Optimizing Nuclear Configuration Interaction Calculations on GPUs: A Comparative Performance Study of Programming Models

Abdullah Alperen*, Nan Ding†, Khaled Z. Ibrahim†, Pieter Maris‡, Leonid Oliker†, Chao Yang†, Hasan Metin Aktulga*
* Michigan State University
{alperena, hma}@msu.edu
† Lawrence Berkeley National Laboratory
{nanding, kzibrahim, LOliker, CYang}@lbl.gov
‡Iowa State University
{pmaris}@iastate.edu

***Abstract*—MFDn (Many-body Fermion Dynamics for nucleons) is a cutting-edge Configuration Interaction (CI) code designed to tackle nuclear quantum many-body problems. The core computational task in MFDn involves solving a large sparse eigenvalue problem using iterative methods, where the most costly computational step is the sparse matrix-vector multiplication (SpMV). Recently, an MPI/OpenACC based SpMV was developed to enable MFDn to run efficiently on NVIDIA GPUs. In this work, we explore various strategies to further enhance the performance of MFDn by exploiting architecture features of GPUs and making use of alternative programming models such as CUDA kernels as well as communication protocols such as the asynchronous point-to-point (P2P) message passing and NVIDIA Collective Communication Library (NCCL). We demonstrate the performance gain achieved by using these techniques. In particular, we show that, on problem sizes with up to 1.3 trillion nonzeros, we can obtain up to $2.0\times$ improvement in the overall solver time by switching from OpenACC to a hand-optimized CUDA implementation across 1540 GPUs. Additionally, when combined with asynchronous P2P message passing and NCCL, we observe performance boosts of $2.9\times$ (P2P) and $4.9\times$ (NCCL) compared to the baseline optimized version using the MPI/OpenACC programming model. Our study has uncovered a few limitations of the existing directive based programming models such as OpenACC and highlights the challenges of using CUDA-aware MPI for certain collective communications. While the primary focus of this work is on optimizing the performance of MFDn, the insights and solutions we provide are likely to be relevant to a broad range of applications.**

***Index Terms*—quantum many-body problem, Lanczos, distributed SpMV, MPI, NCCL, OpenACC, CUDA, roofline model**

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science under the Scientific Discovery through Advanced Computing (SciDAC) Program via Grants DE-SC0023175 and DE-SC0023495 from the Office of Nuclear Physics, and via funding for the SciDAC FASTMath Institute by the Office of Advanced Scientific Computing Research through Contract No. DE-AC02-05CH11231. We also acknowledge funding from the U.S. National Science Foundation, Office of Advanced Cyberinfrastructure under Grant 1845208. This work used computing resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the DOE Office of Science under Contract No. DE-AC02-05CH11231, using NERSC Awards ASCR-ERCAP m1027 for 2024 and NP-ERCAP m94 for 2024, as well as those provided by the High Performance Computing Center (HPCC) at Michigan State University.

I. Introduction

Solving quantum many-body problems is one of the most challenging tasks in computational science. MFDn (Many-body Fermion Dynamics for nucleons) is a cutting-edge Configuration Interaction (CI) code specifically designed to solve nuclear quantum many-body problems and enable scientists to study the structures of atomic nuclei [1]. The core computational task in MFDn involves solving a large sparse eigenvalue problem in which the many-nucleon Hamiltonian, expanded in Slater determinants of an harmonic oscillator single-particle basis, is partially diagonalized [2]. The dimension of this matrix, which we denote by n, depends on the number of nucleons and a truncation parameter $N_{\max}$ that determines the number of many-body basis functions (Slater determinants) used to represent the nuclear Hamiltonian. n can easily exceed a billion for a relatively large $N_{\max}$ even for a nucleus with a few protons and neutrons (collectively called nucleons) [1]. However, the Hamiltonian matrix is extremely sparse. In most cases, one needs to compute five to ten eigenvalues at the low end of the spectrum and the corresponding eigenvectors. As a result, iterative methods that make use of sparse matrix vector multiplication are suitable for solving this type of problem. MFDn contains an efficient implementation of the Lanczos algorithm [3]; other algorithms such as the Locally Optimal Block Preconditioned Conjugated Gradient (LOBPCG) algorithm have also been implemented in MFDn [4].

In these iterative algorithms, the multiplication of the sparse Hamiltonian matrix with one or several vectors (Sparse Matrix Vector and Sparse Matrix Matrix, SpMV and SpMM) is the most costly computational step in each iteration. A significant amount of effort has been made to improve the efficiency of such computation on distributed memory parallel computers with multi-core processors. In particular, a hybrid MPI/OpenMP implementation was developed to enable SpMV and SpMM operations to be performed efficiently on hundreds

of thousands CPU cores [5–9]. The implementation uses a special matrix partition and distribution scheme while organizing the computation to overlap with communication. More recently, an MPI/OpenACC implementation was developed to enable SpMV and SpMM operations to be performed efficiently on NVIDIA GPUs [10, 11]. This effort has led to an order of magnitude reduction in wall clock time for small to medium sized problems.

However, for large problems that must be solved on many computational nodes, each with multiple GPUs, the performance of the MPI/OpenACC implementation is less than satisfactory. The lackluster performance is partly due to high communication overhead relative to the number of floating point operations carried out on GPU devices. Furthermore, due to the irregular sparsity pattern of the MFDn Hamiltonian, OpenACC has some limitations in mapping concurrent operations in an SpMV and SpMM to GPU thread blocks and threads in an optimal fashion.

In this work, we demonstrate the limitations of this initial GPU-parallel version based on an MPI/OpenACC scheme. We then propose several ways to improve the performance of the Lanczos algorithm on large-scale multi-node multi-GPU configurations. For these experiments, we use Perlmutter nodes equipped with NVIDIA A100 GPUs at National Energy Research Scientific Computing (NERSC) [12]. We apply the proposed improvements to each SpMV, which is by far the most time-consuming portion of the Lanczos algorithm.

The contributions of this work can be listed as follows:

- We demonstrate a factor of two speedup in the Lanczos solver time by switching from an optimized OpenACC-based to a hand-tuned CUDA-based implementation of the SpMV kernel;
- We show that the MPI collectives – `MPI_Reduce` and `MPI_Reduce_scatter` – limit the scaling performance of Lanczos on multi-GPU experiments due to local reductions needed by these two calls being performed on the host side;
- We design and implement a P2P/CUDA scheme by employing non-blocking MPI point-to-point (P2P) communication instead of MPI collectives which can achieve a factor of three speedup over MPI/OpenACC (using collectives) on 69 Perlmutter GPU nodes at NERSC;
- We design and implement a NCCL/CUDA scheme by replacing MPI collectives with NVIDIA Collective Communications Library (NCCL) collectives. This scheme leverages the massive number of GPU threads for large messages and achieves a factor of five speedup over MPI/OpenACC on 48 Perlmutter GPU nodes at NERSC;
- We perform in-depth analysis for all implementations to help understand each implementation's performance benefits and limitations.

Our experiences may help guide other applications that involve expensive 2D distributed-memory SpMVs or SpMMs.

II. MFDn's Algorithm and Implementation

In MFDn there are two distinct workloads: one is the evaluation of matrix elements (both for the construction of the many-nucleon Hamiltonian at the start of a run, and for the evaluation of physical observables at the end of a run); and the other is the iterative solver for the partial diagonalization of the Hamiltonian. The evaluation of matrix elements is excellently load-balanced, does not involve any significant amount of communication overhead, and typically takes (significantly) less time than the iterative solver. For most production runs the iterative Lanczos solver takes between 40% and 80% of the total runtime; and the most time-consuming part of the Lanczos algorithm is the distributed SpMV. Furthermore, during the first 100 Lanczos iterations, the orthogonalization time (between each iteration) is negligible (less than 2%) compared to the SpMV time. An efficient implementation of the SpMV is therefore crucial in order to obtain a high-performance Lanczos solver. This is particularly true on distributed-memory where the communication overhead can easily become a bottleneck and hinder performance if not addressed properly, see Table II in Section III below. We therefore concentrate on the performance of the distributed SpMV, and in particular on the communication overhead during the distributed SpMV. For all our performance tests we use a fixed number of 100 Lanczos iterations (sufficient to converge the lowest two eigenvectors), so the orthogonalization time remains negligible compared to the SpMV time.

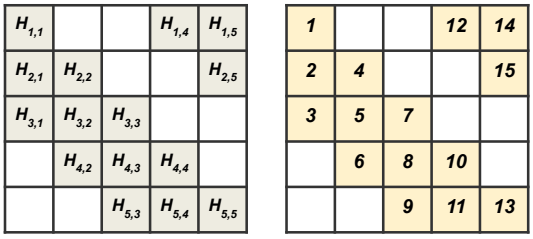
A. Data Distribution

The dimension of the sparse matrix H can be extremely large, and thus, it is often partitioned and distributed across multiple processes [2]. Additionally, in MFDn, only half of the symmetric matrix H is stored, utilizing a specialized data distribution scheme as described below [5, 6].

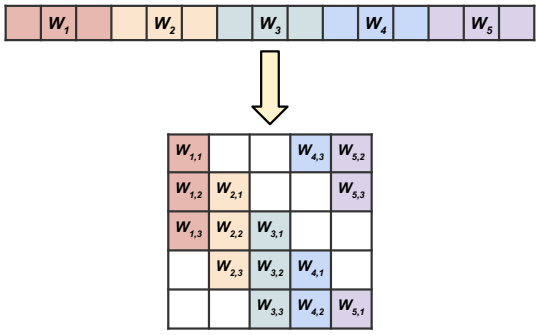
We first divide the rows and columns of H into $n_d \times n_d$ submatrices. Since the matrix is symmetric, we only need $n_d \times (n_d + 1)/2$ out of $n_d \times n_d$ submatrices to describe the complete matrix. These submatrices are mapped to a two-dimensional process grid. The processes are then organized into (sub)communication groups based on the row and column indices of their respective submatrices. Well-balanced row and column (sub)communication groups are key to achieving a meaningful performance for the distributed SpMV algorithm.

Figure 1a shows an example of this partition and selection process. The symmetric sparse matrix H is partitioned into 5×5 matrices. We also highlight the selected submatrices, each of which is assigned to its own process. Notice that each row and column holds three submatrices. This means there will be five row and five column communicators, each containing three MPI processes.

We partition the input vector W and the output vector U among $n_d \times (n_d + 1)/2$ processes in two stages. First, we partition W and U into n_d sub-vectors to distribute among column groups. Then, we further partition each sub-vector into $(n_d + 1)/2$ segments to distribute among the processes within a column group. Thus, each process is assigned to $1/(n_d \times$



(a) Symmetric sparse matrix H



(b) Input vector W

Fig. 1: (a) The partition of a symmetric sparse matrix H into 5 × 5 submatrices (left) and the distribution of H and W among 15 processes where we use a column-major order. (b) The partition of an input vector W into 5 sub-vectors on each column where we partition each sub-vector further into 3 segments.

$(n_d + 1)/2)$ of W and U. We show the partition of W in Figure 1b for the same example given in Figure 1a.

Note that MFDn employs a data distribution scheme designed to achieve load balance by ensuring that both the sizes of the distributed matrix blocks and the number of nonzero matrix elements in each block are approximately equal. However, these quantities are not perfectly identical across all MPI ranks. This may lead to some load imbalance in practice.

B. Distributed-memory SpMV

A specialized SpMV multiplication procedure has been developed to accommodate this unique data distribution scheme when multiplying H with W. Here, we refer to the jth block of sub-vectors of W as W_j. In MFDn, the distributed-memory parallel multiplication of H and W is performed as follows:

- **Step 1/4:** We first gather the distributed segments of the sub-vector W_j onto each process within the jth column communicator by using an `MPI_Allgatherv` call.
- **Step 2/4:** The ith diagonal process then broadcasts the gathered sub-vector W_j, where $i = j$ for the diagonal processes, across the ith row communicator. This is done in preparation for the distributed SpMV-transpose computations of the next step. We overlap this `MPI_Bcast` call with the local SpMV, which is $U_i = H_{i,j}W_j$, by using the gathered input sub-vector from the previous step.
- **Step 3/4:** We now reduce the partial results of the output sub-vectors U_i along each row communicator onto the ith diagonal process. We overlap this `MPI_Reduce` call with the local SpMV-transpose, which is $U_j = H_{i,j}^T W_i$, by using the broadcast input sub-vector from the previous step.
- **Step 4/4:** After the local SpMV-transpose, on the diagonal processes, we add the reduced output sub-vector U_i to the local output sub-vector U_j. Finally, we reduce and scatter the sub-vectors U_j into $(n_d + 1)/2$ segments within the jth column communicator by using `MPI_Reduce_scatter`.

Each process starts with a segment of a sub-vector of W and ends up with the corresponding segment of a sub-vector of U as intended. We show all four steps in Figure 2 to help visualize the algorithm.

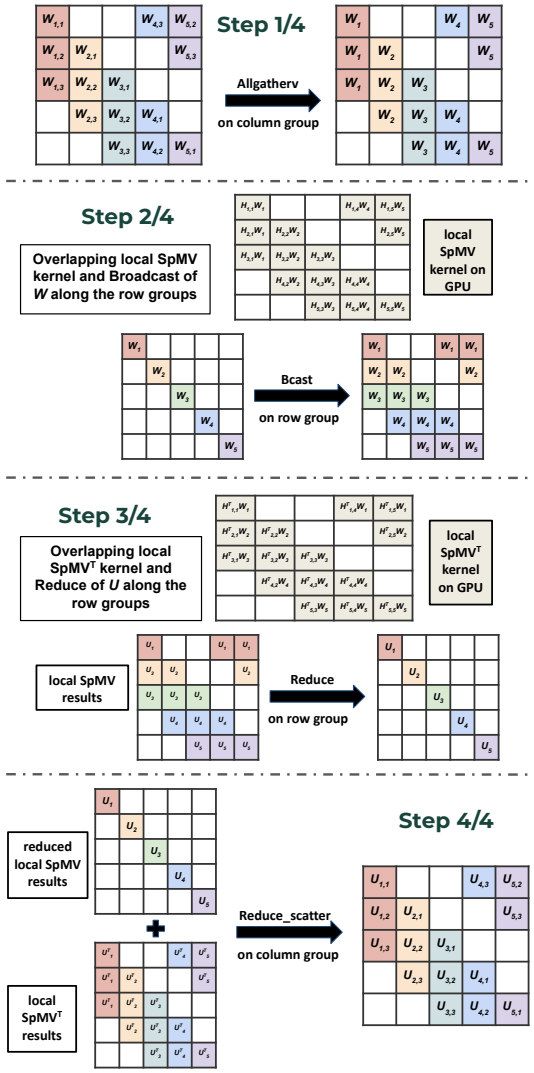


Fig. 2: Four steps of the SpMV algorithm employed in MFDn.

C. Task-to-Process Mapping

At large scales, communication overhead can limit the scalability of the Lanczos Algorithm. Topology-aware mapping of computational tasks to physical processors is one of these methods to reduce the communication overhead and thus improve the efficiency of a Lanczos implementation.

When mapping the submatrices of H and the segments of W and U to the processes, we use a column-major mapping scheme developed in [5]. Figure 1a shows this column-major mapping for the same example where $n_d = 5$. This way, we can partially (or completely) hide the cost of `MPI_Bcast` and `MPI_Reduce` by overlapping them with local SpMV and SpMV-transpose. Both MPI collectives are performed along row communicators. However, there is no compute task that can be overlapped with `MPI_Allgatherv` or `MPI_Reduce_scatter`. Therefore, performing these two MPI collectives on column communicators as efficiently as possible would improve the overall performanc [5]. Additionally, collective operations such as `MPI_Reduce` can introduce load imbalance. Specifically, only rank 0 in each sub-communication domain performs the local reduction computation, while other ranks remain idle, waiting for rank 0 to complete the operation.

The MPI library itself considers the network topology when assigning ranks to processes. For example, processes within a single node will be given consecutive ranks by default. Therefore, ordering processes in column-major order will ensure that the processes in a column group and their corresponding tasks are assigned to the physically nearby processors (or GPUs).

Although topology-aware mapping is shown to be an NP-complete problem [13], this heuristic suggested by [5] is proven to be quite efficient for the performance of the given SpMV algorithm. As an example, executing 28 MPI ranks on 7 nodes, with 4 ranks per node, is highly efficient because the MPI collectives on column communicators are all performed within a node, while the MPI collectives on row communicators overlap with local computations. Although the merit of this heuristic for the MPI/OpenMP implementation of MFDn was demonstrated on a distributed CPU cluster [5], the same heuristic is used for the GPU-parallel MPI/OpenACC scheme [11] as well, following the same underlying principles.

D. Current MPI/OpenACC implementation

MFDn uses an MPI/OpenACC scheme for the distributed SpMV algorithm outlined above. Specifically, it uses OpenACC as the GPU programming model for the local SpMV and SpMV-transpose, in combination with MPI collectives for the Allgatherv, Bcast, Reduce and Reduce_scatter operations.

Each local Hamiltonian submatrix $H_{i,j}$ referred to in section II-B is a block sparse matrix, with each block being sparse as well. The local SpMV procedure used in MFDn loops over nonzero blocks to perform a sequence of SpMVs on each sparse block, and accumulates the result into a local vector. In addition to a local SpMV, it also performs a local SpMV-transpose with the same data structure, because only half of the symmetric matrix is stored. The OpenACC implementation of such an SpMV procedure for GPUs uses `parallel loop` and `loop vector` directives to map sparse matrix blocks to thread blocks and non-zero matrix elements within each block to multiple threads to initiate concurrent executation of the local SpMV [11]. An `atomic update` directive is used to avoid write conflicts. A snipped of the OpenACC implementation is shown in Figure 3.

```
!$acc parallel loop collapse(2) default(present)       &
!$acc vector_length(CACHESIZE) private(c_ar,r_ar, x_ ar)
do i = 1, nrowblks
   do j = 1, ncolblks
      if (CSBnnz(i,j) > 0) then
         cbase = colCSBoffset(j) - 1
         rbase = rowCSBoffset(i) - 1
         do k = 1, CSBnnz(i,j), CACHESIZE
            kmax = min(CACHESIZE, CSBnnz(i,j)-k+1)
            koffset = CSBnnzoffset(i,j)+k-1
            !$acc loop vector
            do k = 1, kvmax
               c_ar(k) = nrhs*(cbase + Hcloc(koffset+k))
               r_ar(k) = nrhs*(rbase + Hrloc(koffset+k))
               x_ar(k) = Hval(koffset+k)
            enddo
            !$acc loop vector collapse(2)
            do k = 1, kvmax
               do ii = 1, nrhs
               c = c_ar(k) + ii
               r = r_ar(k) + ii
               x = x_ar(k)
               !$acc atomic update
               Hamp(r) = Hamp(r) + x * amp(c)
               !$acc end atomic
            end do
         end do
      end if
   end do
end do
!$acc end parallel loop
```

Fig. 3: The OpenACC implementation of the local SpMM. The sparse Hamiltonian matrix is stored in the arrays `Hcloc`, `Hrloc`, and `Hval` using a Compressed Sparse Block (CSB) format. The input and output vectors are stored in the arrays `amp` and `Hamp` respectively, and `nrhs` is the numbers of vectors (for Lanczos, `nrhs=1`) .

III. Performance Optimization Methodologies

We discuss the three proposed optimization schemes listed in Table I in this section. The three proposed optimization schemes are MPI/CUDA, P2P/CUDA and NCCL/CUDA. MPI/CUDA uses MPI collectives as is, but replaces the OpenACC kernels for local SpMV and SpMV-transpose with the corresponding CUDA kernels. P2P/CUDA is designed and implemented on top of MPI/CUDA. It replaces `MPI_Reduce` and `MPI_Reduce_scatter` collectives with MPI P2P calls and local GPU reduction kernels. NCCL/CUDA is also designed and implemented on top of MPI/CUDA, and it replaces all MPI collectives with NCCL collectives.

A. Improving Local SpMV with CUDA (MPI/CUDA)

OpenACC relies on the compiler to launch kernels with multiple thread blocks to execute nested loops shown in Figure 3. The number of thread blocks and the number of threads within each block is automatically determined by the

TABLE I: Summary of the default MPI/OpenACC as well as the proposed P2P/CUDA and NCCL/CUDA schemes.

	MPI/OpenACC (Baseline)	MPI/CUDA	P2P/CUDA	NCCL/CUDA
Local SpMV	OpenACC	CUDA	CUDA	CUDA
Local SpMV^T	OpenACC	CUDA	CUDA	CUDA
Allgatherv	MPI Coll	MPI Coll	MPI Coll	NCCL Coll
Bcast (overlapped)	MPI Coll	MPI Coll	MPI Coll	NCCL Coll
Reduce (overlapped)	MPI Coll	MPI Coll	MPI P2P	NCCL Coll
Reduce_scatter	MPI Coll	MPI Coll	MPI P2P	NCCL Coll

compiler. In our CUDA implementation, shown in Figure 4, we leverage shared memory as software cache to both the indexes and the values of block of data to hold active blocks. Accessing architectural features, such as shared memory, may be used by high-level offloading programming models only under certain constraints, for instance for special explicit compile-time array sizes [11].

Moreover, CUDA implementation enables more flexible mapping of computation to the three levels of architectural parallelism (warp-level, intra thread-block, and inter thread-blocks) and control of scheduling the thread blocks. The performance of atomic operations on GPU accelerators relies heavily on reducing the level of conflicts [14], which can be done more effectively using a low-level implementation.

In general, a low-level programming model like CUDA enables more precise control of GPU resource utilization for kernel launch with an optimal thread block size, thread configurations, and cache size. The performance gap between OpenACC and CUDA can be influenced by the code generation process of the OpenACC variant and the mechanism used to guarantee the atomicity of updates.

B. Improving Communication with P2P (P2P/CUDA)

In the current GPU implementation of MFDn, CUDA-aware MPI implicitly handles any necessary data movement between devices and hosts. Table II shows the time spent on each collective communication operation over 100 Lanczos iterations for two of our test cases (see Sect. IV for details). For an accurate measurement, we placed an `MPI_Barrier` before these collectives. One can immediately observe that communication time dominates the total solver time as the number of GPUs increases. Such observation is consistent with the that made in [11].

TABLE II: Collective communication time distribution over 100 Lanczos iterations, percentage of communication time to baseline solver time, and predicted communication speedup using P2P.

	Small			Medium		
# of GPUs	45	91	190	91	190	378
Allgatherv	0.51	0.36	0.23	1.42	0.76	0.56
Bcast	1.13	1.07	0.87	3.64	3.07	2.21
Reduce	5.56	3.85	2.84	13.4	9.61	7.01
Reduce_scatter	3.89	3.36	3.36	8.59	6.94	6.91
Tot	11.1	8.64	7.30	27.1	20.4	16.7
% of Solver Time	42.5	55.7	61.8	60.8	67.1	77.6
Predicted Speedup	2.35	1.79	1.61	1.64	1.49	1.28
Typical Message Size (MB)	132	95	55	329	230	164

```
__global__ void spmm_kernel(. . .)
{
   i = blockIdx.x;
   j = blockIdx.y;

   r_base = rowCSBoffset[i] - 1;
   c_base = colCSBoffset[j] - 1;

   start = CSBnnzoffset[i*ncolblks + j];
   end = start + CSBnnz[i*ncolblks + j];

   rhs_assigned = threadIdx.x & (nrhs - 1);
   cache_begin = threadIdx.x >> lognrhs;
   cache_jump = blockDim.x >> lognrhs;

   __shared__ int32_t Hrloc_local[CACHESIZE];
   __shared__ int32_t Hcloc_local[CACHESIZE];
   __shared__ float Hval_local[CACHESIZE];

   for(k = start; k < end; k += CACHESIZE)
   {
      len = min(CACHESIZE, end - k);
      last = k + len;
      for(l = k + threadIdx.x; l < last; l += blockDim.x) {
         Hrloc_local[l-k] = (r_base+Hrloc[l])<<lognrhs;
         Hcloc_local[l-k] = (c_base+Hcloc[l])<<lognrhs;
         Hval_local[l-k] = Hval[l];
      }
      __syncthreads();
      for(l = cache_begin; l < len; l += cache_jump) {
         r = Hrloc_local[l] + rhs_assigned;
         c = Hcloc_local[l] + rhs_assigned;
         xcoef = Hval_local[l];
         atomicAdd(Hamp+r, xcoef*amp[c]);
         atomicAdd(HampT+c, xcoef*ampT[r]);
      }
      __syncthreads();
   }
}
```

Fig. 4: The CUDA implementation of the local SpMM. Compared with the OpenACC implementation, the cache is implemented using software managed storage on the GPU shared memory. Moreover, mapping of CSB blocks to thread blocks impacts the atomic conflicts and the observed performance.

Table II suggests that `Reduce` and `Reduce_scatter` collectives are more expensive than `Allgatherv` and `Bcast`. On the Small problem with 190 GPUs, for instance, we spend 6.2s total on `Reduce` and `Reduce_scatter` while `Allgatherv` and `Bcast` time adds up to 1.1 seconds. Therefore, one needs to address the `MPI_Reduce` and `MPI_Reduce_scatter` calls in MPI/CUDA to achieve further improvements. The predicted speedups are obtained using the Message Roofline Model [15] which estimates time spent in a `MPI_Reduce` call *vs.* time spent in a reduction process implemented by using MPI P2P communication calls to collect data and performing summation on a GPU. Message Roofline Model benchmarks the achieved bandwidth according to the correlated number of messages per synchronization (square root of the number of GPUs) and typical message sizes. Intuitively, one would expect the predicted speedups to increase if communication time as a percentage of solver time increases. Conversely, our predicted speedups decrease as the number of GPUs increases. This combined effect of message size and number of messages results in P2P based reduction exhibiting non-constant speedups over MPI collectives. This

is why we see a smaller predicted speedup when using more GPUs.

Figure 5 shows a snapshot of the Nsight Systems profiling of the baseline GPU implementation. The bottom part shows the blocking nature of `MPI_Reduce` and `MPI_Reduce_scatter` calls on the CPU side. The top part, which displays the activity on the GPU hardware, highlights the copy operations between host and device: device-to-host data movements (*DtoH memcpy*) occur before the `MPI_Reduce` and `MPI_Reduce_scatter` calls, followed by one or multiple host-to-device data movements (*HtoD memcpy*) after all participating processes complete the reduction on CPUs. These observations make it clear that summation in the CUDA-aware `MPI_Reduce` takes place on the host side, which is considerably slower than GPU operations for large reductions.

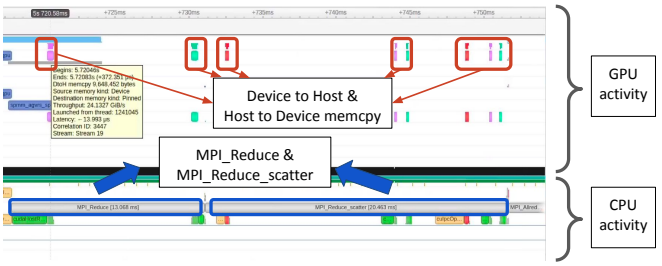


Fig. 5: NSight Systems profiling showing device to host and host to device copy operations performed during Reduce and Reduce_scatter calls with MPI.

Point-to-point (P2P) communication is a straightforward method to address the above problems. One only needs to replace `MPI_Reduce` and `MPI_Reduce_scatter` calls with non-blocking point-to-point (P2P) sends and blocking receives, and implement the summation performed in the reduction on GPUs. Figure 6 demonstrates a P2P implementation of `MPI_Reduce`. We use `MPI_Isend` and `MPI_Recv` with `MPI_ANY_SOURCE` calls between two processes to pass local data from multiple senders to a single target. Note that the single target approach does not create a significant bottleneck, as both the reduce and reduce-scatter operations are carried out over row and column groups whose cardinality scale with the square root of the number of GPUs used. Furthermore, the use of `MPI_ANY_SOURCE` in `MPI_Recv` allows the messages to arrive in any order, improving the communication efficiency. Finally, the `MPI_Recv` is issued right before the `saxpy_kernel` to enable communication and computation overlap.

C. Improving Communication with NCCL (NCCL/CUDA)

The MPI standard and its CUDA-aware variant allow good program and performance portability across architectures, but they lack support for leveraging massive parallelism for large messages. As listed in Table II, a typical message size in MFDn can be tens to hundreds of megabytes. According to the Message Roofline model [15], MFDn is in the network

```
!P2P implementation for MPI_Reduce
if (idiag .gt. 0) then   !root process
    !nsegments = number of participated processes
    do i = 1, nsegments - 1
        call MPI_Recv(Hamp_buf, nvecs * nrows, MPI_REAL4, &
                      MPI_ANY_SOURCE, tag, row_comm,    &
                      status, ierr)

        !GPU summation
        tBlock = dim3(1024, 1, 1)
        grid = dim3(108, 1, 1)
        call saxpy_kernel<<<grid, tBlock, 0, stream12>>>  &
                      (nvecs * nrows, Hamp_buf, Hamp)
        istat = cudaStreamSynchronize(stream12)
    end do
else
    call MPI_Isend(Hamp, nvecs * nrows, MPI_REAL4, 0,     &
                   tag_reduce, row_comm, request, ierr)
end if

! GPU summation
attributes(global) subroutine saxpy_kernel(n, x, y)
      implicit none
      real(kind=4), device :: x(*), y(*)
      integer, value :: n
      integer :: i
      !stride = blockDim%x * gridDim%x
      !       = 108 * 1024 = 110592
      do i = blockDim%x * (blockIdx%x - 1) + threadIdx%x, n
           , 110592
        y(i) = y(i) + x(i)
      end do
  end subroutine saxpy_kernel
```

Fig. 6: Point-to-point implementation of `MPI_Reduce`. The *saxpy_kernel* is the user-implemented summation for reductions on GPUs.

bandwidth-bound region. Thus, leveraging the parallelism and advanced communication links (i.e., NVLink) on GPUs for such large communication operations in MFDn could improve the communication performance. The NVIDIA Collective Communications Library (NCCL) is designed to accelerate inter-GPU communication by leveraging massive parallelism on GPUs as well as the NVLink hardware (when available). NCCL supports all four collectives needed by the distributed SpMV in MFDn with some adjustments for `Allgatherv` and `Reduce_scatter`. NCCL's `ncclAllGather` and `ncclReduceScatter` calls expect equal-sized input vectors from all participating processes. Therefore, we pad shorter messages (vectors) with zeros to make them all equal-sized messages. As the vector size varies by less than 1% between different communication groups, this padding adjustment introduces a negligible overhead.

However, direct use of NCCL has limited impact. To achieve high performance, one must carefully coordinate the order of NCCL and CUDA calls to reduce synchronization overheads, and fine-tune thread parallelism in NCCL for communication efficiency. We discuss these in detail in the Results section.

In summary, we use CUDA to leverage the fine-grained control of architectural parallelism and the high-speed shared memory to accelerate computations, which is infeasible in OpenACC. We then use MPI P2P and NCCL to improve communication efficiency. MPI P2P has relatively better performance compared to MPI collectives and good program

portability compared to NCCL across architectures. However, it lacks parallelism support for large reductions, and it requires developers to implement their own summations on GPUs. Conversely, NCCL is more advanced in leveraging GPU parallelism for large reductions, but developers must carefully maintain the orders and synchronizations between NCCL and GPU computation kernels.

IV. Experimental Setup

We report the performance of the Lanczos algorithm implemented in MFDn on the GPU partition of the Perlmutter system at sNERSC [12]. Each Perlmutter GPU node consists of a single AMD EPYC 7763 processor with 64 cores per processor and 4 NVIDIA A100 GPUs, as shown in Fig. 7. GPUs on each node are fully connected via NVLINK 3.0, which provides 100 GB/s/direction for each GPU pair. The CPU and GPUs are connected via PCIe 4.0, which provides a peak bandwidth of 25 GB/s/direction per CPU-GPU pair for data transfers. Each node also has four PCIe 4.0 NICs, which provide 4×25 GB/s/direction at peak. Perlmutter uses a three-hop dragonfly topology with 24 groups, and it has a 28% bisection bandwidth tapering (4×7 GB/s per GPU node) [16]. That is when running on more than 64 GPU nodes, requested GPU nodes are located on two different network groups, and the bisection bandwidth limits the peak network performance.

Note that the Perlmutter GPU partition has 256 large memory GPU nodes, and each GPU on that node is equipped with 80 GB of High Bandwidth Memory (HBM). Throughout the paper, experiments that request less than 256 GPU nodes run on the large memory GPU nodes, while other cases that request more than 256 nodes run on regular GPU nodes, which have 40 GB of HBM per GPU. These two types of GPU nodes share the same architecture, and the only difference is the HBM size (40 GB vs. 80 GB) and its corresponding peak HBM bandwidth per GPU (1.5TB/s vs. 2 TB/s). For all experiments reported in the paper, the software infrastructure included NVIDIA HPC SDK 23.9, CUDA 12.2, Cray-MPICH (8.1.28), and NCCL 2.18.3.

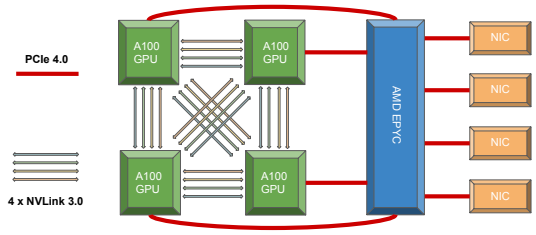


Fig. 7: Node architecture of Perlmutter GPU partition.

We use four test problems corresponding to the Hamiltonian matrices of different nuclei represented in different configuration interaction spaces. The dimensions of these matrices as well as the number of nonzero matrix elements (nnz) in half of each of these symmetric matrices are listed in Table III. The dimensions of the test problems range from 1.6×10^8 to 1.8×10^9 whereas their nnz vary between 1.2×10^{11} and 1.8×10^{12}.

TABLE III: Dimensions of sparse matrices used in performance tests and the number of nonzero matrix elements in each matrix.

Test case	**Small**	**Medium**	**Large**	**XLarge**
Matrix	^{10}B	^{12}C	^{9}Li	^{12}B
$N_{\max}$	8	8	11	9
Matrix dimension ($\times 10^9$)	0.16	0.57	0.97	1.78
# of nonzeros (nnz) ($\times 10^{12}$)	0.12	0.47	1.30	1.81
minimum # of GPUs	28	91	276	378
nnz / min. # of GPUs ($\times 10^9$)	4.4	5.2	4.7	4.8

The smallest problem, labeled as Small, needs at least 28 NVIDIA A100 GPUs with 80GB of HBM. As the problem size increases, more GPUs are needed for Lanczos; Medium, Large and XLarge test cases have to be executed on at least 91, 276 and 378 A100 80GB GPUs, respectively. We report the total time taken by 100 iterations of the Lanczos solver in MFDn, including SpMV and orthogonalization steps.

V. Results

We first demonstrate and explain the strong scaling results and their parallel efficiencies, and then we discuss our NCCL/CUDA variants and how each variant performs.

A. Strong Scaling

Figure 8 shows the achieved speedups of three proposed schemes over baseline MPI/OpenACC scheme. For each test problem, we start with the minimum GPU count needed and increase this number to assess this scheme's strong-scaling performance. For example, on Medium, we first use 91 GPUs on which the MPI/CUDA achieves $1.35\times$ (left) and P2P/CUDA achieves $2.25\times$ (middle) and NCCL/CUDA achieves $3.04\times$ (right) speedups, respectively.

The speedups gained from MPI/CUDA indicate performance improvement from computations leveraging the fine-grained control of architectural parallelism and the fast shared memory of GPUs. The extra speedups gained from P2P/CUDA to MPI/CUDA indicate improved performance from a faster communication and reduced load imbalance effect. The additional speedups obtained from NCCL/CUDA to P2P/CUDA indicate further improvement by leveraging GPU's massive parallelism for large messages.

MPI/CUDA Results. By changing the local SpMV and SpMV transpose kernels originally implemented using OpenACC directives to hand-tuned CUDA implementations, we achieve up to $2\times$ speedup. However, we observe that the highest speedup is on the lowest GPU count for each problem. Taking the Small problem as an example, the $2.00\times$ speedup obtained on 45 GPUs gradually diminishes to an only 3% improvement on 276 GPUs (left of Figure 8), essentially indicating no advantage of using CUDA over OpenACC in any test cases on the highest GPU count. This is due to the MPI collectives becoming the bottleneck as we increase the number of GPUs in the strong-scaling experiments. As Table II

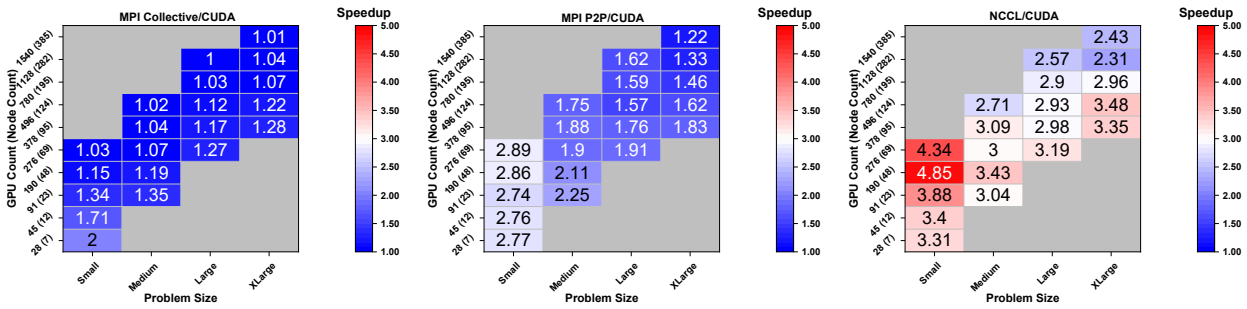


Fig. 8: Speedup achieved with MPI/CUDA, P2P/CUDA and NCCL/CUDA over MPI/OpenACC for 100 Lanczos iterations.

shows, the communication takes 62% of the total solver time for the Small case, and nearly 80% for the Medium case.

P2P/CUDA Results. We consistently observe around 2.8× speedup over MPI/OpenACC for the Small problem. The communication speedup can be obtained by subtracting the speedup factors in the left subfigure from the corresponding ones in middle subfigure. For example, the predicted maximum speedup for the Small problem in Table II is 1.61× using 190 GPUs, but we observe a 1.71× speedup in practice. This is because the potential speedup listed in the table is based on the speed of communication only. It does not account for load imbalance. The additional speedup is gained from reducing load imbalance between diagonal and off-diagonal ranks.

As we use more GPUs, we see a decline in the relative performance. For example, for Medium, we start with a 2.3× speedup on 91 GPUs and end up with 1.8× speedup on 496 GPUs. For the XLarge problem, we only observe a 22% improvement on 1540 GPUs. Such performance degradation results from the combined effect of diminishing gain from P2P communication and worse load balance.

NCCL/CUDA Results. NCCL/CUDA achieves the highest speedup among the three proposed optimizations. NCCL's advantages over P2P/CUDA are immediately apparent. NCCL leverages GPU parallelism for communications, and with careful synchronization and stream priorities manipulation, the NCCL/CUDA results in improvements over P2P/CUDA. We will discuss the NCCL/CUDA details in Sec. V-C.

B. Parallel Efficiency

Figure 9 shows the strong scalability and parallel efficiency for the four implementations of MFDn. We show two problem sizes, i.e., Small (minimum GPU counts is 28) and XLarge (minimum GPU counts is 378). The number of GPUs increases from 28 all the way to 276 for the Small test size. We take the NCCL/CUDA implementation as the baseline (shortest run time) for parallel efficiency. Thus, the baseline of the Small size is NCCL/CUDA using 28 GPUs, which denotes a 100% efficiency. The parallel efficiency is then obtained by $\frac{P_{baseline} \times T_{baseline}}{P \times T}$, where P refers to the total number of GPUs and T is the time. One can immediately observe that MPI/CUDA improves parallel efficiency at a small GPU count from 30% to 60%. The efficiency is flattened at a large number of GPUs, i.e., 11% and 12% parallel efficiency at 276 GPUs. This is because the non-overlapped communication dominates the total run time of MFDn in the two cases. Thus, the P2P/CUDA scheme further improves its parallel efficiency: 84% parallel efficiency at 28 GPUs and 32% at 276 GPUs. Eventually, NCCL/CUDA outperforms the other schemes. The gap between the observed NCCL/CUDA scaling and ideal scaling is due to the orthogonalization and atomic updates in addition to the increased communication overheads, which together make MFDn lose efficiency at scale.

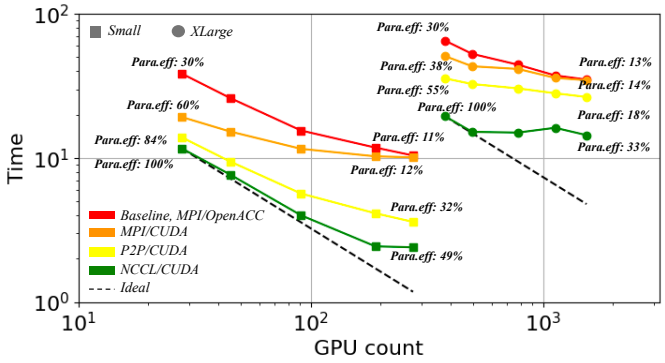


Fig. 9: Strong-scaling and parallel efficiency of Small and XLarge cases for 100 Lanczos iterations. We take NCCL/CUDA implementation as the baseline for parallel efficiency. MPI/CUDA can improve the parallel efficiency at a small GPU count. P2P/CUDA can further improve the efficiency by faster communication and reduced load imbalance effect, and finally, NCCL/CUDA outperforms all three implementations because it further leverages GPU's massive parallelism for large messages.

For the XLarge testcase, the number of GPUs start from 378. We observe that the parallel efficiency drops from 30% to 13%, 38% to 14%, 55% to 18% and 100% to 33% for MPI/OpenACC, MPI/CUDA, P2P/CUDA and NCCL/CUDA, respectively. The drop in efficiency from XLarge to Small is mainly due to the decrease in the ratio between FLOPs and communication volume.

To see this, the FLOP count in the SpMV is proportional to the number of nonzeros in the sparse matrix Hamiltoninan, which is fixed for a specific problem. The communication

volume required for a distributed SpMV performed over $P = n_d(n_d+1)/2$ GPUs in MFDn is proportional to

$$n_d \cdot \frac{n_d+1}{2} \cdot \frac{n}{n_d} \cdot \frac{n_d-1}{n_d+1} = \frac{n(n_d-1)}{2}, \quad (1)$$

where n is the dimension of H and the number of elements in W. We assume W is partitioned evenly among $(n_d+1)/2$ GPUs and each GPU sends or receives $(n_d-1)/2$ subvectors in each collective communication using the ring algorithm, which seems to be the default reduction algorithm in NCCL. If the number of nonzeros in the Hamiltonian is a constant multiple of n, the ratio between the total FLOPs and communication volume scales as $1/n_d$ or $1/\sqrt{P}$.

Table IV shows the ratio between the measured total FLOP count and total communication volume, which corresponds to the number of operations performed per byte communicated. We see that FLOPs/Byte ratio decreases with respect to P. Therefore, the lower FLOPs/Byte is the reason for the reduced efficiency at scale.

TABLE IV: Number of floating point operations performed per byte communicated in the distributed SpMV algorithm.

Test Case	# of GPUs	Tot FLOPs	Tot Comm Bytes	FLOPs/Byte
Small	28	5.16E11	3.97GB	130
	276	5.16E11	14.6GB	35.4
XLarge	378	7.25E12	185GB	39.0
	1540	7.25E12	385GB	18.8

C. NCCL/CUDA variants

Figure 10 summarizes the speedups achieved with different variants of NCCL/CUDA over the baseline MPI/OpenACC implementation for 100 Lanczos iterations. NCCL/CUDA variants can be described as follows:

Variant-1: In this variant, we use as few synchronization calls as possible. All CUDA kernels and NCCL calls are assigned to asynchronous CUDA streams which are synchronized only when there is data dependency between them.

Variant-2: Variant-2 is the same as Variant-1, except that before each NCCL call, we add an `MPI_Barrier` on the corresponding communicator and a `cudaDeviceSynchronize()`. We observe in the second subplot from the left in Figure 10 that adding barriers improve the solver time in 17 out of 20 cases (four test cases, five strong-scaling experiments each). For Medium on 496 GPUs, for example, adding such synchronization leads to roughly 60% improvement. This is because, although NCCL initiates communication, the operations may not complete immediately. Adding `MPI_Barrier` ensures synchronization across ranks, preventing some from progressing too soon while others are still catching up, thereby minimizing delayed message handling.

Variant-3: This variant truly overlaps a GPU kernel with a NCCL call. Based on Variant-2, we launch the GPU kernel first, followed by the NCCL call on the CPU side. We assign a higher priority to the stream used by NCCL. This way, the NCCL call can not block the GPU kernel, while the CUDA runtime can still allocate resources needed by the NCCL call due to its higher priority. We see in the second subplot from the right in Figure 10 that this variant performs better than Variant-2 in all but 2 cases. For example, we see a 33% improvement over Variant-2 for the Large case on 496 GPUs (22.9s vs 17.2s).

Variant-4: Our last optimization effort with NCCL/CUDA is to adjust the number of thread blocks used by the NCCL communicators based on Variant-3. NCCL by default selects a thread block count that can saturate the memory bandwidth whenever we create a NCCL communicator. NCCL then assigns that many thread blocks to the communicator each time it is used. However, this default number may not always yield the best performance. One can observe that this variant performs better than the Variant-3 in 19 out of 20 cases in the right plot of Figure 10.

Table V summarizes the slowdown that occurs when assigning a non-optimal thread block count for NCCL. These numbers suggest that configuring the communicators to use two thread blocks yields the worst performance in general. It is worth noting that this configuration can be up to 92% slower on XLarge. Using four thread blocks, on the other hand, work well on the Small test case although it can be up to 20% and 30% slower than the best option on Medium and XLarge, respectively.

TABLE V: Percentage of slowdown observed over the optimal performance when the NCCL communicators are configured to use 2, 4, 8, 12 and 16 thread blocks (TB) for Variant-4. OPT refers to Optimal configuration.

Test Case	# of GPUs	2 TB	4 TB	8 TB	12 TB	16 TB
Small	28	4.8%	1.8%	0.5%	**Opt**	0.3%
	45	1.9%	**Opt**	0.5%	0.7%	1.0%
	91	2.0%	**Opt**	1.0%	1.3%	1.8%
	190	4.6%	**Opt**	1.9%	1.6%	2.2%
	276	16%	0.5%	**Opt**	3.3%	6.4%
Medium	91	33%	12%	3.0%	3.5%	**Opt**
	190	35%	14%	7.0%	3.6%	**Opt**
	276	30%	2.2%	0.3%	**Opt**	0.3%
	378	48%	20%	**Opt**	11%	5.9%
	496	23%	13%	3.6%	**Opt**	6.9%
Large	276	53%	15%	**Opt**	4.0%	0.1%
	378	39%	4.6%	0.1%	**Opt**	0.3%
	496	43%	15%	6.2%	**Opt**	7.3%
	780	29%	12%	3.4%	2.2%	**Opt**
	1128	13%	2.4%	2.1%	**Opt**	0.7%
XLarge	378	81%	30%	13%	**Opt**	-
	496	92%	25%	1.6%	**Opt**	1.0%
	780	-	-	1.5%	**Opt**	4.4%
	1128	34%	6.3%	3.4%	**Opt**	4.2%
	1540	54%	15%	13%	0.1%	**Opt**

For the remaining three options, there is a trade-off when transitioning from 8 to 16 thread blocks for NCCL calls. While eight blocks may result in slightly slower NCCL communication, it allows more streaming multiprocessors (SMs) to complete the concurrent GPU kernels a bit faster. As such, the overall solver time does not vary much when we go from using 8 to 16 thread blocks. We find 8, 12, and 16 thread block configurations suitable for our experiments since all

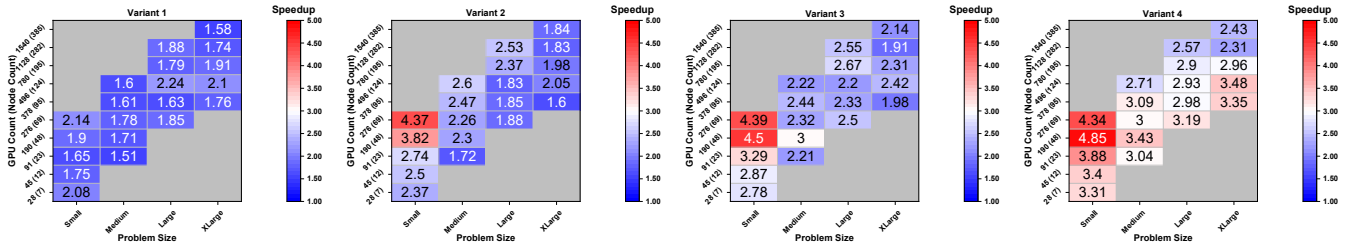


Fig. 10: Speedup achieved with NCCL/CUDA for all variants over MPI/OpenACC for 100 Lanczos iterations.

three configurations are never more than 13% slower than the optimal configuration in our tests.

The message size per GPU (or MPI rank) remains constant for a given test case and number of GPUs. When multiple thread blocks are used in communication, the message is evenly distributed among them, causing the message size per thread block to vary with the number of thread blocks. Better performance is achieved with a relatively higher number of thread blocks in Table V because the increased number of overlapped messages more effectively hides latencies. Conversely, performance declines when the number of thread blocks exceeds the optimal value ('OPT'), as scheduling additional blocks for a single message introduces unnecessary overhead.

Essentially, parallel configuration in NCCL is essential at run time. However, how to configure NCCL heavily depends on the application's communication patterns and the underlying network architecture. Proper synchronizations and stream priorities are also critical for designing and implementing high-performance NCCL applications.

D. Comparison with other work

Many of the existing SpMV (or SpMM) algorithms [17–19] use specific sparse matrix representations (such as the compressed row/column format) that are not applicable to MFDn. A special sparse matrix storage scheme is used for MFDn to reduce memory usage. We explored cuSPARSE to improve local SpMV. cuSPARSE is a library provided by NVIDIA that offers optimized sparse matrix operations on NVIDIA GPUs. We investigated various algorithmic options provided by the *cusparseSpMV()* function through the *cusparseSpMVAlg_t* parameter. The optimal one achieves performance comparable to that of our CUDA implementation. However, in the cuSPARSE implementation, we must store the matrix in CSR and CSC formats to avoid atomics in SpMV and transposed SpMV operations. Thus, it doubles the memory requirements, which is a critical limitation for many MFDn experiments, as these are applications with high memory capacity requirements. As such, we did not list cuSPARSE in the paper.

E. Programmer Productivity and Portability

Application developers often find it burdensome to maintain several codebases for different architectures. While the baseline MPI/OpenACC has a relatively good portability across architectures, it suffers from poor performance.

The MPI/CUDA implementation requires the most coding effort among the three optimization schemes. For example, it requires completely re-writing the OpenACC code in CUDA. However, thanks to emerging code development tools, such as HIPify [20], which automatically translates CUDA source code into HIP, and oneAPI [21], which converts CUDA into SYCL, MPI/CUDA option can be portable via those tools on AMD and Intel GPUs.

Starting with the MPI/CUDA version, P2P/CUDA version is a straightforward implementation task. In addition, the GPU-aware MPI has good portability across architectures. RCCL [22] and HCCL [23] can be considered replacements for NCCL on AMD and Intel clusters for inter-GPU collective communications.

Ultimately, with the help of the existing tools, the NCCL/CUDA (highest performance implementation) has the potential to be portable to other architectures, allowing a single code base that can be executed on different architectures with relatively low maintenance efforts.

VI. Conclusion

We presented several techniques to improve the performance the distributed SpMV used in the MFDn software for performing large-scale nuclear configuration interaction calculations on NVIDIA GPUs. We showed that, on problem sizes with up to 1.3 trillion nonzeros, we could obtain up to 2.0× improvement in the overall solver time by switching from OpenACC to a hand-optimized CUDA implementation across 1540 GPUs. Additionally, when combined with asynchronous P2P message passing and NCCL, we observed performance boosts of 2.9× (P2P) and 4.9× (NCCL). All these improvements resulted from using alternative programming models that involved using CUDA in place of OpenACC, asynchronous P2P message passaging in place of standard MPI collectives, and NCCL in place of CUDA-aware MPI. The relative large performance gap between the baseline MPI/OpenACC implementation and the NCCL/CUDA implementation suggests that the currently used directive based programming model and the standard MPI library, while providing a portable solution with a relatively small amount development effort, can severely limit the performance of SpMV.

To achieve higher performance on a single GPU, lower level programming protocals (such as CUDA) that give the devel-

oper a fine-grained control over shared memory as software cache. For large-scale problems solved on a large number of GPUs, the SpMV used in MFDn is communication bound. Therefore, optimizing the communication among different GPUs becomes more important at a large scale. To achieve better performance on distributed GPU nodes, we must make sure that the arithmetic operations in collective communications are executed on the device. Vendor developed communication libraries such as NCCL appear to offer additional features and functionalities that can reduce communication overhead. However, to fully take advantage of these features, we need to use such libraries judiciously by including appropriate synchronization points, maximizing the overlap between communication and computation, as well as choosing the optimal number of thread blocks for certain communications.

While the primary focus of this work is on optimizing the performance of MFDn, the insights and solutions we provide are likely to be relevant to a broad range of applications.

REFERENCES

[1] B. R. Barrett, P. Navratil, and J. P. Vary, "Ab initio no core shell model," *Prog. Part. Nucl. Phys.*, vol. 69, pp. 131–181, 2013.

[2] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le, "Accelerating configuration interaction calculations for nuclear structure," in *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. IEEE, 2008, pp. 1–12.

[3] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *J. Res. Nat'l Bur. Std.*, vol. 45, no. 4, pp. 255–282, 1950.

[4] M. Shao, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver," *Computer Physics Communications*, vol. 222, pp. 1–13, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465517302904

[5] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Topology-aware mappings for large-scale eigenvalue problems," in *European Conference on Parallel Processing*. Springer, 2012, pp. 830–842.

[6] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1213–1222.

[7] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of a symmetric iterative eigensolver for multi-core platforms," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 16, pp. 2631–2651, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3129

[8] D. Oryspayev, H. M. Aktulga, M. Sosonkina, P. Maris, and J. P. Vary, "Performance analysis of distributed symmetric sparse matrix vector multiplication algorithm for multi-core architectures," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5019–5036, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3499

[9] B. Cook, P. Maris, M. Shao, N. Wichmann, M. Wagner, J. O'Neill, T. Phung, and G. Bansal, "High performance optimizations for nuclear physics code mfdn on knl," in *International Conference on High Performance Computing*. Springer, 2016, pp. 366–377.

[10] B. Cook, P. J. Fasano, P. Maris, C. Yang, and D. Oryspayev, "Accelerating quantum many-body configuration interaction with directives," in *International Workshop on Accelerator Programming Using Directives*. Springer, 2021, pp. 112–132.

[11] P. Maris, C. Yang, D. Oryspayev, and B. Cook, "Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on gpus using openacc," *Journal of Computational Science*, vol. 59, p. 101554, 2022.

[12] NERSC, "Perlmutter Node Specifications," https://docs.nersc.gov/systems/perlmutter/architecture/#node-specifications, 2024, accessed: 2024-08-28. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/#node-specifications

[13] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 75–84.

[14] K. Z. Ibrahim, C. Yang, and P. Maris, "Performance portability of sparse block diagonal matrix multiple vector multiplications on gpus," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2022, pp. 58–67.

[15] N. Ding, M. Haseeb, T. Groves, and S. Williams, "Evaluating the performance of one-sided communication on cpus and gpus," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1059–1069.

[16] N. Ding, S. Williams, H. A. Nam, T. Groves, M. G. Awan, L. Lindsey, C. Daley, O. Selvitopi, L. Oliker, and N. Wright, "Methodology for evaluating the potential of disaggregated memory systems," in *2022 IEEE/ACM International Workshop on Resource Disaggregation in High-Performance Computing (REDIS)*. IEEE, 2022, pp. 1–11.

[17] G. Chu, Y. He, L. Dong, Z. Ding, D. Chen, H. Bai, X. Wang, and C. Hu, "Efficient algorithm design of optimizing spmv on gpu," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023, pp. 115–128.

[18] H. Cui, N. Wang, Y. Wang, Q. Han, and Y. Xu, "An effective spmv based on block strategy and hybrid compression on gpu," *The Journal of Supercomputing*, pp. 1–22, 2022.

[19] J. Gao, W. Ji, and Y. Wang, "Optimization of large-scale sparse matrix-vector multiplication on multi-gpu systems," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 4, pp. 1–24, 2024.

[20] AMD, "HIPIFY," https://github.com/ROCm/HIPIFY.git, 2024, accessed: 2024-10-09. [Online]. Available: https://github.com/ROCm/HIPIFY.git

[21] INTEL, "Migrate from CUDA to C++ with SYCL," https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/migrate-from-cuda-to-cpp-with-sycl.html#gs.gfvfnb, 2024, accessed: 2024-10-09. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/training/migrate-from-cuda-to-cpp-with-sycl.html#gs.gfvfnb

[22] AMD, "RCCL documentation," https://rocm.docs.amd.com/projects/rccl/en/latest/, 2024, accessed: 2024-10-09. [Online]. Available: https://rocm.docs.amd.com/projects/rccl/en/latest/

[23] INTEL, "Habana Collective Communications Library (HCCL) API Reference," https://docs.habana.ai/en/latest/API_Reference_Guides/HCCL_APIs/index.html, 2024, accessed: 2024-10-09. [Online]. Available: https://docs.habana.ai/en/latest/API_Reference_Guides/HCCL_APIs/index.html