applied instead, keeping the bias values and the layer under consideration. Distributed-memory SpGEMM is also harder to optimize, and incurs higher synchronization costs.

B. Data distribution

Data distribution impacts the granularity of computation, affecting the overall load balance and scalability. The high-level cartoon in Fig. 2 depicts the distribution of the row-blocks across the PEs for matrix computations in Sparse Global Arrays (SGA). A Sparse Global Array is comprised



Every PE:
1. Scan through the column A-blocks and figure out the corresponding B-block
2. Initiate get operation to fetch identified block data
3. Perform local multiplication (e.g., SpGEMM/SpMM)
4. Store intermediate results (i.e., C-block) in hash table or directly in target buffer
5. Serialize hash table into CSR (for SpGEMM)

Fig. 2: Simplified high-level depiction of distributed sparse matrix operations per layer in Sparse Global Arrays (SGA). Broken horizontal lines indicate the nonzero row-blocks owned by a particular PE.

of blocks of nonzero elements, represented via Compressed Sparse Row (CSR) format, spread across a 1-D process/PE grid, $1 \times npes$, where $npes$ is the number of PEs. Although multidimensional process grids are common for (partially) dense data, for sparse data, currently there is no consensus on whether multidimensional PE grids can lead to sustainable performance improvements for diverse sparse inputs and algorithms. Considering $M(rows) \times N(columns)$ sparse matrix, SGA arranges the sparse blocks such that each process owns $M/npes$ block of rows (a.k.a. row-block), and each row-block is further subdivided into $N/npes$ column-blocks, leading to a local block size of $M/npes \times N/npes$. Fig. 2 presents a simple scenario where the local block dimensions are same for the three SGAs; this is true for SpGEMM which guarantees that the the column dimension of A is partitioned exactly the same as the row dimension of B. This is sufficient to guarantee that the individual blocks of B that are needed by the blocks of A for local matrix multiplication in SpGEMM is located on a single process. However, for SpMM, since A is sparse and B is dense, the row/column partitioning into blocks might lead to dissimilar block dimensions. Consequently, fetching a remote block of B might involve more than one PE, since the underlying data might be distributed across several PEs. Each block for SpGEMM is a represented via CSR (for SpMM B-blocks are dense), and associated bounding indices, block pointers, strides, process IDs, etc. are held as metadata in a distributed $npes \times npes$ array, to optimize the memory usage. Each PE knows the range of the blocks pertaining to a specific SGA held by the other PEs, and must inquire parameters (such as physical pointer offsets across PEs corresponding to a particular "block") before initiating data communication. Specific *progress* ranks/PEs (initialized at execution time)

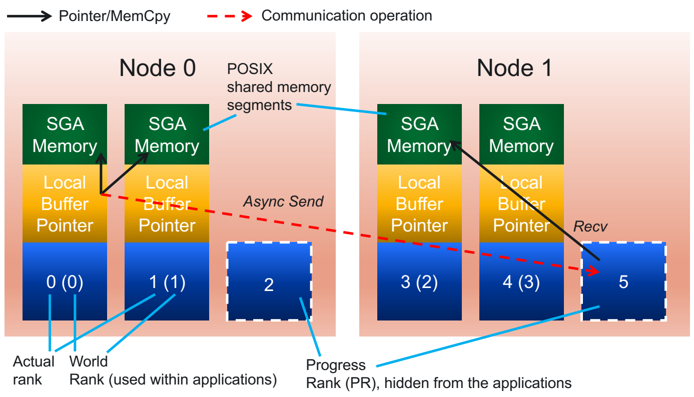


Fig. 3: Demonstrating an asynchronous "Put" operation invoked by process #0 in Node #0 to process #2 on Node #1 via Progress Rank (PR) #5. Each node has two user PEs and one hidden PR. Users must invoke `sync` operation for remote completion, SGA is implicitly locally complete (i.e., after `Put` returns, local buffers can be reused).

for each SMP node handle communication in SGA. Thus, progress ranks/PEs must be able to access the memory of another process on the node to facilitate direct access to the underlying pointers, to avoid unnecessary communication from one process's memory to another. As such, SGA uses POSIX Shared Memory Segments; every PE creates a shared-memory segment (i.e., a region of memory), allowing other PEs to attach to the memory segment such that it becomes a part of its userspace memory. This allows the progress PEs to manage communication on behalf of the other PEs (i.e., asynchronous background progress), leaving them free to spend the time on local computations. The number of shared memory segments increases significantly with the number of SGAs (i.e., number of shared-memory segments is equivalent to the #SGAs $\times$ processes-per-node, the #SGAs is proportional to the #layers). Fig. 3 demonstrates a typical one-sided communication scenario (i.e., Put) in SGA. Internally, the origin process asynchronously sends a *header* message to the target process's progress rank with the *payload* information, which then posts a subsequent receive to handle the incoming payload. For small data transfers, the payload can fit into the outgoing header, and after receiving the header, a target progress rank can perform memcpy to transfer the data directly to the specific process. Users must invoke Sync for remote communication completion, however, local buffers can be updated after a communication call returns.

C. Distributed pseudocode

Pseudocode 1 presents the high-level steps in distributed-memory sparse matrix multiplication in SGA, depicting the operations underlying SpMM and SpGEMM, invoked by every PE (refer to Fig. 2). Since the size of the destination sparse matrix cannot be known in advance for SpGEMM (without performing a "symbolic" multiplication to track the nonzeroes), there is an extra step of maintaining an extensible hash table to store the intermediate values (as seen in Line 14), before the final SGA can be constructed. Initially, each PE reads its row-block and fetches the A-block from the local memory (Line 6), and issues a *get* operation to retrieve the associated B-block from remote PE(s), as shown in Line 19. Prior to communicating the data, the *get* operation must retrieve the block parameters from the distributed metadata (as discussed in §II-B), as shown in Line 20 of Pseudocode 1. The local matrix computations are relatively similar for SpMM and SpGEMM, except SpGEMM allocates a temporary buffer (which is subsequently reused) for storing the intermediate values (Line 13), which must be inserted into a hash table, as seen in Line 14. The hash table is automatically expanded when it grows beyond a threshold, which can be expensive due to subsequent reallocations and memory copies. Additionally, towards the end, the hash table must be serialized and organized into the distributed sparse global array after global synchronization, as shown in Line 18. In contrast, for SpMM, the dimensions of the output (dense) global array is known in advance, so the results of the local matrix computations can be directly updated in the local view of the global array (in Line 10). The progress PEs runs continuously probing the network and returning data to the requesting PEs, as shown in Line 25. When the requesting process is local to the node, the progress PE uses direct memory copy instead of MPI.

Algorithm 1 Sparse-matrix computations in Sparse Global Arrays.
Input: Sparse Global Arrays (SGA): $A(M \times K)$ and $B(K \times N)$, distributed over *npes* PEs. The nonzero column block dimensions are calculated by dividing the original row/column dimensions by *npes*, for e.g., $M_b \leftarrow M/npes$, $N_b \leftarrow N/npes$ and $K_b \leftarrow K/npes$, *npes* excludes the #progress PEs involved in asynchronous progress;
Specific progress PEs handle communication progress, $progress(p)$ returns the progress PE corresponding to a regular PE, p.
Op specifies SpMM or SpGEMM.
SpGEMM requires an expandable hash table HT_c initially set to a size of 4096 entries for storing intermediate data.
Output: Sparse/Dense Global Array (result): $C(M \times N)$.

```
1:  for $i \in npes$ do    ▷Row block owned by PE
2:      for $j \in npes$ do    ▷Column blocks on PE
3:          for $k \in npes$ do
4:              Load block $a_{ik}$ from local memory
5:              Get block $b_{kj}$ from remote PE(s)
6:              for $ii \in M_b$ do    ▷Scan rows in block $a_{ik}$
7:                  for $kk \in K_b$ do    ▷Scan non-zeroes in row ii
8:                      for $jj \in N_b$ do    ▷Scan non-zeroes in row kk of $b_{kj}$
9:                          if Op is SpMM then
10:                             $C(global(ii,jj)) \leftarrow a_{ik}(ii,kk) * b_{kj}(kk,jj)$
                                ▷Directly update local portions of global array
11:                         else
12:                             Allocate intermediate block $c_{ij}(ii,jj)$
13:                             $c_{ij}(ii,jj) \leftarrow a_{ik}(ii,kk) * b_{kj}(kk,jj)$    ▷Local
14:                             $HT_c[ii,jj].insert(c_{ij}(ii,jj))$    ▷Hash table insert
15: Progress()    ▷Invoke communication progress
16: Sync()    ▷Synchronize across PEs
17: if Op is SpGEMM then
18:     $C \leftarrow HT_c.serialize()$    ▷Organize C into blocks for SpGEMM
19: function Get ($b_{kj}$):    ▷Get the
20:     $params \leftarrow get\_params(b_{kj})$    ▷Parameters describing $b_{kj}$
21:     $b_{kj}.allocate(params.dims)$    ▷Allocate memory for $b_{kj}$
22:     for $p \in (params.npes)$ holding data of $b_{kj}$ do
23:         Send($params.header, progress(p)$)    ▷To progress rank
24:         $b_{kj}$ ←Recv ($data, progress(p)$)    ▷post receive for data
25: function Progress():    ▷Progress PEs invokes this loop
26:     if $progress(my\_pe()) = my\_pe()$ then
27:         while true do
28:             $request \leftarrow$ Probe    ▷Probe incoming requests
29:             if $request$ is true then    ▷Message available to PE
30:                 $request.header \leftarrow$ Recv($request.params$)
31:                 $response.data \leftarrow fetch\_local\_data(request.header)$
32:                 Send($response.data$)    ▷Send back requested data
```

III. Evaluations and Analysis

a) Testbed: We have used two evaluation platforms to evaluate SGA at different scales. *PNNL Junction* cluster is comprised of AMD EPYC® "Milan" 7543 processors. Each compute node has two sockets with a 2.8 GHz 32-core AMD ® Milan processor per socket (256MB L3 cache), and 256GB DDR4 3200 MHz memory per node with 8 memory channels per processor. Each node has a Mellanox HDR-100 ConnectX-6 InfinBand HCA, providing about $1\mu s$ communication latency between nodes. We use OpenMPI 4.1.2 and GNU C++ compiler v10.3 to build the code, and pass "-O3" as compilation options. We have also used the *NERSC Perlmutter* supercomputer consisting of 3,072 CPU-only compute nodes. Each Perlmutter node uses 64-core 2.4GHz AMD EPYC 7763 CPUs with 256GB of DDR4 memory, 256MB L3 cache and 8 memory channels with HPE Slingshot 11 interconnect, providing up to 200Gbps (25GB/s) bandwidth [36]. We use Cray/MPICH 8.1.28, craype/2.7.30 and GNU C++ compiler v12.3 on Perlmutter.

b) SGA implementation: The DNN datasets includes weights for each layer and true labels. MNIST dataset [11] contains the initial feature matrix (Y_0). Currently, the implementation utilizes binary variants of the input (weight) matrices and features, to minimize I/O. Each PE reads portions of the inputs and assembles the local part of the distributed sparse array (like the GraphBLAS API [19], SGA has a NGA_Sprs_array_add_element64(<SGA_handle>, <ii>, <jj>, <value>) function to add an element corresponding to a coordinate). An SGA is created for every weight and bias matrices (like legacy Global Arrays, an SGA is uniquely identified by an opaque integer handle), equivalent to the #layers. Therefore, total number of SGAs created are equivalent to $\#layers + 2$, considering two input/output feature matrices (which are reused) in addition to per-layer weight. Consequently, the number of shared segments are also proportional to the #layers, and the administrator had to increase the maximum allowable shared memory segments on Junction for SpDNN with SGA to work for large #layers.

```
#include <ga.h>
int currfeat_s;
int nextfeat_s;
int* weights;
...
void kernel_gspmm(int l) {
#ifdef USE_SPMM
   nextfeat_s = NGA_Sprs_array_sprsdns_multiply(weights[
        l],currfeat_s);
#else
   nextfeat_s = NGA_Sprs_array_matmat_multiply(weights[l
        ],currfeat_s);
#endif
   void *tbias = static_cast<void*>(&bias);
#ifdef USE_SPMM
   Dense_ReLU(nextfeat_s, tbias);
#else
   NGA_Sprs_array_activate_ReLU64(nextfeat_s, tbias);
#endif
#ifdef USE_SPMM
    GA_Destroy(currfeat_s);
#else
    NGA_Sprs_array_destroy(currfeat_s);
#endif
    currfeat_s = nextfeat_s;
}

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);
    GA_Initialize();

    weights = new int[layer]; // SGA integer handles
    // process inputs and create SGAs
    read_weights_features(layer, weights, currfeat_s);

    // perform layer computations
    for(int i = 0; i < layer; ++i) {
      kernel_gspmm(i);
    }

    // cleanup and destroy SGAs before termination
    for (int l=0; l<layer; l++) {
      NGA_Sprs_array_destroy(weight_matrices[l]);
    }
#ifdef USE_SPMM
    GA_Destroy(currfeat_s);
#else
    NGA_Sprs_array_destroy(currfeat_s);
#endif
    delete [] weights;
    GA_Terminate();
    MPI_Finalize();
    return 0;
}
```

Listing III-0b lists the abridged SGA code for SpDNN including the necessary initializations, with macro USE_SPMM to distinguish SpMM-related code paths with SpGEMM. The output positions above the threshold after application of bias are maintained on a separate array, which is excluded from the code snippet in Listing III-0b for brevity. Since existing SGAs are identified using integer handles, the sparse matrix multiplication functions can conveniently refer to the feature and per-layer weight SGAs. Since SGA is built on top of MPI, it is interoperable with MPI, such that MPI and SGA functions can be mixed freely without performing any coarse-grain synchronization. SGA and the SpDNN kernels will be made available on https://github.com/pnnl/aiams-sparse.

A. Baseline performance of SpDNN using SpMM-based SGA

We discuss the baseline performance of SpDNN on Junction and Perlmutter platforms, on 16 (256 PEs) and 32 nodes (1024 PEs), respectively. Each node uses an extra "ghost"/progress PE, therefore we actually use 272 PEs on Junction and 1056 PEs on Perlmutter. For all the input configurations, the bias activation times are at most half a second per layer (observed for 65536 neurons), and often in the milliseconds range. Per-layer SGA creation overheads are also excluded because, $> 95\%$ of the total time is spent in the distributed SpMM operation. Since Tensor Parallelism also increases the communication and synchronization times, for larger neurons and layers, we estimate the total time from a sample of runs. Our implementation is entirely process/PE based, each PE computes the matrix operations without engaging threads or explicit vectorization. Table II lists the baseline

TABLE II: Baseline performance of SpMM-based SpDNN on PNNL Junction and NERSC Perlmutter platforms.

Neurons	Layers	PNNL Junction (n=16:ppn=17)	NERSC Perlmutter (n=32:ppn=33)
1024	120	221 (0.06 hrs)	1275 (0.35 hrs)
	480	883 (0.24 hrs)	5101 (1.41 hrs)
	1920	3532 (0.98 hrs)	20404 (5.66 hrs)
4096	120	414 (0.11 hrs)	1056 (0.29 hrs)
	480	1654 (0.45 hrs)	4223 (1.17 hrs)
	1920	6616 (1.83 hrs)	16890 (4.69 hrs)
16384	120	2216 (0.61 hrs)	6039 (1.67 hrs)
	480	8863 (2.46 hrs)	24154 (6.7 hrs)
	1920	35450 (9.84 hrs)	96617 (26.83 hrs)
65536	120	17981 (4.99 hrs)	14248 (3.95 hrs)
	480	71925 (19.97 hrs)	56992 (15.83 hrs)
	1920	287700 (79.91 hrs)	227967 (63.32 hrs)

performances (estimated for the largest layers of 4096, 16384 and 65356 neurons) across the layers based on distributed SpMM implementation of SGA. We observe the negative impact of increasing the #PEs by 4× on 1024/4096/16384 neurons between PNNL Junction (256 PEs) and NERSC Perlmutter (1024 PEs), resulting in about 3× performance degradation in latter. However, larger #PEs pays off for 65536 neurons, where we observe end-to-end performance improvements up to 20%. Observed results are still orders of magnitude worse than the best results reported in past Graph Challenges on a single-node (and reducing the #nodes will bring down our communication/synchronization costs, affecting the bottom-line), but with larger neurons (models) and deeper layers, distributed-memory implementations will become more viable and cost effective.

B. Layer-wise performance of SpDNN using SpMM-based SGA

Due to the sparsity pattern of the inputs, we observed non-trivial variation in the SpMM performances across the layers.

TABLE III: Major performance disparities in SpGEMM between small and large layers due to large number of shared memory segments, as demonstrated by comparing the timings for an arbitrary layer for the same #neurons.

Layers	**Total time (s)**		**Time for Get operations (s)**							**Misc.**		
	Init	**Mult**	**find**	**offset**	**copy**	**send**	**recv**	**wait**	**sync**	**Segments searched**	**#gets**	**#bytes (in gets)**
120	0.0045	7.31	0.0375	0.0002	0.0004	0.0025	0.0016	4.0428	3.1903	3298485	65478	73033624
1920	0.0049	140.82	1.1030	0.0002	0.0007	0.0040	0.0019	81.5682	58.0978	52265685	65478	73033624

Fig. 4 captures the variation across both the experimental platforms. On smaller #PEs (i.e., Junction), the lower layers are about 4–10× faster than Perlmutter; whereas, on higher #PEs at Perlmutter, the differences are apparent beyond ten layers. This indicates that for Tensor Parallelism, gradually increasing the resources with #neurons (as in weak scalability) can lead to a better outcome.

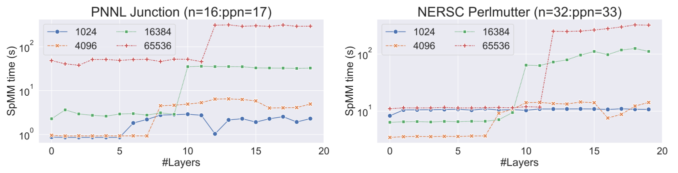


Fig. 4: Performance variations of SpMM across 20 layers on PNNL Junction and NERSC Perlmutter platforms.

C. SpGEMM vs. SpMM performance

We perform evaluations between SpMM and SpGEMM on four nodes (n=4:ppn=24) of PNNL Junction for 1024 neurons on 120 and 1920 layers. On 120 layers, we observe about 1.6× better performance of SpGEMM vs. SpMM; however, bias activations on the dense feature matrices are about 6× faster than the sparse counterpart in the SpGEMM variant of SpDNN, as shown in Fig. 5. However, the trend observed in Fig. 5 falls apart with larger #layers, specifically due to about 20× performance degradation in the SpGEMM version of SpDNN. We have profiled the SpGEMM versions on 120 and 1920 layers, and the per-layer timing break-down points to significant disparity in the numbers of shared memory segments (refer to §II-B, the number of shared memory segments is proportional to the #PEs and the SGAs, which depends on the #layers), as shown in Table III. By studying an arbitrary layer of the SpGEMM version of SpDNN between 120 and 1920 layers on same #PEs (as Fig. 5), we observe that on 1920 layers there are about 16× higher shared-memory segments, leading to significant overheads in synchronization for the progress PE (both `wait` and `sync` in Table III, which is typically $> 95\%$ of the total time), despite exactly similar volumes of data movement (indicated by the #gets and #bytes transferred). Upon doubling the number of progress ranks, we observed about 2× improvement in finding the shared segments, but the overall synchronization time remained unchanged.

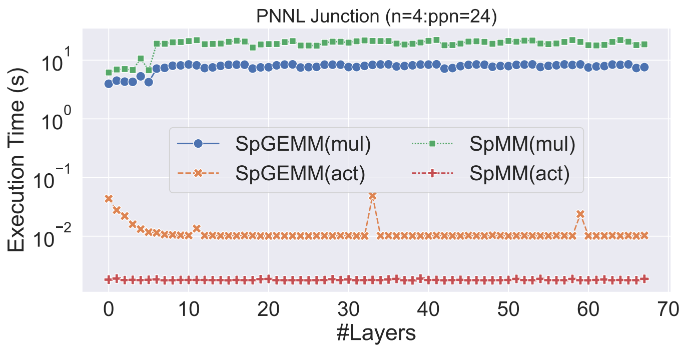


Fig. 5: SpMM vs. SpGEMM variants of SpDNN (multiplication and activation times) on four nodes of PNNL Junction.

Interestingly, the same progress rank based mechanism in (Sparse) Global Arrays is used to effectively scale quantum chemistry computations in the popular NWChem software package [33]. Therefore, this is an opportune moment to re-evaluate the established optimization techniques in contemporary HPC programming models and runtimes for modern AI/ML workloads.

IV. Concluding remarks

In this paper we introduce Sparse Global Arrays (SGA) as a convenient PGAS model for building applications based on distributed sparse matrix operations, and develop SpMM and SpGEMM versions of SpDNN sparse inference workload on distributed-memory platforms. Initial design of SGA considered distributed-memory structured grid based science applications, where it is more likely that the number of nonzero blocks per row is smaller than the number of available PEs (predicated on using optimal partitioners like Metis [18]). But, this is not the case for neural networks; there can be irregularly (sparse) nonzero blocks unevenly spread throughout. This requires revisiting the original data distribution of SGA, and perform reordering to arrange the nonzeroes in fixed-size groups by introducing another level of indices. Currently we considered fully distributed weights and features, but with the steady increase in the memory capacity (and access to node-local persistent memory), replicating the weights can reduce the communication overheads for relatively small-medium inputs.

Another possible optimization criteria is to use the MPI Remote Memory Access (RMA) or one-sided interface for communication instead of present two-sided progress PE based implementation (i.e., MPI-PR), since it can conveniently express the one-sided semantics (i.e., *get*) of SGA. Using progress ranks/PEs can lead to sustainable scalability, and contemporary MPI implementations natively support such options. Other relatively recent communication alternatives, such as MPI Neighborhood Collectives are also promising, however, it is yet to reach its full potential in terms of performance.

Acknowledgment

This work was supported by the US DOE Office of Science project "Advanced Memory to Support Artificial Intelligence for Science" at PNNL. PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

References

[1] E Anderson, Z Bai, C Bischof, J Demmel, J Dongarra, J DuCroz, A Greenbaum, S Hammarling, A McKenney, and D Sorensen. Lapack: A portable linear algebra library for high-performance computers. 1990.

[2] Aart JC Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in mlir. *arXiv preprint arXiv:2202.04305*, 2022.

[3] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[4] William W Carlson, Jesse M Draper, David E Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to upc and language specification. Technical report, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[5] Brad Chamberlain, Steve Deitz, Mary Beth Hribar, and Wayne Wong. Chapel. *Padua et al.[32]*, pages 249–256, 2005.

[6] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[8] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers — design issues and performance. In *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pages 95–106. Springer, 1996.

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

[10] Jeff Daily, Abhinav Vishnu, Bruce Palmer, Hubertus Van Dam, and Darren Kerbyson. On the suitability of mpi as a pgas runtime. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10. IEEE, 2014.

[11] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012.

[12] Tim Dettmers and Luke Zettlemoyer. Sparse networks from scratch: Faster training without losing performance. *arXiv preprint arXiv:1907.04840*, 2019.

[13] James Dinan, Pavan Balaji, Jeff R Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the global arrays pgas model using mpi one-sided communication. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 739–750. IEEE, 2012.

[14] Iain S Duff, Michael A Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.

[15] Nitin Gawande, Karol Kowalski, Bruce Palmer, Sriram Krishnamoorthy, Edoardo Apra, Joseph Manzano, Vinay Amatya, and Jonathan Crawford. Accelerating the global arrays comex runtime using multiple progress ranks. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*, pages 29–38. IEEE, 2019.

[16] Mert Hidayetoğlu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-mei Hwu. At-scale sparse deep neural network inference with efficient gpu implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2020.

[17] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

[18] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

[19] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *IEEE High Performance Extreme Computing (HPEC)*, 2016.

[20] Jeremy Kepner, Simon Alford, Vijay Gadepally, Michael Jones, Lauren Milechin, Ryan Robinett, and Sid Samsi. Sparse deep neural network graph challenge. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.

[21] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems. In *2010 39th International Conference on Parallel Processing Workshops*, pages 413–420. IEEE, 2010.

[22] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[23] Mohammad Hasanzadeh Mofrad, Rami Melhem, Yousuf Ahmad, and Mohammad Hammoud. Accelerating distributed inference of sparse deep neural networks via mitigating the straggler effect. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2020.

[24] Jarek Nieplocha and Jialin Ju. Armci: A portable aggregate remote memory copy interface. In *Citeseer*. Citeseer, 2000.

[25] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global arrays: A portable" shared-memory" programming model for distributed memory computers. In *Supercomputing'94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 340–349. IEEE, 1994.

[26] Jaroslaw Nieplocha, Robert J Harrison, and Richard J Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[27] Robert W Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[28] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

[29] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[30] Min Si, Antonio J Pena, Jeff Hammond, Pavan Balaji, and Yutaka Ishikawa. Scaling nwchem with efficient and portable asynchronous communication in mpi rma. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 811–816. IEEE, 2015.

[31] Min Si, Antonio J Pena, Jeff Hammond, Pavan Balaji, Masamichi Takagi, and Yutaka Ishikawa. Casper: An asynchronous progress model for mpi rma on many-core architectures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 665–676. IEEE, 2015.

[32] Yufei Sun, Long Zheng, Qinggang Wang, Xiangyu Ye, Yu Huang, Pengcheng Yao, Xiaofei Liao, and Hai Jin. Accelerating sparse deep neural network inference using gpu tensor cores. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2022.

[33] Marat Valiev, Eric J Bylaska, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Hubertus JJ Van Dam, Dunyou Wang, Jarek Nieplocha, Edoardo Apra, Theresa L Windus, et al. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477–1489, 2010.

[34] Hubertus JJ van Dam, Wibe A De Jong, E Bylaska, Niranjan Govind, Karol Kowalski, TP Straatsma, and Marat Valiev. Nwchem: scalable parallel computational chemistry. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 1(6):888–894, 2011.

[35] Jie Xin, Xianqi Ye, Long Zheng, Qinggang Wang, Yu Huang, Pengcheng Yao, Linchen Yu, Xiaofei Liao, and Hai Jin. Fast sparse deep neural network inference with flexible spmm optimization space exploration. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2021.

[36] Charlene Yang and Jack Deslippe. Accelerate science on perlmutter with nersc. *Bulletin of the American Physical Society*, 65, 2020.