Accelerating quantum many-body configuration interaction with directives

Brandon Cook[1*], Patrick J. Fasano[2[0000−0003−2457−4976]], Pieter Maris[3], Chao Yang[4], and Dossay Oryspayev[5]

[1] National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA, USA bgcook@lbl.gov
[2] Department of Physics, University of Notre Dame, Notre Dame, IN, USA pfasano@nd.edu
[3] Department of Physics and Astronomy, Iowa State University, Ames, IA, USA pmaris@iastate.edu
[4] Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA cyang@lbl.gov
[5] Computational Science Initiative, Brookhaven National Laboratory, Upton, NY, USA doryspaye@bnl.gov

Abstract. Many-Fermion Dynamics—nuclear, or MFDn, is a configuration interaction (CI) code for nuclear structure calculations. It is a platform-independent Fortran 90 code using a hybrid MPI+X programming model. For CPU platforms the application has a robust and optimized OpenMP implementation for shared memory parallelism. As part of the NESAP application readiness program for NERSC's latest Perlmutter system, MFDn has been updated to take advantage of accelerators. The current mainline GPU port is based on OpenACC. In this work we describe some of the key challenges of creating an efficient GPU implementation. Additionally, we compare the support of OpenMP and OpenACC on AMD and NVIDIA GPUs.

Keywords: Fortran · GPUs · OpenACC · OpenMP · Accelerators · Nuclear Configuration Interaction.

1 Introduction

Many-Fermion Dynamics—nuclear, or MFDn, is a configuration interaction (CI) code for *ab initio* nuclear physics calculations. It calculates the approximate many-body wave function of self-bound atomic nuclei, starting from two- and three-nucleon interactions. In these calculations, the nuclear many-body Hamiltonian is represented as a large sparse symmetric matrix in configuration space. The lowest eigenvalues of this matrix correspond to the energy levels of the low-lying spectrum, and the eigenvectors represent the corresponding many-body wave function.

* corresponding author

Table 1. Test platforms

	Cori GPU	Cori DGX	Spock
GPU	NVIDIA V100	NVIDIA A100	AMD MI100
CPU	Intel Skylake	AMD Rome	AMD Rome
GPUs per node	8	8	4
CPUs per node	2	2	1
Bus	PCIe 3.0	PCIe 4.0	PCIe 4.0
per GPU Memory	16GB	40 GB	32 GB

Table 2. Compilers. Cray Fortran compiler version 12.0.1 claims full OpenACC 2.0 and partial OpenACC 2.6 support for Fortran.

Vendor	Version	`_OPENACC`	`_OPENMP`	V100	A100	MI100
NVIDIA	21.7	201711 (2.6)	202011 (5.1)	✓	✓	
HPE/Cray	12.0.1	201306 (2.0)	201511 (4.5)	✓		✓

MFDn is a platform-independent Fortran 90 code using a hybrid MPI+X programming model. Over the past decade, it has been successfully deployed on multi-core supercomputers such as Jaguar at the Oak Ridge Leadership Class Facility (OLCF), Mira at the Argonne Leadership Class Facility (ALCF), and Edison at the National Energy Research Scientific Computing Center (NERSC) using MPI + OpenMP [2, 10, 11, 13]. Currently it is in production on many-core systems [4, 5, 7] such as Cori at NERSC and Theta at ALCF, as well as other supercomputers worldwide.

As part of the NESAP application readiness program for NERSC's latest Perlmutter system, MFDn is being updated to take advantage of accelerators. The current mainline GPU port uses OpenACC. In this work we consider several kernels that are representative for some of the most time-consuming parts of MFDn [5, 12, 16]. We describe some of the challenges and limitations of running them efficiently on GPUs with OpenMP and/or OpenACC directives, using both the NVIDIA and Cray Fortran compilers. We test their performance on NVIDIA V100 "Volta", NVIDIA A100 "Ampere", and AMD MI100 GPUs, as well as on Intel multi-core CPUs; see Table 1 for more details of the test systems.

We performed tests on Cori GPU (NVIDIA V100 GPUs) and Cori DGX (NVIDIA A100) with the NVIDIA compiler only, whereas our tests on Spock (AMD MI100) are performed with the Cray compiler only (see Table 2). The Cray compiler version 12.0 was not available for the Cori GPU system at the time of writing and it does not support NVIDIA A100 GPUs. All tests were done on exclusively-allocated nodes. In order to minimize any NUMA effects, tests on GPUs used a single GPU, while tests on CPUs used a single socket.

In our implementations we have primarily explored three models: OpenACC with `parallel` and `loop` directives, i. e., no `kernels` directives; OpenMP target offload with prescriptive style directives (`teams distribute parallel do`); and OpenMP target with the new `loop` construct introduced in OpenMP 5.0. In all cases maintaining performance on CPUs and code maintainability was also

a priority. The NVIDIA compiler claims support for OpenACC 2.6 with "many features" from 2.7 and for a subset of OpenMP 5.1, while the Cray Fortran compiler claims support for OpenACC 2.0 with "partial support" for 2.6 and full support OpenMP 4.5 with partial support for 5.0. We highlight where we encountered missing features or shortcomings throughout and in particular in architectural specialization in Section 2.3 and reductions on arrays in Section 2.4.

2 Computational Motifs in Configuration Interaction code MFDn

The key computational challenges for MFDn are (1) efficient localization of the nonzero Hamiltonian matrix elements and evaluation of the corresponding matrix elements, and (2) efficient sparse matrix–vector and matrix–matrix products used in the solution of the eigenvalue problem, both while effectively using the available aggregate memory [5, 12, 16]. Figure 1 shows the overall structure of MFDn. The GPU port of the LOBPCG eigensolver [15] using OpenACC is described in [14]. In this work we concentrate on the matrix construction phase and the evaluation of observables, which each take up about one-quarter to one-third of the total runtime, while iterative eigensolving takes about one-third to one-half of the total runtime. In CI calculations, the many-body wave functions are approximated by an expansion in *many-body basis states*; in MFDn we use antisymmetrized products of single-particle states with quantum numbers (n, ℓ, j, m) (see, e.g., Ref. [17] for the meaning of these quantum numbers). The many-body basis states can then be characterized by the set of single-particle quantum numbers for each nucleon. It is convenient to group together many-body states with the same sets of values for the quantum numbers (n, ℓ, j), but different sets of values for the magnetic projection quantum numbers m. This grouping leads to a natural hierarchy in the sparsity structure of the Hamiltonian matrix, which in turn allows for efficient localization and evaluation of nonzero matrix elements. In addition, this grouping also facilitates an efficient block-diagonal preconditioner for the LOBPCG algorithm [15]. We refer to these groups of many-body states as *many-body basis orbitals*. To describe the sparsity structure, we furthermore define tiles as pairs of row and column many-body basis orbitals.

In order to efficiently locate the nonzero matrix elements we exploit this hierarchical structure, by first determining which tiles can contain nonzero matrix elements (lines 2 through 5 of Fig. 1) and next counting how many nonzero matrix elements there are (lines 6 through 8) in each tile. The actual construction of the sparse matrix starts in line 11, with the nonzero matrix elements and their location evaluated and stored in line 14 of Fig. 1. Note that the structure of the double loops starting in lines 2 and in line 7 is essentially the same; the corresponding motif is discussed in Section 2.1 below. Also the double loops starting in line 12 have a similar structure, except that in this case, the obtained results in the innermost loop are stored in an array; this motif is discussed in Section 2.3 below. (Note that this motif is also used implicitly in line 5 of Fig. 1.)

1. Enumerate and distribute many-body basis orbitals (Ψ_j)
2. Loop over column orbitals (Ψ_m)
   Loop over row orbitals (Ψ_l)
3. Compare the single-particle orbitals of Ψ_l and Ψ_m
   increment *tile_cnt* if up to $2d$ differences
   End Loop
   End Loop
4. Allocate arrays of length *tile_cnt* to store nonzero tiles (T_k)
5. Repeat 2 and store nonzero tiles ($T_k = (\Psi_l, \Psi_m)$):
   pairs of orbitals with up to $2d$ differences
6. Loop over tiles ($(\Psi_l, \Psi_m) = T_k$)
7. Loop over column states ($\Phi_j \in \Psi_m$)
   Loop over row states ($\Phi_i \in \Psi_l$)
8. Compare the single-particle states of Φ_i and Φ_j
   increment cnt_k if up to $2d$ differences
   End Loop
   End Loop
   End Loop
9. Convert cnt_k's to $offset_k$'s
10. Allocate arrays of length $\sum cnt_k$ to store H_{ij}
11. Loop over tiles ($(\Psi_l, \Psi_m) = T_k$)
12. Loop over column states ($\Phi_j \in \Psi_m$)
    Loop over row states ($\Phi_i \in \Psi_l$)
13. Compare the single-particle states of Φ_i and Φ_j
    cycle if more than $2d$ differences
14. Compute the nonzero matrix entry H_{ij} and store
    End Loop
    End Loop
    End Loop
15. Obtain lowest n eigenvalues E and eigenvectors $\vec{c}$ of H_{ij}
    using distributed LOBPCG or Lanczos algorithm
16. Loop over tiles ($(\Psi_l, \Psi_m) = T_k$)
17. Loop over column states ($\Phi_j \in \Psi_m$)
    Loop over row states ($\Phi_i \in \Psi_l$)
18. Compare the single-particle states of Φ_i and Φ_j
    cycle if more than $2d$ differences
19. Compute m nonzero matrix elements $O_{ij}(1:m)$
20. Update $a(1:n*m) = a(1:n*m) + c_i(1:n) * O_{ij}(1:m) * c_j(1:n)$
    End Loop
    End Loop
    End Loop

Fig. 1. Schematic outline of the structure of MFDn during the matrix construction phase (lines 2 through 14) and evaluation of physical observables (lines 16 through 20), for a d-body Hamiltonian and d-body operators for observables.

Once the nonzero matrix elements are evaluated and stored, we can obtain the lowest n eigenvalues and eigenvectors $\vec{c}$ using an iterative eigensolver. In MFDn we use either LOBPCG or a Lanczos algorithm with reorthogonalization, as indicated in line 15 of Fig. 1. Details of the GPU port of our LOBPCG eigensolver using OpenACC are described in Ref. [14].

Finally, in lines 16 through 20 of Fig. 1 we calculate m different physical observables using the coefficients c_i of the lowest n eigenvectors and m two-body operators. Typically, we use 8 or 16 eigenvectors, and up to $m \sim 16$ different operators corresponding to different observables. Note that lines 16 to 18 have the same structure as lines 11 to 13, but instead of storing the nonzero matrix elements of the operators, we contract them with the n eigenvectors. The corresponding motif for these loops is discussed in Section 2.4 below.

2.1 Matrix Sparsity Determination

A typical loop structure in the matrix construction phase, as well as in the evaluation of observables, is shown in Fig. 2. The (i, j)th entry of the many-body Hamiltonian with a d-body interaction, $H_{ij} = \langle \Phi_i | H | \Phi_j \rangle$, can only be nonzero if the many-body states Φ_i and Φ_j differ by at most $2d$ single-particle states. Thus, the first step in the matrix construction (and in the evaluation of observables given by the expectation value of a d-body operator) is to determine which matrix elements can be nonzero. This is done in line 3 of the loop; subsequently, lines 4 and 5 indicate the actual evaluation of the nonzero matrix entry, which can be stored in memory (see line 14 of Fig. 1) , or directly used in a matrix–vector multiply or vector–matrix–vector contraction for the calculation of expectation values (see line 20 of Fig. 1). These are more complicated operations which are accomplished by separate (sequential) subroutine calls, the details of which are beyond the scope of this work.

```
1. Loop over column states (Φ_j)
2.    Loop over row states (Φ_i)
3.       Compare the single-particle states of Φ_i and Φ_j
         cycle if more than 2d differences
4.       (optional) Compute the nonzero matrix entry H_ij and store
5.       (optional) FMA of H_ij with ith row (and jth column) vector element
6.    End Loop
7. End Loop
```

Fig. 2. A typical loop structure in the matrix construction phase and during the evaluation of physical observables.

The localization of nonzero matrix elements involves determining which many-body basis states may be connected by the given particle rank (e. g., two-body) Hamiltonian. Given a single-particle basis with $n_{\text{s.p.}}$ single-particle states,

a many-body basis state Φ_i for fermionic systems can be represented by a binary string of length $n_{\text{s.p.}}$, denoted by $\texttt{BIN}(\Phi_i)$, where each binary bit of $\texttt{BIN}(\Phi_i)$ indicates whether the corresponding single-particle state is occupied. The total number of particles in the many-body state Φ_i is the number of 1's in $\texttt{BIN}(\Phi_i)$, i. e., the population count $\texttt{popcount}(\texttt{BIN}(\Phi_i))$. Information regarding all differently-occupied single-particle states between two bit-representations $\texttt{BIN}(\Phi_i)$ and $\texttt{BIN}(\Phi_j)$ is encoded by $\texttt{BIN}(\Phi_i) \oplus \texttt{BIN}(\Phi_j)$, where $\oplus$ denotes the bitwise exclusive-or operation. The number of differently-occupied single-particle states is then $\texttt{popcount}(\texttt{BIN}(\Phi_i) \oplus \texttt{BIN}(\Phi_j))$. If both bit-representations describe states with the same number of particles, then the number of differently-occupied single-particle states is always even; with a two-body potential, only many-body matrix elements with 0, 2, or 4 differently-occupied single-particle states can be nonzero. If there are more than four differently-occupied single-particle states the matrix element must be zero.

The storage requirements of such a bit representation is proportional to the number of single-particle states, but independent of the number of particles in the system. Alternatively, one can represent an N-body basis state Φ_i by an array or tuple of N short integers $\texttt{MBS}(a_1{:}a_N)$ with $a_1 < a_2 < \cdots < a_N$, where each element a_i indicates which single-particle state is occupied. The storage requirement of this method is proportional to N, the number of particles, but independent of the number of single-particle states $n_{\text{s.p.}}$. For a relatively small number of particles ($N \sim 10 - 20$ in MFDn), but a large ($n_{\text{s.p.}} \gtrsim 1,000$) number of single-particle states, the `MBS` representation is more advantageous in terms of memory than storing the states as a bit representation. However, determining the differently-occupied single-particle states is significantly more complex when Φ_i and Φ_j are in the `MBS` representation (see Listing 1). Note the factor of two to ensure consistent counts with the population count on the bit representation.

In MFDn the low-lying single-particle states are most likely to be occupied. For this reason we use a bit representation for the 64 lowest single-particle states, in combination with an array of N 16-bit integers $\texttt{MBS}(a_1{:}a_N)$ to store the full many-body state. This allows for efficient filtering of pairs of states with bit arithmetic on the most likely to be occupied states while incurring minimal additional storage overhead. Our OpenACC implementation of the algorithm used to count the number of different single-particle states by first performing a population count on the 64 lowest single-particle states, followed by a detailed comparison of the `MBS` representation if the population count is at most $2d$, is given in code Listings 1 and 2.

OpenMP prescriptive, OpenMP loop and OpenMP loop with bind hints directives for the first and second level of parallelism in Listing 2 are shown in Listings 3 and 4 respectively. The first level is typically mapped to cores on CPUs, thread blocks on NVIDIA GPUs and workgroups on AMD GPUs while the second level is typically mapped to SIMD lane(s) on CPUs, threads on NVIDIA GPUs and work items on AMD GPUs. Note that the comparison of the two complete many-body states is performed in a function (see Listing 1), which needs a

```
integer function count_difference(s1, n1, s2, n2)
  integer, intent(in) :: n1, n2, s1(n1), s2(n2)
  integer :: i1, i2, d, diffs1, diffs2
  !$acc routine seq
  i1 = 1
  i2 = 1
  diffs1 = 0
  diffs2 = 0
  do
     if ( (i1 > n1) .or. (i2 > n2) ) exit
     d = s1(i1) - s2(i2)
     if (d < 0) then
        diffs1 = diffs1 + 1
        i1 = i1 + 1
     else if (d > 0) then
        diffs2 = diffs2 + 1
        i2 = i2 + 1
     else
        i1 = i1 + 1
        i2 = i2 + 1
     end if
  end do
  count_difference = 2*max(diffs1, diffs2)
end function count_difference
```

Listing 1: Sequential function for detailed comparison of two many-body states.

```
!$acc parallel loop
do i = 1, n
   c = 0
   !$acc loop reduction(+:c)
   do j = 1, n
      d = popcnt(ieor(bitrep1(i), bitrep2(j)))
      if (d > 4) cycle
      d = count_difference(mbstate1(:,i), np, mbstate2(:,j), np)
      if (d <= 4) c = c + 1
   end do
   !$acc end loop
   counts(i) = c
end do
!$acc end parallel loop
numnnz = sum(counts)
```

Listing 2: Counting nonzero matrix elements with OpenACC, using both a bit representation for the first 64 single-particle states of Φ_i and Φ_j and a detailed comparison for the `MBS` representation if needed. The highlighted lines show the directives used for the two levels of parallelism.

```
1 !$acc parallel loop
2 !$omp target teams distribute private(d)
3 !$omp target teams loop private(d)
4 !$omp target teams loop bind(teams) private(d)
```

Listing 3: OpenACC, OpenMP prescriptive, OpenMP loop and OpenMP loop with hints directives to express the first level of parallelism highlighted in line 1 of Listing 2.

```
1 !$acc loop reduction(+:c)
2 !$omp parallel do reduction(+:c) private(d)
3 !$omp loop reduction(+:c) private(d)
4 !$omp loop bind(parallel) reduction(+:c) private(d)
```

Listing 4: OpenACC, OpenMP prescriptive, OpenMP loop and OpenMP loop with hints directives to express the second level of parallelism highlighted in line 4 of Listing 2.

`!$acc routine seq` directive so that the OpenACC loops in Listing 2 do indeed get parallelized; with OpenMP there is no need for a similar directive when the routine is defined in the same compilation unit, though `!$omp declare target` may be used when this is not the case.

For our performance tests we used many-body states with 4, 8, 12, 16 and 20 particles, 128 single-particle states, and a bit representation based on only the lowest 64 single-particle states. We randomly generated many-body states biased towards the lowest states, and counted the number of nonzero matrix elements for a two-body interaction. We present in Figs. 3 and 4 results with 8 particles as the density of nonzeros (median density was 6×10^{-6}) most closely represents the regime of interest for MFDn. Note that fewer particles with the state generation scheme correspond to more nonzero elements and more particles result in fewer nonzero elements. For comparison and correctness, we also ran two additional versions: a version without the `popcount` on the bit representation, as well as a version with a complete bit representation of all 128 single-particle states and using exclusively the `popcount` on this extended bit representation.

Figure 4 shows the performance of the bit representation and combined versions of the counting routines on MI100 and A100 GPUs implemented with OpenACC and OpenMP. On NVIDIA and AMD GPUs the OpenACC directives provide the best performance in most cases. In several instances there were performance issues when a function or subroutine call was introduced: particularly with OpenMP, usually manifest as a failure to generate parallel code for the second level of parallelism. In the case of `!$omp loop` directives we found that it was necessary to include bind annotations to recover the performance obtained by the OpenACC implementations on NVIDIA GPUs. The performance difference between all versions and implementations is shown in Fig. 3.

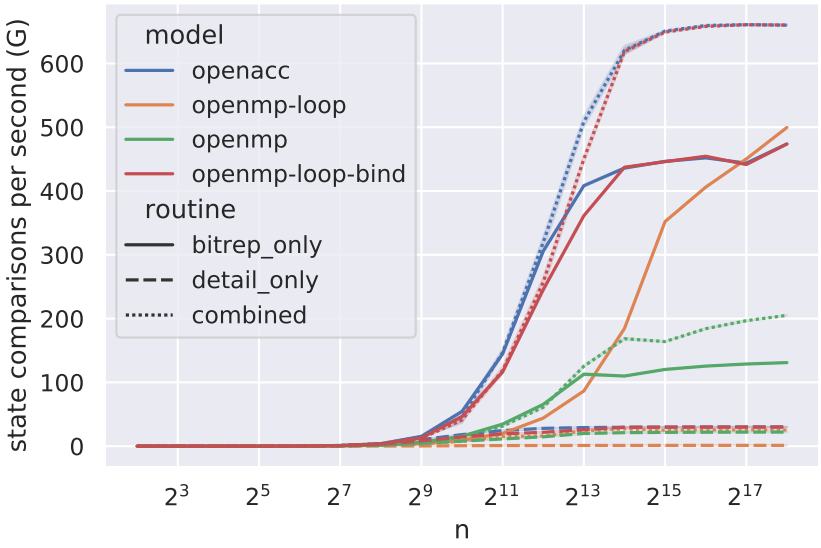

Fig. 3. Performance of interacting state counting routines on A100 with 8 particles (median density of nonzero elements is $\sim 6 \times 10^{-6}$). The vertical axis shows the number of state comparisons made per second, rate $= n^2/\text{time}$ (higher is better).

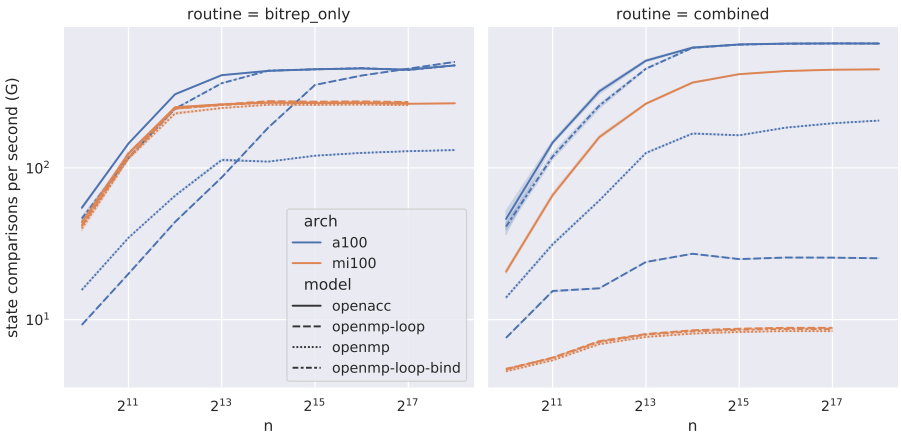

Fig. 4. Performance of bit representation only and combined interacting state counting routines on A100 and MI100 with 8 particles (median density of nonzero elements is 0.000006). The vertical axis shows the number of state comparisons made per second, rate $= n^2/\text{time}$ (higher is better).

On NVIDIA platforms we found there is overhead associated with OpenMP with prescriptive `!$omp target teams distribute` and `!$omp parallel do` directives compared to the `!$omp loop` directives. The compiler parallelizes on both teams and threads, but shows reduced performance. This is likely due to the additional semantic constraints for the `!$omp parallel` directive introducing some overhead in code generated by the compiler.

2.2 Parallel prefix sum

On multi-core CPUs, it is often convenient to use private arrays – and as long as there is sufficient memory, there is no intrinsic limitation on the private array size. In practice, it is limited by the `OMP_STACKSIZE` which the user can increase from its default value if necessary. Further, the use of thread-private arrays can often result in good performance as it helps ensure cache locality on CPUs. However, the situation on GPUs is different. Although private arrays can be used in both OpenACC and with OpenMP offload, there are more limitations on the size of such private arrays and/or on the number of gangs/teams or vector length one can use, due to the order of magnitude more parallelism available in GPUs. In particular in inner loops, private arrays should be avoided or limited to small arrays with only a handful of array elements; and even at the gang/team level, large arrays with (tens of) thousands of array elements severely limits the number of gangs/teams one can use.

In order to reduce (or better, completely avoid) the need for private arrays, it can be useful to convert counts, such as the counts of nonzero matrix elements discussed in Section 2.1, into offsets, so that one can use a single large shared array with appropriate offsets, instead of many allocatable private arrays. Specifically, converting counts x_i into offsets y_i can be implemented as

$$y_{i+1} = \sum_{j=0}^{i} x_j = y_i + x_i \tag{1}$$

with $y_1 = 0$, and is often referred to as a prefix sum or cumulative sum or scan. Here we focus on addition of integers, but in general only a binary associative operator is required.

On CPUs this operation is fast, and furthermore OpenMP 5.0 introduced a `scan` directive that extends reductions. However, this feature is not currently

```
1 !$acc serial present(x,y)
2 y(1) = 0
3 do i = 1, n-1
4    y(i+1) = y(i) + x(i)
5 end do
6 !$acc end serial
```

Listing 5: Serial prefix sum with OpenACC.

Algorithm 1 Work-efficient parallel prefix sum (Algos. 3 and 4 of Ref. [8])

```
1: for p ← 0, log2 n − 1 do                                  ▷ sweep up (or reduce)
2:     for j ← 0, n − 1 by 2^(p+1) do                        ▷ parallel
3:         y(j + 2^(p+1)) ← y(j + 2^p) + y(j + 2^(p+1))
4:     end for
5: end for
6: x(n) ← 0
7: for p ← log2 n,0 do                                       ▷ sweep down
8:     for j ← 0, n − 1 by 2^(p+1) do                        ▷ parallel
9:         tmp ← y(j + 2^p)
10:        y(j + 2^p) ← y(j + 2^(p+1))
11:        y(j + 2^(p+1)) ← tmp + y(j + 2^(p+1))
12:    end for
13: end for
```

supported for GPUs by the compilers tested in this work. OpenACC provides no equivalent directive. We note that production quality implementations of this operation may be available in C++ libraries such as Thrust [6] or Kokkos [1], but that including C++ or vendor specific frameworks in a Fortran code with a goal of portability introduces significant maintenance costs. In many cases it is preferable to avoid data transfers between the host and accelerator, even at the cost of inefficient use of the device. With OpenACC's `!$acc serial` directive a potentially expensive data transfer can be avoided as shown in Listing 5. Generally one should consider a performance model that includes bandwidth between host and device, performance on either host and device and the potential for any latency/ blocking effects introduced by the data motion when considering an implementation.

For prefix sums over large sequences, a parallel implementation can realize significant speedups. We note that in MFDn the offsets can be reused many times so that the overall impact for the application run time is small, but this common primitive can be found in many applications [3]. A work-efficient algorithm for a parallel scan is shown in Algorithm 1 - the work-efficient algorithm consists of an up and down sweep [3, 8]. The main idea is to sweep up and down a binary tree of the input data. The "up" or "reduce" sweep proceeds from the leaves to the root, computing partial sums in place. In the "down" sweep phase the binary tree is traversed from the last element down (root) to the leaves.

An implementation of Algorithm 1 in OpenACC is shown in Listing 6. As written it assumes power of two arrays; non-power of two arrays can be padded with zeros. We note that there are many possible further optimizations and refer interested readers to Refs. [3, 8]. In OpenACC, each `parallel` region will result in a new kernel launch, but with the `async` clause we can queue them all in non-blocking manner and rely that they will be executed in order. A similar approach can be implemented in OpenMP with `nowait` and `depend` clauses. Figure 5 shows the performance of serial and parallel prefix sum with OpenACC on A100, V100, and Skylake. Unfortunately the Cray compiler's partial support

```
!$acc data present(x,y)
!$acc parallel loop async
do j = 1, n
   y(j) = x(j)
end do
!$acc end parallel
offset = 1
! sweep up, reduction in place
do while (offset < n)
   !$acc parallel loop firstprivate(offset) present(y) async
   do concurrent (j=0:n-1:2*offset)
      y(j + 2*offset) = y(j + offset) + y(j + 2*offset)
   end do
   !$acc end parallel
   offset = 2*offset
end do
! sweep down, complete the scan
!$acc serial async
y(n) = 0
!$acc end serial
offset = rshift(offset, 1)
do while(offset > 0)
   !$acc parallel loop firstprivate(offset, tmp) present(y) async
   do concurrent(j=0:n-1:2*offset) local(tmp)
      tmp = y(j + offset)
      y(j + offset) = y(j + 2*offset)
      y(j + 2*offset) = tmp + y(j + 2*offset)
   end do
   !$acc end parallel
   offset = rshift(offset, 1)
end do
!$acc wait
!$acc end data
```

Listing 6: Parallel scan with OpenACC corresponding to Algorithm 1

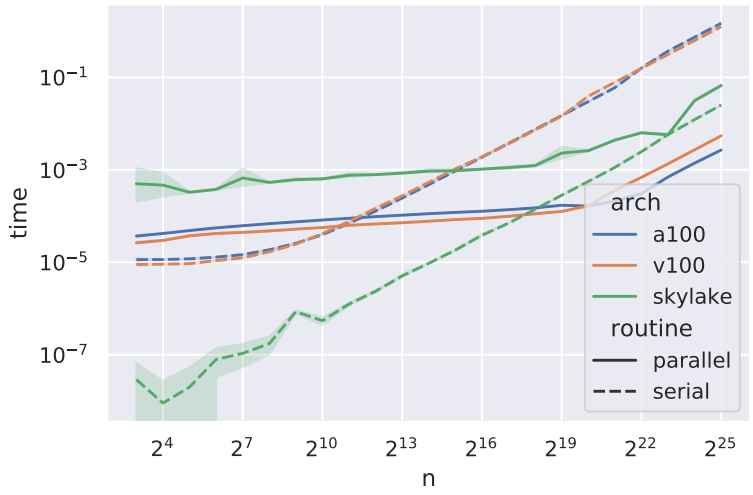

Fig. 5. Performance of prefix sum implemented with OpenACC on different architectures in parallel and serial (lower is better).

for OpenACC 2.6 does not include the `serial` directive so we do not include MI100 results for these implementations. For large arrays on A100 GPUs the parallel implementation can be over 500x faster than a serial implementation. This demonstrates the importance of support for serial work on accelerators (and corresponding constructs) for productivity, support for asynchronous work for performance and the desirability of a good language/ library support for parallel primitives such as prefix sums.

2.3 Filling shared arrays

After an initial pass to obtain the nonzero counts and offsets as described in Sections 2.1 and 2.2 a second pass is often performed to store relevant information such as a row or column index, or the nonzero value, into a global (shared) array, see e.g. line 14 of Fig. 1. Since we are using a multilevel hierarchical structure for the sparse matrix, this motif appears in several situations, not only for the matrix elements themselves. Generally there are two levels of parallelism: an outer loop, with no data dependencies, and an inner loop, where order does not matter but there is a dependency. The outer loop alone typically has enough parallelism to saturate a CPU but not a GPU.

On GPUs, OpenACC directives can be used to efficiently implement such a motif as shown in Listing 7. Equivalent directives are available in OpenMP. The `!$acc atomic capture` directive ensures that the value of the shared array element `indx(i)` gets incremented by one and assigned to the local (private) variable `k`, which can then be used as an index for filling the desired array with

```
1  !$acc parallel loop
2  do i = 1, n
3     indx(i) = offset(i)
4     !$acc loop device_type(host) seq
5     do j = 1, m
6        if (mod(j,p) == 0) then
7           !$acc atomic capture
8           indx(i) = indx(i) + 1
9           k = indx(i)
10          !$acc end atomic
11          arr(k) = j
12       end if
13    end do
14 end do
15 !$acc end parallel
```

Listing 7: Filling shared arrays on GPUs using OpenACC.

the appropriate value. On GPUs the performance penalty attributed to atomic operation in the inner loop is rather modest, and the exposed parallelism of the inner loop overwhelms this penalty, resulting in significant speedup over CPUs as shown in Fig. 6.

The same source code can also be compiled for and run on CPUs. On CPUs, when the parallelism available in the outermost level is sufficient, we can indicate that the inner loop should be sequential with the addition of the `!$acc device_type(host) seq` clauses. Unfortunately, the OpenACC specification does not support the `!$acc device_type` clause on `!$acc atomic` constructs. The performance of Listing 7 on multiple architectures is shown in Figure 6. With the atomic operation explicitly in the inner loop, this implementation performs worse on CPUs than necessary despite there being no contention between threads on the atomic operations due to the overhead of atomic semantics. In OpenMP 5.0 `metadirective` was introduced to support this use case, but compiler support is not available at the time of writing. In principle, `declare variant` in OpenMP or runtime calls in OpenACC to selectively choose between multiple subroutine versions could be used to enable performance on CPUs and GPUs with a single source at the expense of code duplication. Otherwise use of the preprocessor would be required.

2.4 Array Reductions

Finally, to compute physical observables one needs to calculate the expectation values of the corresponding operators O_k (see line 20 of Fig. 1):

$$a_k = \sum_{ij} x_i \, (O_k)_{ij} \, y_j \; . \tag{2}$$

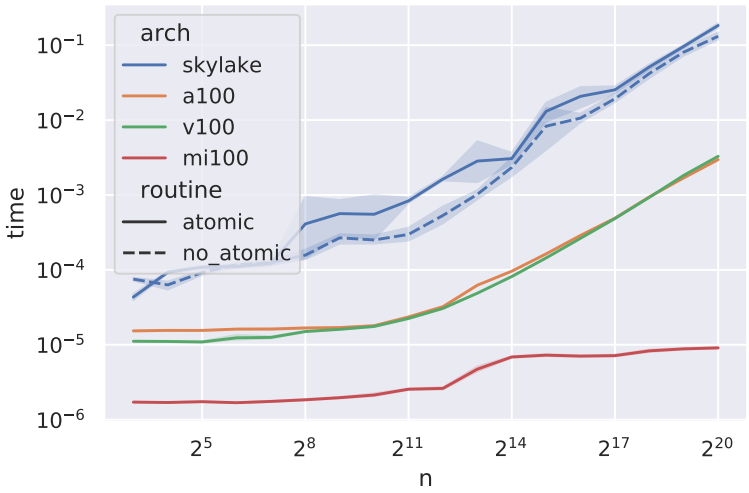

Fig. 6. Performance of filling a shared array with OpenACC (Listing 7) with $m = 512$ and $p = 1$ on MI100, A100, V100 and Skylake (lower is better).

In practice this requires a reduction of an array with a small number of elements. In the context of the MFDn application the array is typically of dimension $m \in [8, 256]$. Here we consider a simplified version of the motif (shown in Listing 8) that omits the check if two many-body states interact and any computation of additional physical matrix elements in order to explore programming model support for the motif. In this application trading memory for performance as in [9] is undesirable since it will limit scaling to large problems on full systems. We consider 3 implementations for "small" array reductions:

- reduction clauses with array arguments;
- atomic updates to individual array elements;
- generating a scalar reduction for each element of the array with `fypp`.

direct array reduction support Both the OpenACC 2.7+ and OpenMP 4.5+ specifications support arrays as arguments to reduction clauses in directives. However, compiler support for these features varies. With OpenACC, the Cray compiler only provides partial OpenACC 2.6 support and array reductions were not introduced until 2.7. The NVIDIA compiler also does not support arrays in reduction clauses with OpenACC. With OpenMP both Cray and NVIDIA compilers support array reductions. However, this feature was only introduced in NVIDIA's 21.7 release. With Cray the compiler warns, "An OpenMP teams construct with an array reduction is limited to a single team." With OpenMP, the NVIDIA compiler encounters a run time error as the array size is increased,

```
!$acc parallel loop collapse(2) reduction(+:a)
do i = 1, n
   do j = 1, n
      do k = 1, m
         a(k) = a(k) + x(k,i) * y(k,j)
      end do
   end do
end do
!$acc end parallel
```

Listing 8: Array reduction: array variable in a `reduction` clause.

```
!$acc parallel loop collapse(3)
do i = 1, n
   do j = 1, n
      do k = 1, m
         !$acc atomic
         a(k) = a(k) + x(k,i) * y(k,j)
         !$acc end atomic
      end do
   end do
end do
!$acc end parallel
```

Listing 9: Array reduction: atomic updates

or fails to compile when managed memory is used. Figure 7 shows the performance of array reduction on A100, MI100 and Skylake CPUs respectively where supported. Performance of array reductions was only competitive with other solutions on Skylake.

array reduction with atomics With atomic constructs we are able to compile a single version for all architectures. Compared to the case in Section 2.3 where there is no contention, when $n \gg m$, the contention is quite high which results in a significant performance penalty compared to an optimized reduction algorithm on the Skylake CPU as seen in the top panel of Fig. 7. Our results indicate that the implementation shown in Listing 9 can achieve reasonable performance on GPUs but not CPUs. We also note that there is an additional danger with the use of `collapse` with many loops, the combined iteration space may manifest a integer overflow for realistic array sizes if 32 bit integers are used as loop indices even though no individual loop overflows in a serial implementation.

generated scalar reductions In the case of small arrays another approach is to use a preprocessor that enables templating and metaprogramming such as

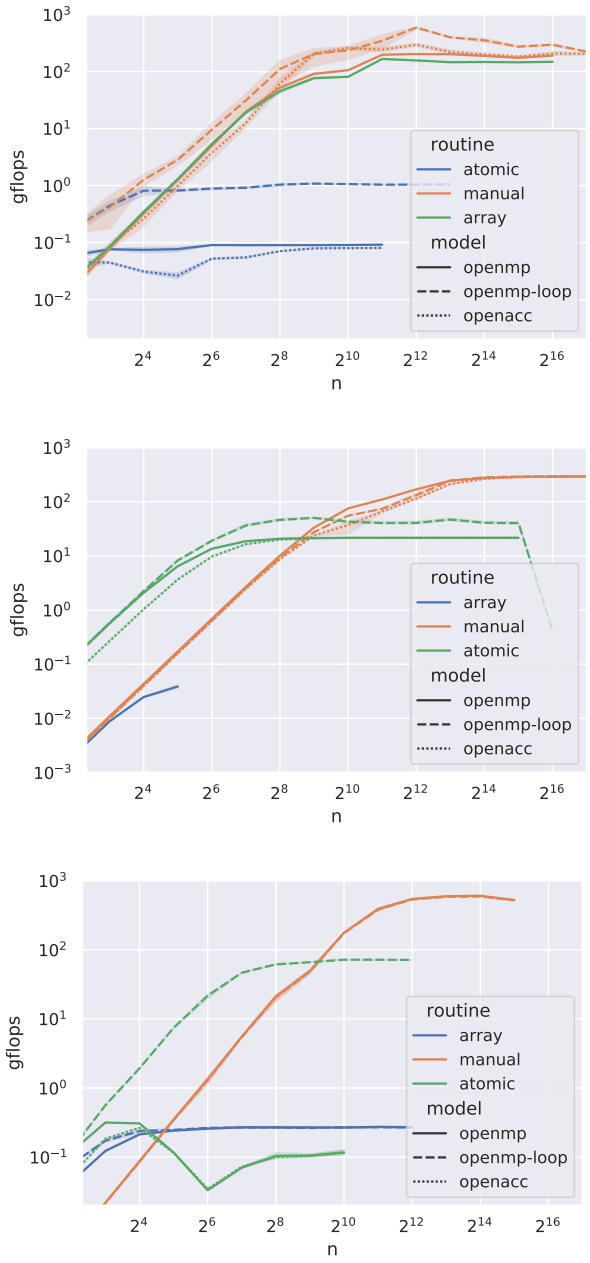

Fig. 7. Performance of array reduction with array size of 64 (higher is better).
Top: On Skylake CPUs; *Middle:* On A100, where we encountered run time errors for $n > 2^5$ with OpenMP array reduction and a compile error with OpenMP with loops; *Bottom:* On MI100, where for $mn^2 \geq 2^{32}$ there appears to be a correctness error due to integer overflow on the collapsed loops with the Cray compiler.

`fypp` to generate routines that implement a scalar reduction for each element of the array as in Listings 10 and 11. As shown in Fig. 7 this approach achieves good performance on all architectures. However, this approach works for small arrays, but as array size is increased runs into several issues either with directive line length (NVIDIA) or compiler register allocation routines (HPE/Cray). Further it can result in significant undesirable cognitive and compilation time overhead. In summary no one method for array reductions is best in all situations, but in this case the manually generated reductions on scalars with OpenACC give the best overall cross-platform performance.

3 Conclusion and Outlook

We highlighted several important features of programming for accelerators with directives that were key for an efficient GPU accelerated port of MFDn. Further we explored the performance implications of these modifications with CPUs and with multiple GPU and compiler vendors.

Avoiding use of private arrays in a production application that has undergone several years of optimization for multicore CPU platforms was a key challenge. The conversion of counts to offsets followed by indexing of shared array in a way that preserves CPU performance while enabling GPU offload was a key pattern that involved restructuring of many key data structures and routines in the application.

Our key points for application developers can be summarized as: avoid private arrays; check compiler diagnostic output to ensure parallel code is in fact generated; carefully check correctness along with performance; and be mindful of atomic operations when developing single source code for CPU and GPU architectures.

Our findings have shown several shortcomings of both the OpenACC and OpenMP specification/implementation with respect to specialization of code for different architectures that can hopefully be addressed in future editions of those specifications and compilers. Finally, we have identified several areas and motifs that compiler vendors may use to improve their products.

Acknowledgements

This work is supported by the U.S. Department of Energy (DOE) under Award Nos. DE-FG02-95ER40934 and DE-SC0018223 (SciDAC/NUCLEI), and by the DOE Office of Science, Office of Workforce Development for Teachers and Scientists, Office of Science Graduate Student Research (SCGSR) program (administered by the Oak Ridge Institute for Science and Education (ORISE), managed by ORAU under contract number DE-SC0014664).

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231, as well as resources of the Oak Ridge Leadership Computing Facility at the Oak

```
#:def CSV(x,n)
${",".join(f"{x}{i}" for i in range(1, n+1))}$
#:enddef CSV
#:for num_elements in range(2, max_elements+1)
  subroutine reduction${num_elements}$(x, y, a, n, dt)
    integer, parameter :: m = ${num_elements}$
    integer, intent(in) :: n
    real(sp), dimension(m, n), intent(in) :: x, y
    real(sp), intent(out) :: a(m)
    integer :: i,j
    real(dp) :: t0
    real(dp), intent(out) :: dt
#:for i in range(1, num_elements+1)
    real(sp) :: a${i}$
#:endfor
    !$acc data present(x,y)
    t0 = wtime()
#:for i in range(1, num_elements+1)
    a${i}$ = a(${i}$)
#:endfor
    !$acc parallel loop collapse(2) &
    !$acc reduction(+:${CSV("a",num_elements}$)
    do i = 1, n
       do j = 1, n
#:for i in range(1, num_elements+1)
          a${i}$ = a${i}$ + x(${i}$,i) * y(${i}$,j)
#:endfor
       end do
    end do
    !$acc end parallel
#:for i in range(1, num_elements+1)
    a(${i}$) = a${i}$
#:endfor
    dt = wtime() - t0
    !$acc end data
  end subroutine reduction${num_elements}$

#:endfor
```

Listing 10: Template code for generating reductions with `fypp`. Listing 11 shows a routine generated for $m = 3$.

```
1   subroutine reduction3(x, y, a, n, dt)
2     integer, parameter :: m = 3
3     integer, intent(in) :: n
4     real(sp), dimension(m, n), intent(in) :: x, y
5     real(sp), intent(out) :: a(m)
6     integer :: i,j
7     real(dp) :: t0
8     real(dp), intent(out) :: dt
9     real(sp) :: a1
10    real(sp) :: a2
11    real(sp) :: a3
12    !$acc data present(x,y)
13    t0 = wtime()
14    a1 = a(1)
15    a2 = a(2)
16    a3 = a(3)
17    !$acc parallel loop collapse(2) &
18    !$acc reduction(+:a1,a2,a3)
19    do i = 1, n
20       do j = 1, n
21          a1 = a1 + x(1,i) * y(1,j)
22          a2 = a2 + x(2,i) * y(2,j)
23          a3 = a3 + x(3,i) * y(3,j)
24       end do
25    end do
26    !$acc end parallel
27    a(1) = a1
28    a(2) = a2
29    a(3) = a3
30    dt = wtime() - t0
31    !$acc end data
32  end subroutine reduction3
```

Listing 11: Routine for $m = 3$ case generated by `fypp` code in Listing 10.

Ridge National Laboratory, which is supported by the DOE Office of Science under Contract No. DE-AC05-00OR22725.

Bibliography

[1] Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for CUDA. In: Hwu, W.m.W. (ed.) GPU Computing Gems Jade Edition, pp. 359–371. Applications of GPU Computing Series, Morgan Kaufmann, Boston (2012). https://doi.org/10.1016/B978-0-12-385963-1.00026-5

[2] Binder, S., Calci, A., Epelbaum, E., et al.: Few-nucleon systems with state-of-the-art chiral nucleon-nucleon forces. Phys. Rev. C **93**(4), 044002 (2016). https://doi.org/10.1103/PhysRevC.93.044002

[3] Blelloch, G.E.: Prefix sums and their applications. Tech. Rep. CMU-CS-90-190, School of Computer Science, Carnegie Mellon University (Nov 1990), `http://www.cs.cmu.edu/~scandal/papers/CMU-CS-90-190.html`

[4] Caprio, M.A., Fasano, P.J., Maris, P., McCoy, A.E.: Quadrupole moments and proton-neutron structure in p-shell mirror nuclei. Phys. Rev. C **104**(3), 034319 (2021). https://doi.org/10.1103/PhysRevC.104.034319

[5] Cook, B., Maris, P., Shao, M., et al.: High Performance Optimizations for Nuclear Physics Code MFDn on KNL. In: Taufer, M., Mohr, B., Kunkel, J.M. (eds.) High Performance Computing. pp. 366–377. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-46079-6_26

[6] Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling many-core performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing **74**(12), 3202 – 3216 (2014). https://doi.org/10.1016/j.jpdc.2014.07.003

[7] Epelbaum, E., et al.: Few- and many-nucleon systems with semilocal coordinate-space regularized chiral two- and three-body forces. Phys. Rev. C **99**(2), 024313 (2019). https://doi.org/10.1103/PhysRevC.99.024313

[8] Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: GPU gems, vol. 3, chap. 39, pp. 851–876. Addison-Wesley Professional (2007)

[9] Kim, J.Y., Kang, J.S., Joh, M.: GPU acceleration of MPAS microphysics WSM6 using OpenACC directives: Performance and verification. Computers & Geosciences **146**, 104627 (2021). https://doi.org/10.1016/j.cageo.2020.104627

[10] Maris, P., Caprio, M.A., Vary, J.P.: Emergence of rotational bands in ab initio no-core configuration interaction calculations of the Be isotopes. Phys. Rev. C **91**(1), 014310 (2015). https://doi.org/10.1103/PhysRevC.91.014310

[11] Maris, P., Vary, J.P., Navratil, P., et al.: Origin of the anomalous long lifetime of ^{14}C. Phys. Rev. Lett. **106**(20), 202502 (2011). https://doi.org/10.1103/PhysRevLett.106.202502

[12] Maris, P., Aktulga, H.M., Binder, S., et al.: No core CI calculations for light nuclei with chiral 2- and 3-body forces. Journal of Physics: Conference Series **454**, 012063 (2013). https://doi.org/10.1088/1742-6596/454/1/012063

[13] Maris, P., Vary, J.P.: *Ab initio* nuclear structure calculations of p-shell nuclei with JISP16. Int. J. Mod. Phys. E **22**, 1330016 (2013). https://doi.org/10.1142/S0218301313300166

[14] Maris, P., Yang, C., Oryspayev, D., Cook, B.: Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on GPUs using OpenACC (2021). http://arxiv.org/abs/2109.00485

[15] Shao, M., Aktulga, H., Yang, C., et al.: Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver. Computer Physics Communications **222**, 1–13 (2018). https://doi.org/10.1016/j.cpc.2017.09.004

[16] Sternberg, P., Ng, E.G., Yang, C., et al.: Accelerating configuration interaction calculations for nuclear structure. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. SC '08, IEEE Press (2008). https://doi.org/10.5555/1413370.1413386

[17] Suhonen, J.: From Nucleons to Nucleus: Concepts of Microscopic Nuclear Theory. Theoretical and Mathematical Physics, Springer, Berlin, Germany (2007). https://doi.org/10.1007/978-3-540-48861-3