Title

Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on GPUs using OpenACC

Authors

Maris, Pieter
Yang, Chao
Oryspayev, Dossay
et al.

Publication Date

2022-03-01

DOI

10.1016/j.jocs.2021.101554

Peer reviewed

Journal of Computational Science 59 (2022) 101554

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on GPUs using OpenACC

Pieter Maris [a,*], Chao Yang [b], Dossay Oryspayev [c], Brandon Cook [d]

[a] Department Of Physics and Astronomy, Iowa State University, Ames, IA, USA
[b] Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA
[c] Computational Science Initiative, Brookhaven National Laboratory, Upton, NY, USA
[d] National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA, USA

ARTICLE INFO

Keywords:
Iterative eigensolver
Nuclear configuration interaction
Large-scale parallel sparse matrix computation
GPU acceleration
OpenACC
Scalable performance

ABSTRACT

To accelerate the solution of large eigenvalue problems arising from many-body calculations in nuclear physics on distributed-memory parallel systems equipped with general-purpose Graphic Processing Units (GPUs), we modified a previously developed hybrid MPI/OpenMP implementation of an eigensolver written in Fortran 90 by using an OpenACC directives based programming model. Such an approach requires making minimal changes to the original code and enables a smooth migration of large-scale nuclear structure simulations from a distributed-memory many-core CPU system to a distributed GPU system. However, in order to make the OpenACC based eigensolver run efficiently on GPUs, we need to take into account the architectural differences between a many-core CPU and a GPU device. Consequently, the optimal way to insert OpenACC directives may be different from the original way of inserting OpenMP directives. We point out these differences in the implementation of sparse matrix–vector multiplications with multiple vectors, which constitutes the main cost of the eigensolver, as well as other differences in the preconditioning step and dense linear algebra operations. We compare the performance of the OpenACC based implementation executed on multiple GPUs with the performance on distributed-memory many-core CPUs, and demonstrate significant speedup achieved on GPUs compared to the on-node performance of a many-core CPU. We also show that the overall performance improvement of the eigensolver on multiple GPUs is more modest due to the communication overhead among different MPI ranks.

1. Introduction

One of the most challenging problems in the computational study of the structure of atomic nuclei is the solution of a large-scale eigenvalue problem

$$H\psi = \lambda\psi, \tag{1}$$

where H is an approximation to the many-body Hamiltonian associated with a target nucleus, λ is an eigenvalue of H and ψ is the corresponding eigenvector [1–3]. When H is approximated in a subspace spanned by Slater determinants of some single-particle basis functions, the finite-dimensional eigenvalue problem defined in (1) is often called a Configuration Interaction (CI) approximation, and the matrix H is often referred to as a CI Hamiltonian.

Due to the many-body nature of the eigenvalue problem, the dimension of the CI Hamiltonian H, which is a function of the number of nucleons and a basis truncation parameter, can become extremely large. However, H is generally very sparse. Therefore, iterative methods that can take advantage of an efficient implementation of a Sparse Matrix–Vector multiplication (SpMV) procedure are often preferred, especially when only a limited number of eigenpairs of H are needed [4, 5].

In nuclear physics, one is often interested in a few (five to ten) low-lying eigenpairs of H. In recent work [6], we have shown that these eigenvalues can be computed efficiently by using the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) method [7]. The advantages of the LOBPCG algorithm, which we will describe with some detail in the next section, over the widely used Lanczos algorithm [8], are

- The algorithm is a block method that allows us to multiply H with several vectors simultaneously. That is, instead of an SpMV, one performs an Sparse Matrix–Matrix multiplication (SpMM) of

* Corresponding author.
E-mail addresses: pmaris@iastate.edu (P. Maris), cyang@lbl.gov (C. Yang), doryspaye@bnl.gov (D. Oryspayev), bgcook@lbl.gov (B. Cook).

https://doi.org/10.1016/j.jocs.2021.101554
Received 22 June 2021; Received in revised form 24 October 2021; Accepted 28 December 2021
Available online 14 January 2022

a sparse square matrix on a tall skinny matrix at every iteration, which introduces an additional level of concurrency in the computation and enables us to exploit data locality better.
- The algorithm allows us to make effective use of approximations to several eigenvectors.
- The algorithm allows us to take advantage of a preconditioner that can be used to accelerate convergence.

We have implemented the LOBPCG algorithm in a nuclear structure computation software called Many-body Fermion Dynamics for nuclei (MFDn). The implementation uses a hybrid MPI/OpenMP parallelization scheme, and has been optimized [9] to achieve scalable performance for distributed-memory many-core systems such as the Cori KNL [10] system maintained at the National Energy Research Scientific Computing (NERSC) Center.

The increased availability of high performance computing platforms equipped with general purpose GPUs has motivated us to consider modifying the many-core CPU implementation of the LOBPCG algorithm, which is written in Fortran 90, to enable it to run efficiently on accelerator based systems. Instead of rewriting the code using, e.g., CUDA Fortran [11] or OpenCL [12] programming models, which would take a substantial amount of work, we decided to use the OpenACC directive based programming model, with cuBLAS and cuSOLVER instead of BLAS and LAPACK, in combination with CUDA-aware MPI [13,14], to modify the original code which is based on MPI+OpenMP. This approach has been used successfully in several other applications [15–19]. In this paper, we will show how to incorporate OpenACC and CuBLAS/CuSolver in the LOBPCG eigensolver. We will discuss the changes needed in order to achieve good performance on GPUs in the context of the LOBPCG algorithm. In particular, we will focus on the efficiency of the OpenACC implementation of the SpMM, which constitutes a significant portion of the overall computation, as well as the performance of the preconditioning step and other dense linear algebra operations required in the LOBPCG algorithm. Although our discussions focus on the LOBPCG algorithm, the same techniques we developed are applicable to other block-iterative eigensolvers such as the Davidson method [20] and the block-Lanczos algorithm [21].

The paper is organized as follows. In Section 2, we briefly review the main components of the LOBPCG algorithm. We discuss the hybrid MPI/OpenMP implementation of MFDn, and its key components with regard to the LOBPCG algorithm in Section 3. The modification of the code using OpenACC directives is presented in Section 4. Numerical examples that demonstrate the performance improvement on the GPUs achieved by using OpenACC directives are presented in Section 5. Further improvement of the existing approach is discussed in Section 6. We use the terminology GPU and device, as well as CPU and host, interchangeably throughout this manuscript. Preliminary results have been presented at the 2020 OpenACC Summit [22] and at the 2021 SIAM CSE conference [23].

2. The LOBPCG algorithm

We denote the eigenvalues of the $n \times n$ nuclear CI Hamiltonian H arranged in an increasing order by $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$. Their corresponding eigenvectors are denoted by $x_1, x_2, \ldots, x_n$. The first $k \le n$ eigenvectors and eigenvalues are given by $X = [x_1, x_2, \ldots, x_k]$ and $\Lambda = \mathrm{diag}\left\{\lambda_1, \lambda_2, \ldots, \lambda_k\right\}$, respectively, satisfying $HX = X\Lambda$. It is well known that X is the solution to the trace minimization problem

$$\min_{X^T X = I} \mathrm{trace}(X^T H X). \tag{2}$$

The LOBPCG algorithm developed by Knyazev [7] seeks to solve (2) by using the updating formula

$$X^{(i+1)} = X^{(i)} C_1^{(i+1)} + W^{(i)} C_2^{(i+1)} + P^{(i-1)} C_3^{(i+1)}, \tag{3}$$

Algorithm 1: The basic LOBPCG algorithm

Input: The sparse matrix H, an initial guess to the k desired eigenvectors $X^{(0)} \in \mathbb{R}^{n\times k}$, convergence tolerance (tol) and maximum number of iteration allowed ($maxiter$);

Output: (Λ, X), where Λ is a $k \times k$ diagonal matrix containing the desired eigenvalues, and $X \in \mathbb{R}^{n\times k}$ contains the corresponding eigenvector approximations;

1 $[C^{(1)}, \Theta^{(1)}] = \texttt{RayleighRitz}(H, X^{(0)})$;
2 $X^{(1)} = X^{(0)} C^{(1)}$;
3 $R^{(1)} = HX^{(1)} - X^{(1)}\Theta^{(1)}$;
4 $P^{(0)} = \emptyset$;
5 **do** $i = 1, 2, \ldots,$ *maxiter*
6 $\quad W^{(i)} = K^{-1} R^{(i)}$;
7 $\quad S^{(i)} = \left[X^{(i)}, W^{(i)}, P^{(i-1)}\right]$;
8 $\quad [C^{(i+1)}, \Theta^{(i+1)}] = \texttt{RayleighRitz}(H, S^{(i)})$;
9 $\quad X^{(i+1)} = S^{(i)} C^{(i+1)}$;
10 $\quad R^{(i+1)} = HX^{(i+1)} - X^{(i+1)}\Theta^{(i+1)}$;
11 $\quad P^{(i)} = W^{(i)} C_2^{(i+1)} + P^{(i-1)} C_3^{(i+1)}$;
12 $\quad$ Determine number of converged eigenpairs n_c by checking the relative norm of each column of $R^{(i+1)}$ using the convergence tolerance tol;
13 $\quad$ exit if $n_c \ge k$;
14 $\Lambda \leftarrow \Theta^{(i)}$; $X \leftarrow X^{(i)}$;

to approximate the eigenvector corresponding to the k leftmost eigenvalues of H, where $W^{(i)} \in \mathbb{R}^{n\times k}$ is the preconditioned gradient of the Lagrangian

$$\mathscr{L}(X, \Lambda) = \frac{1}{2}\,\mathrm{trace}(X^T H X) - \frac{1}{2}\,\mathrm{trace}\left[(X^T X - I)\Lambda\right] \tag{4}$$

associated with (2) at $X^{(i)}$, and $P^{(i-1)}$ is the search direction obtained in the $(i-1)$st iterate of the optimization procedure, and $C_1^{(i+1)}$, $C_2^{(i+1)}$, $C_3^{(i+1)}$ are a set of coefficient matrices of matching dimensions that are obtained by minimizing (4) within the subspace $S^{(i)}$ spanned by

$$S^{(i)} \equiv \begin{pmatrix} X^{(i)} & W^{(i)} & P^{(i-1)} \end{pmatrix}. \tag{5}$$

The preconditioned gradient $W^{(i)}$ can be computed as

$$W^{(i)} = K^{-1}(HX^{(i)} - X^{(i)}\Theta^{(i)})$$

where $\Theta^{(i)} = X^{(i)T} H X^{(i)}$, and K is a preconditioner that approximates H in some way. The subspace minimization problem that yields the coefficient matrix $C_1^{(i+1)}$, $C_2^{(i+1)}$, $C_3^{(i+1)}$, which are three block rows of a $3k \times k$ matrix $C^{(i+1)}$, can be solved as a generalized eigenvalue problem

$$\left(S^{(i)T} H S^{(i)}\right) C^{(i+1)} = \left(S^{(i)T} S^{(i)}\right) C^{(i+1)} D^{(i+1)}, \tag{6}$$

where $D^{(i+1)}$ is a $k\times k$ diagonal matrix containing k leftmost eigenvalues of the projected matrix pencil $\left(S^{(i)T} H S^{(i)}, S^{(i)T} S^{(i)}\right)$. The procedure that forms the projected matrices $S^{(i)T} H S^{(i)}$ and $S^{(i)T} S^{(i)}$ and solves the projected eigenvalue problem (6) is often referred to as the *Rayleigh–Ritz* procedure [24].

Note that the summation of the last two terms in (3) represents the search direction followed in the ith iteration, i.e.,

$$P^{(i+1)} = W^{(i)} C_2^{(i+1)} + P^{(i-1)} C_3^{(i+1)}. \tag{7}$$

Algorithm 1 outlines the main steps of the basic LOBPCG algorithm. The most computationally costly step of Algorithm 1 is the multiplication of H with a set of vectors. Although it may appear that we need to perform such calculations in steps 8 (where the projected matrix $S^{(i)T} H S^{(i)}$ is formed) and 10, the multiplication of H with $X^{(i)}$, $X^{(i+1)}$ and $P^{(i)}$ can be avoided because $HX^{(i+1)}$ and $HP^{(i)}$ satisfy the following recurrence relationships

$$HX^{(i+1)} = HX^{(i)} C_1^{(i+1)} + HW^{(i)} C_2^{(i+1)} + HP^{(i-1)} C_3^{(i+1)}, \tag{8}$$

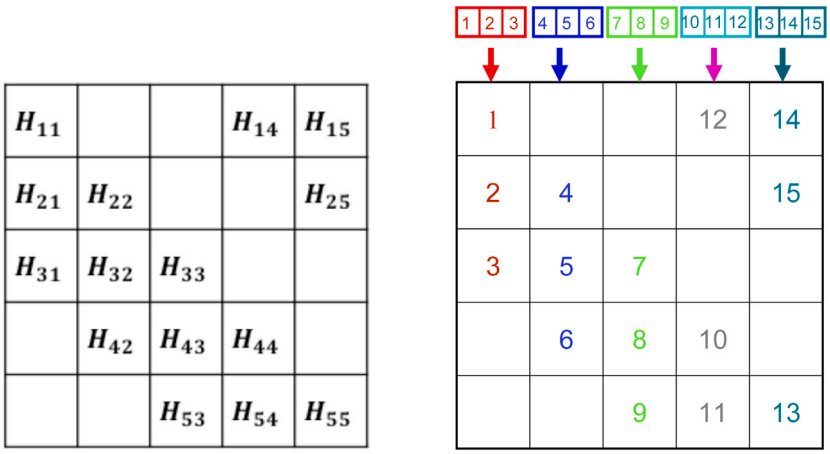


Fig. 1. The partition of H into 5×5 sub-matrices (left) and the distribution of H as well as the partition and distribution of a vector W among 15 processors (right). The integer label in the right panel represents the rank of the processes in the world communicator on which the vector segment is stored. Segments distributed among processes within the same column process group are drawn with the same color.

$$HP^{(i)} = HW^{(i)}C_2^{(i+1)} + HP^{(i-1)}C_3^{(i+1)}. \tag{9}$$

Therefore, the only SpMM we need to perform is $HW^{(i)}$. For the nuclear CI calculations of interest, the dimension n of the sparse symmetric matrix H can be several billion, whereas $W^{(i)}$ is a tall skinny $n \times k$ matrix with k typically of the order of 8 to 16.

3. Distributed-memory many-core CPU implementation

3.1. Data distribution

Because the dimension of the sparse matrix H for a nuclear CI calculation can be extremely large, it is partitioned and distributed among multiple processes [5]. Furthermore, in MFDn, we only store half of the symmetric matrix H using a special data distribution scheme described below.

We partition the rows and columns of H into $n_d \times n_d$ sub-matrices, where n_d is an odd integer. We then map the $n_d(n_d + 1)/2$ sub-matrices to $n_d(n_d + 1)/2$ MPI processes using a column major mapping scheme developed in [25], see Fig. 1. These processes are grouped into (sub)communicators based on the row and column indices of the sub-matrix they hold. There are n_d row communicators as well as n_d column communicators, each containing $(n_d + 1)/2$ MPI ranks.

Fig. 1 shows how H is partitioned into 5×5 sub-matrices and how the partitioned matrix is mapped to 15 processes labeled by rank 1 through 15. All sub-matrices with the same column (row) index belong to the same column (row) processor group, which is conveniently organized as a column (row) of the partitioned matrix as shown in the right panel of Fig. 1. Note that the $H_{1,4}$, $H_{1,5}$ and $H_{2,5}$ sub-matrices in the upper triangular part of H are the transposes of the $H_{4,1}$, $H_{5,1}$ and $H_{5,2}$ sub-matrices in the lower triangular part of H respectively. Storing and working with these sub-matrices instead of their lower triangular counterpart makes the mapping of the sub-matrices to MPI processes, and the corresponding column and row communicator groups, well load-balanced.

In addition to distributing the matrix H, we also distribute the vectors that will be multiplied by H. Each vector x is first partitioned into n_d sub-vectors conformal with the partitioning of H. Each of these sub-vectors is then further partitioned into $(n_d + 1)/2$ segments and distributed among the $(n_d + 1)/2$ processes within each column group. This partitioning of the vectors is illustrated in right panel of Fig. 1. A vector is drawn on top of the matrix to illustrate how sub-vectors are aligned with the sub-matrices of H. Each sub-vector is drawn with a distinct color, and further partitioned into 3 segments in the column group.

3.2. Parallel SpMM

A customized SpMM multiplication procedure has been developed to accommodate this particular data distribution scheme in the multiplication of the distributed H with a block of vectors W. In the following, we denote the ith block of sub-vectors of W by W_i. The distributed-memory parallel multiplication of H and W is carried out in MFDn as follows:

1. The segments of the sub-vector W_i, which are distributed among $(n_d + 1)/2$ processes within the ith column communicator, are gathered onto each process of that communicator using a call to `MPI_AllGatherV`.
2. The jth diagonal process broadcasts the gathered sub-vector $W_j = W_i$ across the jth row communicator in preparation for the distributed transpose SpMM computations, overlapping with the local SpMM using the local sub-matrix $H_{j,i}$, that is, $U_j = H_{j,i}W_i$ (see the top section of the code snippet in Fig. 2).
3. The output sub-vectors U_j are reduced along the jth row communicator onto the jth diagonal process, overlapping with the local transpose SpMM on the sub-vector W_j, that is, $U_i = H_{i,j}W_j$ (see the bottom section of the code snippet in Fig. 2).
4. The (reduced) output sub-vector U_j is added to the local output sub-vector U_i on the diagonal processes.
5. Finally, the sub-vectors U_i are reduced and scattered into $(n_d + 1)/2$ segments among the processors within the ith column communicator using a call to `MPI_ReduceScatter`.

Note that we separate the multiplication of $H_{i,j}W_j$ and $H_{i,j}^T W_i$ into two separate subroutines. This was done for two reasons: Firstly, in order to avoid race conditions (or private arrays with a reduction clause) for the OpenMP implementation of a combined local SpMM and transpose SpMM computation; and secondly, to overlap the communication along the row communicators with the local SpMM computations. This technique was proposed in [25] and has been shown to be very effective in both the OpenMP performance and in hiding most of the communication overhead. Furthermore, we use dynamic scheduling for the two OpenMP loops so that thread 0 can join the other threads once the MPI calls are completed, and to alleviate any load imbalance in the distribution of the nonzero matrix elements within an MPI rank. (The distribution of nonzero matrix elements among MPI ranks is very well balanced.) We will not elaborate on this technique in this paper since our focus is on the on-device parallelization of SpMM using OpenACC. However, in order to continue to be able to overlap communication and data transfers with local computation, we do keep this separation of the SpMM and the transpose SpMM.

```
!$omp parallel default(shared) private(tid, i, j, k, rbase, cbase, r, c, xcoef)
tid = omp_get_thread_num()
if (tid .eq. 0) then
   call MPI_Bcast(WT, nt, MPI_REAL4, 0, row_comm, ierr)
end if
!$omp do schedule(dynamic, 1)
do i = 1, nrowblks
   rbase = ...
   do j = 1, ncolblks
      cbase = ...
      do k = 1, nnz(i, j)
         r = rbase + ...
         c = cbase + ...
         U(r) = U(r) + xcoef * W(c)
      end do
   end do
end do
!$omp end do

if (tid .eq. 1) then
   if (idiag .gt. 0) then
      call MPI_Reduce(MPI_IN_PLACE, U, nt, MPI_REAL4, MPI_SUM, 0, row_comm, ierr)
   else
      call MPI_Reduce(U,            0, nt, MPI_REAL4, MPI_SUM, 0, row_comm, ierr)
   end if
end if

!$omp do schedule(dynamic, 1)
do j = 1, ncolblks
   cbase = ...
   do i = 1, nrowblks
      rbase = ...
      do k = 1, nnz(i, j)
         c = cbase + ...
         r = rbase + ...
         UT(c) = UT(c) + xcoef * WT(r)
      end do
   end do
end do
!$omp end do
!$omp end parallel
```

Fig. 2. Communication skeleton code for overlapping the MPI Bcast and Reduce calls with local SpMV computation on a CPU using OpenMP. See Section 3.3 and Fig. 3 below for more details on the local SpMM operations.

3.3. Local sparse SpMM

On the process that holds the submatrix $H_{i,j}$, we multiply the distributed local Hamiltonian sub-matrix $H_{i,j}$ and its transpose with two blocks of sub-vectors W_j and W_i. A special sparse matrix storage scheme called Compressed Sparse Block with coordinate (CSB_Coo) format [26,27] is used to make it easy to perform these two multiplications. The compressed sparse block storage is used to store the starting address of each nonzero block within $H_{i,j}$. Each nonzero matrix element within a nonzero block of $H_{i,j}$ is represented by its local row and column indices (coordinates) as well as its numerical value. By limiting the block sizes to 32,000 (though in practice, block sizes of the order of 4000 to 8000 are used), we can store these local indices as 2-byte integers, and achieve almost the same memory footprint as with the more conventional Compressed Sparse Row (or Column) format for sparse matrices.

Algorithm 2: Local SpMM

Input: Matrix $H_{i,j}$ in CSB_Coo format, a block of n_{vec} vectors W_j;
Output: $U_i = H_{i,j} W_j$;

do $i_b = 1, 2, \ldots, nb_{\text{row}}$
  do $j_b = 1, nb_{\text{col}}$
    if $H_{i_b,j_b} \neq \emptyset$ **then**
      for each nonzero element v of H_{i_b,j_b} **do**
        get the row and column indices (i_r, i_c) of v;
        do $k = 1, n_{\text{vec}}$
          $U_i(i_r, k) \leftarrow U_i(i_r, k) + v \cdot W_j(i_c, k)$;

The multiplication of $H_{i,j}$ with a vector W_j can be described by the procedure given in Algorithm 2. Although the multiplication of $H^T_{i,j} = H_{j,i}$ with W_i can be carried out in the same inner loop above, we implement the multiplication in a separate loop to allow for the local SpMM computation to be overlapped with the communication required to fetch W_i as we indicated earlier.

On a multi-core processor running with multiple threads, Algorithm 2 can be further parallelized by assigning each outer loop iterate (indexed by i_b) a single thread. This can be done by adding an OpenMP directive immediately before the first `do` loop. Fig. 3 shows the code snippet used in MFDn for performing the local SpMM with OpenMP. The input and output blocks of vectors are stored in the arrays `amp` and `Hamp` respectively. For the transpose SpMM using the same data layout, the two outermost loops are simply interchanged, as well as the arguments $(r + ii)$ and $(c + ii)$ in the innermost loop, thus avoiding any race conditions, and obtaining similar performance for the SpMM and transpose SpMM.

In addition to the OpenMP directive placed before the outermost loop to introduce thread-level concurrency, we also use the OpenMP SIMD directive before the innermost loop to vectorize the multiplication of $H_{i,j}$ with several vectors. To facilitate the `gather` and `scatter` of these blocks of vectors among different processes within the same column communicator, multiple vectors are stored in row major order, i.e., the elements in the first row of all vectors in the block are stored contiguously, followed by the elements in the second row etc. Such a storage scheme also enhances the data locality of the SpMM.

Finally, note that $H_{i,j}$ is a block sparse matrix, but each block itself is also sparse. Within each nonzero block, an additional level of blocking partitions each block into zero and nonzero tiles; and at the finest level, the nonzero tiles themselves are sparse as well. Fig. 4 illustrates the block and tile structure of a generic off-diagonal

```
!$omp parallel do default(shared) private(j, k, r, c, rbase, cbase, xcoef, ii)
do i = 1, nrowblks
   rbase = rowCSBoffset(i) - 1
   do j = 1, ncolblks
      if (CSBnnz(i,j) > 0) then
         cbase = colCSBoffset(j) - 1
         do k = CSBnnzoffset(i,j) + 1, CSBnnzoffset(i,j) + CSBnnz(i,j)
            c = numvecs*(cbase + Hcloc(k))
            r = numvecs*(rbase + Hrloc(k))
            xcoef = Hval(k)
            !$omp simd
            do ii = 1, numvecs
               Hamp(r+ii) = Hamp(r+ii) + xcoef * amp(c+ii)
            enddo
         enddo
      endif
   enddo
enddo
!$omp end parallel do
```

Fig. 3. Code snippet for local SpMM with OpenMP parallelization.

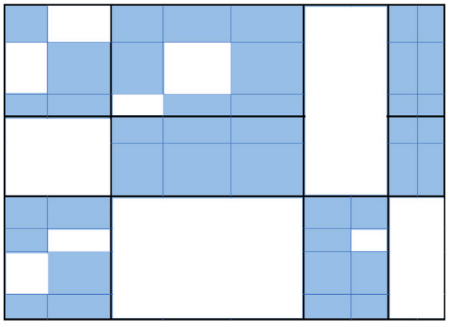


Fig. 4. The nonzero block and tile structure of a generic off-diagonal submatrix $H_{i,j}$. Each blue rectangle (with blue border lines) represents a nonzero tile, which is a sparse matrix. A larger rectangle with thick and black border lines represents a block. A nonzero block contains several nonzero tiles. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

submatrix $H_{i,j}$. Each blue rectangle (with blue border lines) represents a nonzero tile contained in a nonzero block, which is a larger rectangle with thick and black border lines. Note that $H_{i,j}$ may contain many zero blocks that are not stored. Within each nonzero block, the zero tiles are not stored. For a diagonal submatrix $H_{i,i}$, all diagonal blocks are nonzero. Within a diagonal block, all diagonal tiles are nonzero (and within a diagonal tile, all diagonal matrix elements are nonzero). These diagonal tiles are used in a preconditioner to be discussed next.

3.4. Preconditioner

MFDn uses a block diagonal preconditioner to accelerate the convergence of LOBPCG, i.e., the preconditioner K that appears in line 7 of Algorithm 1 is a block diagonal matrix of the form

$$K = \begin{pmatrix} K_1 & & & \\ & K_2 & & \\ & & \ddots & \\ & & & K_b \end{pmatrix}. \tag{10}$$

For K_j, we use the shifted diagonal tiles (introduced in the previous section) of the Hamiltonian, and b is the total number of diagonal tiles. The shift used in each diagonal tile is close to the approximation to one of the desired eigenvalue.

The shifted diagonal tiles K_j are distributed among all processes (not just the diagonal processes). Each process holds several diagonal tiles. The total number of rows in the diagonal tiles a process holds matches with the number of rows of the vector segment distributed to that process. We use the Formal Orthogonal Method (FOM) [28] to perform the preconditioning step in line 7 of Algorithm 1 by solving b indefinite linear systems with multiple right hand sides contained in $R^{(i)}$. The FOM algorithm constructs multiple Krylov subspaces associated with K_j and different columns of the corresponding $R^{(i)}$ block simultaneously, from which approximations to the corresponding block of $K^{-1}R^{(i)}$ can be extracted. Typically, three to five iterations of FOM is sufficient to achieve convergence acceleration. It is possible to use the minimal residual (MINRES) [29] algorithms to solve these linear systems also. Since only a few iterations are performed, there is not much difference between FOM and MINRES. Because both K_j and $R^{(i)}$ are distributed on all processes, the preconditioning step is carried out on all processes with no communication. Since each process contains a number of diagonal tiles K_j's that are independent, blocks of $K^{-1}R^{(i)}$ can be computed simultaneously by multiple threads. Within each tile, OpenMP SIMD directives are used to enable vectorization in the SpMMs within each FOM iteration.

3.5. Dense linear algebra operations

In addition to SpMM, which constitutes the largest computational and communication work load of the LOBPCG algorithm, there are several dense linear algebra operations that need to be implemented efficiently. The Rayleigh–Ritz procedure performed in Steps 2 and 9 of Algorithm 1 requires the following dense matrix–matrix multiplications

$$G = (X^{(0)})^T A X^{(0)},$$

and

$$G^{(i)} = (S^{(i)})^T A S^{(i)}, \quad O^{(i)} = (S^{(i)})^T S^{(i)}, \tag{11}$$

to be performed, where $AX^{(0)}$, $AS^{(i)}$ are assumed to have been computed and stored. In MFDn, the multiplication used to produce $G^{(i)}$ in (11) is split into six separate multiplications

$$G^{(i)} = \begin{pmatrix} (X^{(i)})^T A X^{(i)} & & \\ (W^{(i)})^T A X^{(i)} & (W^{(i)})^T A W^{(i-1)} & \\ (P^{(i-1)})^T A X^{(i)} & (P^{(i-1)})^T A W^{(i)} & (P^{(i-1)})^T A P^{(i-1)} \end{pmatrix} \tag{12}$$

shown as submatrices in the lower triangular part of $G^{(i)}$ in (12). The subblocks in the upper triangular part of $G^{(i)}$ (which are not shown above) are simply transposes of the corresponding subblocks in the lower triangular part. A similar split is used to obtain the overlap matrix $O^{(i)}$.

Because $X^{(i)}$, $W^{(i)}$, $S^{(i-1)}$, $AX^{(i)}$, $AW^{(i)}$, $AS^{(i-1)}$ are all distributed among different processes as shown in the right panel of Fig. 1, the multiplications performed above are carried out locally on each process using the portion of the vectors distributed to that process. The local dense matrix–matrix multiplication can be carried out using optimized BLAS subroutine GEMM. A global reduction and broadcast (through the use of `MPI_AllReduce`) allows us to obtain and replicate $G^{(i)}$ and $O^{(i)}$ on all processes.

A similar dense matrix–matrix multiplication is performed in the Cholesky QR procedure used to orthonormalize columns within a matrix block W. A Cholesky QR consists of the following sequence of operations.

$$B = W^T W \quad \text{(Matrix–matrix multiplication)}$$
$$B = R^T R \quad \text{(Cholesky factorization)}$$
$$W \leftarrow W R^{-1} \quad \text{(Triangular back substitution)}$$

We use the LAPACK library function `dsygv` to solve the generalized eigenvalue problem (6), and the function `dpotrf` to perform the Cholesky factorization of B. These computations can be replicated on all processes. The BLAS triangular solve function `trsm` is used to update W locally on all processes.

The eigenvectors in $C^{(i+1)}$ obtained from Step 8 of Algorithm 1 are used to update the approximate eigenvector $X^{(i)}$ and the search direction $P^{(i-1)}$ according to (3) and (7) respectively. These updates can be performed by using optimized `GEMM` also. All these `GEMM`s can be performed locally with no MPI communication because $C^{(i+1)}$ is replicated on all processes.

4. GPU implementation of LOBPCG using OpenACC

4.1. GPU and OpenACC

GPUs are throughput optimized processors with a Single Instruction, Multiple Thread (SIMT) programming model. These devices support massive parallelism and feature high on-device memory bandwidth making them ideal for certain data parallel workloads without complex control flow requirements. Additionally, in the low-conflict regime, performance of atomic operations is high compared to CPUs - a key feature to enable the sparse linear algebra kernels in MFDn.

MFDn is written in standard Fortran 90 which admits several options for a GPU implementation: Fortran with CUDA C/C++, CUDA Fortran, OpenCL, OpenMP target offload or OpenACC. OpenACC is an attractive choice for an initial port since it is a descriptive directives based approach with stable production support in major compilers. The directive approach preserves the flexibility of running on multi-core CPUs and avoids vendor specific language lock-in.

We use `!$acc declare create` in combination with `!$acc update` for all explicit data movement of (large) arrays. For the communication, we use CUDA-aware MPI [13,14], which implicitly handles any necessary data movement between device and host for the MPI communication. We adopt a descriptive approach to the use of directives for computation and exercise minimal control in order to allow the compiler and run time freedom to choose the best options. In practice, this means most loops are annotated only with `!$acc parallel`, `!$acc loop gang` and `!$acc loop vector` directives with few additional clauses.

4.2. SpMM

We assume a one-to-one mapping of MPI processes to GPUs and that each distributed local sparse Hamiltonian $H_{i,j}$ and all other needed data fits in device memory. CUDA-aware MPI is used to facilitate communication among devices so that we do not need to explicitly move data back and forth between hosts (where MPI communication is performed) and devices (where a majority of the computation is performed).

The easiest way to enable the local SpMM to be executed in parallel on a GPU is to replace the OpenMP parallel directives shown in the code snippets in Fig. 3 with the OpenACC `parallel loop` directive for the outer loop, and the OpenACC `loop vector` directive for the inner loop, as shown in Fig. 5. However, this naive port does not take full advantage of the much higher levels of thread concurrency on a GPU device. On a CPU, the number of threads on a processor is typically of the order of ten to a hundred. As a result, coarse grained concurrency achieved by decorating the outermost loop with an OpenMP parallel directive often works well. Thread overhead, which is higher on CPUs, can be minimized in this coarse grained approach. On devices, tens of thousands of lightweight threads can be simultaneously executed on multiple streaming multiprocessors (SMs) of a device. As a result, it is preferred to adopt a more fine-grained parallelization approach by generating many small tasks. Furthermore, on GPUs, one needs a larger vector length than on current CPUs to take full advantage of the device.

```
!$acc parallel loop default(present)
do i = 1, nrowblks
   rbase = rowCSBoffset(i) - 1
   do j = 1, ncolblks
      if (CSBnnz(i,j) > 0) then
         cbase = colCSBoffset(j) - 1
         do k = CSBnnzoffset(i,j) + 1, CSBnnzoffset(i,j) + CSBnnz(i,j)
            c = numvecs*(cbase + Hcloc(k))
            r = numvecs*(rbase + Hrloc(k))
            xcoef = Hval(k)
            !$acc loop vector
            do ii = 1, numvecs
               Hamp(r+ii) = Hamp(r+ii) + xcoef * amp(c+ii)
            enddo
         enddo
      endif
   enddo
enddo
!$acc end parallel loop
```

Fig. 5. Code snippet of a naive OpenACC port of the local SpMM.

Creating more smaller tasks can easily be realized by fusing the two outermost loops by using the OpenACC `collapse(2)` clause. In this approach, the multiplication of each nonzero block of a local Hamiltonian with a block of subvectors can be executed simultaneously. However, unlike the naive OpenACC port in which the update of the output vectors does not suffer from any write conflict, the accumulation of all products of local Hamiltonian blocks with distributed vector blocks may result in a race condition in which the same output vector block is being updated simultaneously with multiple blocked matrix–matrix products. To avoid this race condition, an OpenACC `atomic update` clause is added around the code segment in which `Hamp` is updated. Because the overhead associated with atomic updates on a GPU device is much lower than the OpenMP atomic updates performed on a CPU, having this type of synchronization does not lead to a significant increase in the total wall clock time, as we will show in the next section.

In addition to fusing the two outermost loops, we also fuse the two innermost loops to expose more parallelism at the inner loop level. The use of the OpenACC `loop vector` directive is similar in spirit to decorating the innermost loop in the OpenMP version of the local SpMM with the `simd` clause. However, the number of vectors we work with in the inner most loop is typically 4, 8 or 16, which is significantly less than the minimum number of vector threads on current GPUs, and we cannot take maximum advantage of coalesced memory access on the device. By vectorizing the two fused innermost loops, we expose more parallelism at the innermost loop level and create more opportunities for simultaneous access to multiple memory banks. The atomic updates introduced to avoid potential race conditions when we fused the two outermost loops also serve to avoid potential race conditions due to fusing the two innermost loops.

The code snippet that includes the use of loop fusion, atomic updates and vectorization in the local SpMM is shown in Fig. 6. As we will show in the next section, these slight modifications of the original OpenACC port of the local SpMM procedure results in significant performance improvement on GPUs.

Further improvement in the performance of the local SpMM procedure can be made by introducing an additional level of cache blocking within each nonzero block of the local Hamiltonian. This type of

```
!$acc parallel loop collapse(2) default(present)
do i = 1, nrowblks
   do j = 1, ncolblks
      if (CSBnnz(i,j) > 0) then
         cbase = colCSBoffset(j) - 1
         rbase = rowCSBoffset(i) - 1
         !$acc loop vector collapse(2)
         do k = CSBnnzoffset(i,j) + 1, CSBnnzoffset(i,j) + CSBnnz(i,j)
            do ii = 1, numvecs
               c = numvecs*(cbase + Hcloc(k)) + ii
               r = numvecs*(rbase + Hrloc(k)) + ii
               xcoef = Hval(k)
               !$acc atomic update
               Hamp(r) = Hamp(r) + xcoef * amp(c)
               !$acc end atomic
            enddo
         enddo
      endif
   enddo
enddo
!$acc end parallel loop
```

Fig. 6. A modified OpenACC parallelization of the local SpMM that fuses two outermost loops and two inner most loops. Atomic update is used to avoid write conflicts.

blocking can further improve data locality and memory access patterns. Fig. 7 shows how this additional level of blocking is implemented. The "`do k = ...` " loop in Fig. 6 is replaced with two nested loops in Fig. 7. The outer loop goes through a fixed number (defined by the variable `CacheSize` set at compile time) of nonzero matrix elements at a time. The multiplication of these nonzero elements with the matching segment of the vector blocks are implemented in the two inner most loops that are fused and vectorized. Also, the row and column indices as well as the nonzero elements of each chunk of size `CacheSize` are retrieved in advance and stored in gang-private arrays of length `CacheSize`. When the size of gang private arrays is known at compile time, the NVIDIA compiler will attempt to place those variables in CUDA shared memory — high performance memory shared by threads within a threadblock on NVIDIA GPUs. We note that it is possible to additionally specify this behavior through use of the `cached` directive if required. Placing these arrays in CUDA shared memory also confers an additional advantage — since multiple threads are reading the same values, these requests can be combined into a single broadcast transaction on recent NVIDIA GPUs.

Finally, we show in Fig. 8 how we use the OpenACC `async` clause to overlap the communication between MPI ranks with computation on the GPUs. We launch the local SpMM on the GPU with an `async(1)` clause, and continue on the host to perform an `MPI_Bcast` in preparation for the transpose SpMM. Once that `MPI_Bcast` is completed, we launch the transpose SpMM on the GPU with an `async(2)` clause. The `MPI_Reduce` operation on the output vector of the SpMM can be performed once the SpMM is completed, as indication by the OpenACC directive `$acc wait(1)` before the call to `MPI_Reduce`; and once that is completed, we have to wait until the transpose SpMM is completed as well, as indicated by the OpenACC directive `$acc wait(2)`.

In addition, there is also an `MPI_AllGatherV` before the main loop of the SpMM, which can be overlapped with local computation in a similar fashion; however, the final `MPI_ReduceScatter` cannot be overlapped. Furthermore, the actual summation of local output vectors is currently done on the host, and most likely by a single thread; on many-core CPUs we have implemented a user-defined multithreaded reduction operator which has the potential to significantly improve the performance of `MPI_Reduce` on large arrays. Looking ahead, we may need to implement something similar for use on GPUs.

4.3. Preconditioning

The parallelization of the preconditioning step of the LOBPCG algorithm on a GPU device is similar to the OpenMP parallelization of this step on a CPU. On each device, we loop over the diagonal

```
!$acc parallel loop collapse(2) default(present)            &
!$acc vector_length(VecLen) private(c_ar, r_ar, xcoef_ar)
do i = 1, nrowblks
   do j = 1, ncolblks
      if (CSBnnz(i,j) > 0) then
         cbase = colCSBoffset(j) - 1
         rbase = rowCSBoffset(i) - 1
         !
         do kv = 1, CSBnnz(i,j), CacheSize
            kvmax = min(CacheSize, CSBnnz(i,j) - kv + 1)
            kvoffset = CSBnnzoffset(i,j) + kv - 1
            !$acc loop vector
            do k = 1, kvmax
               c_ar(k) = numvecs*(cbase + Hcloc(kvoffset + k))
               r_ar(k) = numvecs*(rbase + Hrloc(kvoffset + k))
               xcoef_ar(k) = Hval(kvoffset + k)
            enddo
            !$acc loop vector collapse(2)
            do k = 1, kvmax
               do ii = 1, numvecs
                  c = c_ar(k)
                  r = r_ar(k)
                  xcoef = xcoef_ar(k)
                  !$acc atomic update
                  Hamp(r+ii) = Hamp(r+ii) + xcoef * amp(c+ii)
                  !$acc end atomic
               enddo
            enddo
         enddo
         !
      endif
   enddo
enddo
!$acc end parallel loop
```

Fig. 7. Further optimized version the OpenACC parallel local SpMM that introduces an additional level of blocking.

tiles generated on that device, and solve a linear system of equations with multiple right-hand sides by calling the subroutine `FOM_thread`. This loop is decorated with the OpenACC `parallel loop` clause, as is shown in the code snippet in Fig. 9. The amount of work performed in each loop iterate may be different because the size of these diagonal tiles vary from 1 to several thousand. Therefore, the concurrency achieved in this loop is a gang level concurrency that uses a group of workers to perform each task (defined by in the subroutine `FOM_thread`).

Within the subroutine `FOM_thread`, we use OpenACC `loop vector` clauses to solve each linear system with multiple right-hand sides iteratively using many vector threads. The main computation in each iterative FOM solver is the multiplication of a sparse matrix (diagonal tile) with a number of vectors. Because these tiles are often small, there is limited amount of concurrency we can explore in the multiplication of each (sparse) diagonal tile matrix with a block of vectors of length at most a few thousand. However, because these diagonal tiles are independent from each other, the multiplications of different diagonal tiles with different vectors can be executed simultaneously.

4.4. Other linear algebra operations

As we pointed out in Section 3.5, the LOBPCG algorithm contains a number of dense linear algebra operations. The GEMM operations required in the Rayleigh–Ritz procedure (lines 2 and 8–11 of Algorithm 1 or Eqs. (3)) can be performed on the device by using cuBLAS GEMM calls. In OpenACC, such calls can be made easily by simply including the `cublas` module and using the same arguments in, for example, `cublasdgemm`, as those used in a standard `dgemm` subroutine called on a CPU.

The dense Cholesky factorization and the solution of a dense generalized symmetric eigenvalue problem can be performed on the device by using cuSOLVER subroutines `cuSolverDndpotrf` and `cuSolverDnDsygvd`. Before calling these subroutines, one has to create `cuSolver` handles. A separate call to estimate buffer space required

```
!$acc parallel loop collapse(2) default(present) async(1)
do i = 1, nrowblks
   do j = 1, ncolblks
      do k = 1, nnz(i, j)
         r = ...
         c = ...
         !$acc atomic update
         U(r) = U(r) + xcoef * W(c)
         !$acc end atomic
      end do
   end do
end do
!$acc end parallel loop

!$acc host_data use_device(WT)
call MPI_Bcast(WT, nt, MPI_REAL4, 0, row_comm, ierr)
!$acc end host_data

!$acc parallel loop collapse(2) default(present) async(2)
do j = 1, ncolblks
   do i = 1, nrowblks
      do k = 1, nnz(i, j)
         c = ...
         r = ...
         !$acc atomic update
         UT(c) = UT(c) + xcoef * WT(r)
         !$acc end atomic
      end do
   end do
end do
!$acc end parallel loop

!$acc wait(1)

!$acc host_data use_device(U)
if (idiag .gt. 0) then
   call MPI_Reduce(MPI_IN_PLACE, U, nt, MPI_REAL4, MPI_SUM, 0, row_comm, ierr)
else
call MPI_Reduce(U,            0, nt, MPI_REAL4, MPI_SUM, 0, row_comm, ierr)
end if
!$acc end host_data

!$acc wait(2)
```

Fig. 8. Communication skeleton code for overlapping the MPI Bcast and Reduce calls with local SpMM computation on the device using OpenACC async clauses.

```
!$acc parallel loop default(present)                              &
!$acc          private(indx, offset, tiledim, H, r, beta, iter)    &
!$acc          create(W) copyin(shift)
do indx = 1, numDiagTiles
   offset = DiagTiles_Mstate_Offset(indx) - DiagTiles_Mstate_Offset(1)
   tiledim = DiagTiles_Mstate_Offset(indx+1) - DiagTiles_Mstate_Offset(indx)
   call FOM_thread(FOMit, nact, nblk, tiledim, indx, offset,       &
        shift, W(1,1,offset+1), H, r, beta, iter)
end do
!$acc end parallel loop
```

Fig. 9. OpenACC parallelization of the preconditioner.

in each of these cuSOLVER subroutines needs to be made as well. For example, the following calls

```
cus_status_ = cusolverDnCreate(cus_handle_)
!$acc host_data use_device(pamat,eigvals)
cus_status_ = cusolverDnDsygvd_bufferSize
  (cus_handle_,                                  &
  CUSOLVER_EIG_TYPE_1, CUSOLVER_EIG_MODE_VECTOR,
  CUBLAS_FILL_MODE_LOWER,                        &
  width, pamat, width, pamat(1+psize), width,
  eigvals, liwork)
!$acc end host_data
```

creates a handle `cus_status_` and returns the estimated work space requirement in `liwork`. Once the work space array `rwork` is properly allocated, we make the following call

```
cus_status_ = cusolverDnDsygvd
  ( cus_handle_,                          &
  CUSOLVER_EIG_TYPE_1, CUSOLVER_EIG_MODE_VECTOR,
  CUBLAS_FILL_MODE_LOWER,      &
  width, pamat, width, pamat(1+psize), width,
  eigvals, rwork, liwork, cus_info_(1))
```

to compute eigenvalues and eigenvectors of the matrices stored in `pmat`.

Note that, because the cuSOLVER subroutine will be called in every iteration of the LOBPCG algorithm and the work space required in each call does not change from one iteration to another, one can place the calls to create the handle and the work space once, before the main LOBPCG iteration loop starts. This significantly improves the performance by avoiding repeated calls to create and destroy handles, and repeated device memory allocation and deallocation at the appropriate work space size.

Finally, some of the LAPACK subroutines such as `dgelqf` and `dormlq` required to perform an LQ factorization of a non-square matrix $X = LQ$, where L is lower triangular and Q has orthonormal rows are not available in the cuSOLVER library. To overcome this difficulty, we modified the algorithm by replacing the LQ factorization of X by a QR factorization of $X^T = (LQ)^T = Q^T L^T = Q^T R$, where $R = L^T$ is upper triangular. This factorization can be performed by calling the subroutines `cusolverDnDgeqrf` and `cusolverDnDormqr` available in the cuSOLVER library.

5. Performance

In this section, we report the performance of the LOBPCG eigensolver in MFDn when it is executed on multiple GPUs and compare

Table 1
System specifications of Cori GPU.

Node CPU	2 × Intel Xeon (Skylake)
CPU Cores	20 @ 2.40 GHz
GPU	8 × 16 GB NVIDIA V100
CPU–GPU Interconnect	PCIe 3.0
Interconnect	4 dual-port Mellanox EDR

Table 2
The dimensions of sparse matrices used in the performance test and number of nonzero matrix elements in each matrix.

Test case	1	2	3	4	5	6
Matrix dimension ($\times 10^6$)	2.9	10.1	16.9	51.9	68.1	122.4
# of nonzero elements (nnz) ($\times 10^9$)	1.1	4.9	13.0	42.9	59.1	110.3

the GPU performance with the performance of the code on distributed memory many-core CPUs. Our experiments are carried out on the Cori GPU system [30] at the National Energy Research Scientific Computing (NERSC) Center. Each Cori GPU compute node consists of two Intel Skylake Xeon processors with 20 cores per processor, and 8 NVIDIA V100 GPUs. Table 1 gives detailed hardware and system specifications of the Cori GPU system.

We compare the performance of the OpenACC implementation of the LOBPCGsolver on the GPUs with the same implementation and same compiler on the CPUs. Specifically, we use the NVIDIA HPC SDK 20.11 compiler for both the CPU and GPU targets, with the following compiler options

- common flags: `-cpp -fast -mp=numa -Mlarge_arrays -Mipa -tp=skylake`
- flags for GPUs: `-acc -ta=tesla:cc70 -Mcuda -Mcudalib=cublas,cusolver`
- flags for CPUs: `-acc=multicore -Mmkl`

Note that for the GPUs we link with `cuBLAS` and `cuSOLVER`, whereas for the CPUs we link with the Intel Math Kernel Library (MKL) for the BLAS and LAPACK calls.

5.1. Test problems

We use several test problems that correspond to realistic Hamiltonian matrices of different nuclei (with up to 9 protons and neutrons) 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 2. As we see from this table the matrix dimension of the test problems ranges from 2.9×10^6 to 122.4×10^6.

The smallest problem (Test 1) can fit within one GPU on a single node of Cori GPU. As the problem size becomes larger, we need to distribute the Hamiltonian matrix on multiple GPUs and perform distributed-memory parallel computation on multiple MPI ranks. We use one GPU device per MPI process. Because each GPU on Cori GPU has 16 GB high bandwidth memory, we need to use a sufficient number of GPUs (and an appropriate number of nodes) to solve the larger problems. Table 3 shows the number of MPI processes used to solve each one of the six test problems in the following experiments. Note that the number of nonzero matrix elements per MPI process is kept approximately fixed at about 1×10^9 for each of the test problems, and thus the local compute load during the distributed SpMM should be approximately the same for each of them. For completeness we also list the actual amount of memory required to store the distributed Hamiltonian on each GPU device or MPI process in the CSB_Coo format described in Section 3.3.

We compute 5 eigenpairs for each test problem and set the number of vectors in $X^{(i)}$ (and $W^{(i)}$ as well as $P^{(i)}$) to 8, which is slightly larger than the number of desired eigenpairs. This is a common practice for improving the convergence rate of the LOBPCG algorithm as discussed in [6].

Table 3
The number of MPI processes and Cori GPU nodes used to solve each test problem, and the amount of memory required to hold the distributed Hamiltonian matrix per MPI process.

Test case	1	2	3	4	5	6
# of MPI processes	1	6	15	45	66	120
# of nodes	1	1	2	6	9	15
Max. # of nnz. per MPI process ($\times 10^9$)	1.04	1.07	1.03	1.05	0.97	0.98
Max. memory per MPI process (GB)	8.7	8.7	8.4	8.5	7.8	7.8

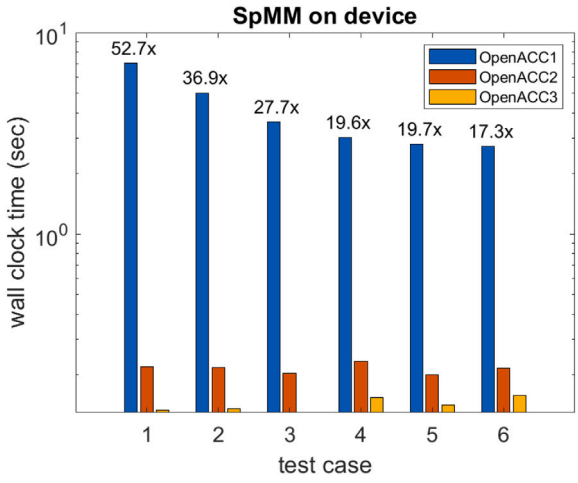


Fig. 10. A on-device performance comparison among three versions of the OpenACC implementation of the SpMM subroutine used in LOBPCG.

For each test problem, we use the same number of Cori GPU nodes (and MPI processes) to run both the CPU and GPU versions of the code, even though there are more CPU cores on each Cori GPU node than V100 GPU devices. In the CPU runs, we use 10 threads per MPI process, that is, we do use hyper-threading on the CPUs. The CPU performance of the OpenACC implementation is similar to that of the original OpenMP implementation when we use the (default) static scheduling policy. However, when we add the scheduling clause `schedule(dynamic, 1)` to the OpenMP do loops in the SpMM (see Fig. 2), the performance improves by a factor of 1.8 to 2.3 for these six test cases. Unfortunately, there is no equivalent clause in OpenACC.

5.2. SpMM performance

Since SpMM constitutes the dominant cost of the LOBPCG solver, we first examine the performance of this computational kernel.

In Fig. 10, we first compare the three versions of the OpenACC implementation of the SpMM described in Section 4 on the device. Recall that, in the first version, which we label by OpenACC1, we simply replace the OpenMP directives in the CPU version by an OpenACC parallel loop directive on the outermost loop, and an OpenACC loop vector directive on the innermost loop, shown in Fig. 5. In the second version (labeled as OpenACC2), we fuse the two outer loops of the SpMM algorithm to create more tasks, and use the gang-level concurrency to launch groups of tasks. Furthermore, we move the `vector` clause from the innermost loop to the two fused inner loops that performs a SpMM of a sparse matrix block with a block of subvectors to increase the level of vectorization and to create more fine-grained tasks, see Fig. 6. The `atomic update` clause is placed inside the innermost loop to ensure asynchronous updates are performed correctly. In the third version (labeled as OpenACC3), we create a level of cache blocking within the fused inner loops to optimize memory access patterns; the

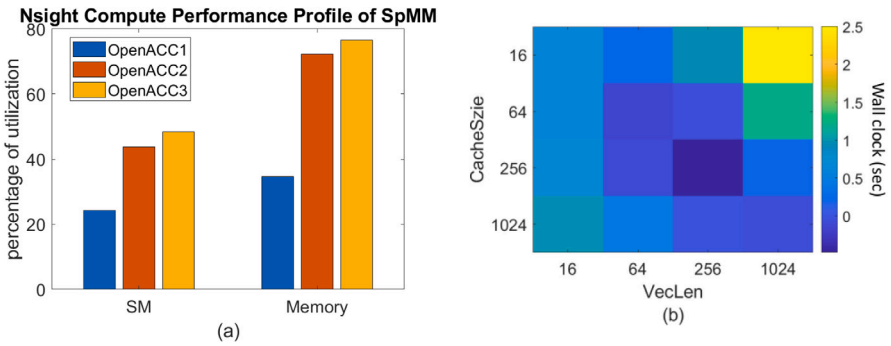


Fig. 11. Left panel: A Comparison of the SM and high bandwidth memory utilization among three versions of OpenACC implementation of the local SpMM algorithm. The blue bars correspond to the naive implementation shown in Fig. 5. The red bars correspond to the improved version shown in Fig. 6. The yellow bars correspond to the further improved version shown in Fig. 7. Right panel: The variation of SpMM OpenACC3 device wallclock time with respect to vectorization length and cache size shown as a heat map for test case 2. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

size of the additional private arrays is controlled by a parameter named `CacheSize` and we also set the vector length at compile time for the vectorization of these fused innermost loops, see Fig. 7. The timing comparison reported in Fig. 10 is for time per iteration spent on local SpMM computations on the device only. It excludes any host/device data transfer time as well as the MPI communication time.

We observe that by fusing the two outermost (row and column block) loops and moving the `vector` clause up from the innermost loop to the loop that goes through the matrix elements within each sparse block, and fuse it with the innermost loop, we can significantly improve the performance of SpMM in version 2. Furthermore, placing the `atomic update` clause within the fused innermost loops does not appear to incur a significant overhead. The low overhead in the OpenACC atomic update on GPUs is in sharp contrast to the high overhead in either OpenMP or OpenACC atomic update on CPUs.

Finally, by introducing an additional level of blocking within each sparse block, using the `CacheSize` parameter to control the inner block size, and tuning the vector length in that loop, we can gain additional efficiency. We report the speedup factor defined as the ratio of OpenACC1 wall clock time over OpenACC3 wall clock time over the top of the blue bar (OpenACC1 wall clock time.) The overall improvement in version 3 over version 1 can be as large as a speedup factor of 57 for the smallest test case and more than a speedup factor of 17 for the largest test case.

The advantage of the second and third versions of OpenACC parallelization of the SpMM algorithm for a GPU device can also be seen from a performance profile collected from the NVIDIA Nsight Compute [31] applied to analyze the on-device performance of the SpMM subroutine. In Fig. 11(a), we show the GPU utilization data on one representative MPI process for test problem 3, running on 15 GPUs on two nodes. Here we see that version 1 of the OpenACC implementation of the SpMM utilizes only ∼25% of the streaming multiprocessors (SM) and ∼35% of the high bandwidth memory. The percentages of SM and high bandwidth memory utilization increase to over 40% and 65% respectively after fusing the two outer loops, as well as fusing the two inner loops, as implemented in version 2 of the OpenACC implementation. Adding an additional level of cache blocking and controlling the vector length in the inner loop, as done in version 3 of the OpenACC implementation, further increases the SM utilization to close to 50%, and high bandwidth memory utilization to over 75%. A typical measured arithmetic intensity, the ratio of floating point instructions to global memory accesses, for version 3 the SpMM subroutine is 0.32. Comparing this to the peak memory bandwidth available on the device suggests the possibility of a further factor of two increase in performance. However, two factors limit further increases in performance. First, both the SpMM operation and the transpose operation are needed, which limits the amount of locality that can be expressed in either routine without the introduction of an additional sorting or ordering step. Second, the use of atomics does introduce some overhead to memory accesses, the number of which could potentially be reduced but not eliminated by employing some techniques such as those used in Ref. [32].

The performance of the OpenACC3 implementation is sensitive to the choice of maximum block size and the vector length specified in the OpenACC directives. Fig. 11(b) shows variation of on device SpMM wall clock time with respect to these parameters as a heat map. We observed that best timing in this case is achieved when both the block size and the vector length are set to 256. We use these values in all tests with OpenACC3 unless otherwise noted.

In Fig. 12, we compare the GPU performance of the OpenACC3 implementation of the SpMM with the CPU performance on the host using the OpenACC1 implementation without the need for atomics (which gives the best performance on the host, similar to the original OpenMP implementation). In the left panel we compare the compute time only, excluding all MPI related costs. We observe that the speedup we achieved on a GPU device ranges from about 15 for the smallest problem to more than 21 on the larger problems.

However, when the cost associated with implicit data device–host transfers and MPI communication are included in the comparison, the speedup of the GPU version is much less spectacular, as can be seen from the right panel of Fig. 12. Nonetheless we still achieve at least a factor of 2 speedup for the six test cases we are considering here. When the larger problems are distributed over many MPI processes, communication overhead starts to dominate the SpMM runtime and cut into the on-device gains achieved by GPUs. Indeed, including the time for data transfers and MPI communication the wall clock time for the largest of our test cases is approximately a factor of 10 larger than the local SpMM compute wall clock time on the device.

5.3. Preconditioner performance

The left panel of Fig. 13 shows how the GPU performance of the preconditioner compares with that of CPU's. As we indicated earlier, the main computation in the preconditioning step is the multiplication of a block diagonal matrix with a block of vectors. Each diagonal block is referred to as a diagonal tile. Both the diagonal tiles and vector blocks are distributed over all MPI processes. No communication is involved in this SpMM, but there is some MPI communication in the distributed dense linear algebra operations of the FOM algorithm — this is included in the time reported in Fig. 13. We observe speedup factors between 2.1 and 4.6 in the GPU version of the preconditioner. These speedup factors are typically less than those observed for the entire Hamiltonian SpMM (except for the larger cases). We attribute the lower speedup factors to the relative small size of many diagonal tiles and the variation in the tile sizes. Fig. 14 shows a histogram of the sizes of the diagonal tiles in the Hamiltonian of the first test problem. A majority of the tiles have

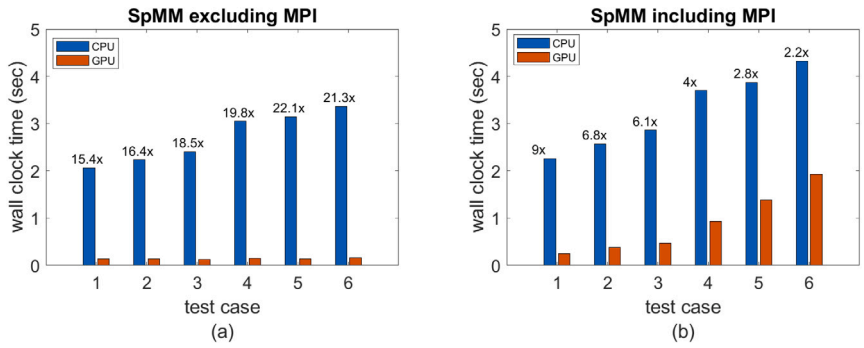


Fig. 12. Comparison between the performance of GPU/OpenACC3 implementation of the SpMM subroutine with CPU/OpenACC1 implementation (a) excluding and (b) including data transfers and MPI communication.

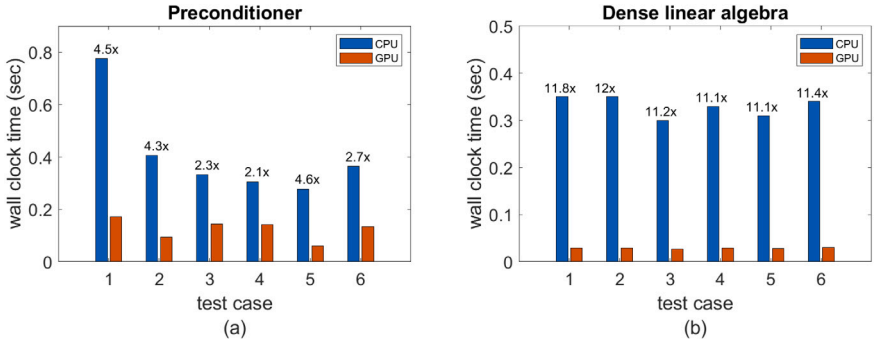


Fig. 13. Comparison between the performance of (a) the preconditioner and (b) the dense linear algebra on the GPU with that on the CPU.

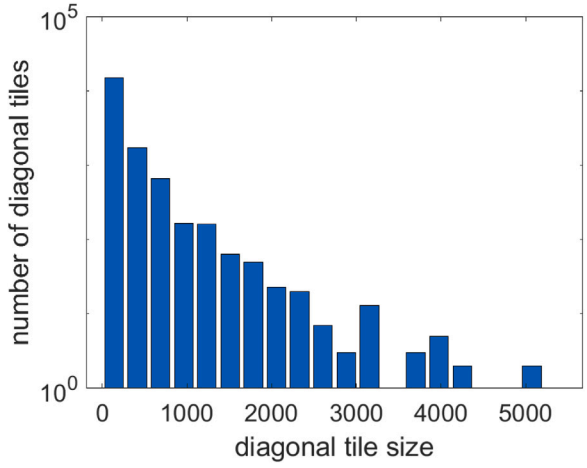


Fig. 14. A histogram of the sizes of the diagonal tiles of H in test problem 1.

sizes less than a few hundreds. The size of the largest tile is ∼ 6000. The small sizes of many of these diagonal tiles limit the amount of concurrency OpenACC can expose in the multiplication of a single tile with a block of 8 vectors.

5.4. Dense linear algebra performance

The right panel of Fig. 13 shows the speedup we achieved in the dense linear algebra operations in LOBPCG on GPU compared to CPU performance. On GPUs, we rely on `cuBLAS` and `cuSOLVER` to perform these operation. The multi-threaded MKL library is used for dense linear algebra computations on the CPUs. Overall we observe more than 11× speedup in all dense linear algebra subroutines used in the LOBPCG algorithm (excluding those used in the preconditioner).

5.5. Overall performance

In Fig. 15, we show how the overall performance of the LOBPCG eigensolver on GPUs compares with that on CPUs for all test problems, including all data transfers and MPI communication time in the iterative solver. For the comparison on the CPUs we use both the original OpenMP production version, and the OpenACC port compiled for multicore on the CPUs. In addition to comparing the total wall clock time spent in each LOBPCG iteration, we also show a breakdown of timing for three main components of the LOBPCG algorithm, i.e., SpMM, preconditioning, and dense linear algebra (again, including data transfer and MPI communication time). We observe that on GPUs, the amount of time spent in dense linear algebra is nearly negligible, whereas on CPUs, the dense linear algebra time is significant, both in the OpenMP and in the OpenACC implementation. On the other hand, the performance of the preconditioner is significantly better with the OpenMP implementation than with the OpenACC implementation. This is mostly due to the large variation in the tile size already discussed in Section 5.3 and shown in Fig. 14; in the OpenMP implementation we use the clause `schedule(guided)` for the main do loop in the preconditioner in order to deal with any load imbalance in that loop. In fact, the OpenMP implementation outperforms the GPU performance slightly for half these test cases — this is likely because of the large number of very small tiles used in the preconditioner.

Nevertheless, on both CPUs and GPUs, the wall clock time is dominated by time spent in SpMM for all except the smallest (single MPI rank) test case. Therefore, the overall speedup of the LOBPCG on GPUs is limited by the speedup of the SpMM calculation. For the OpenACC

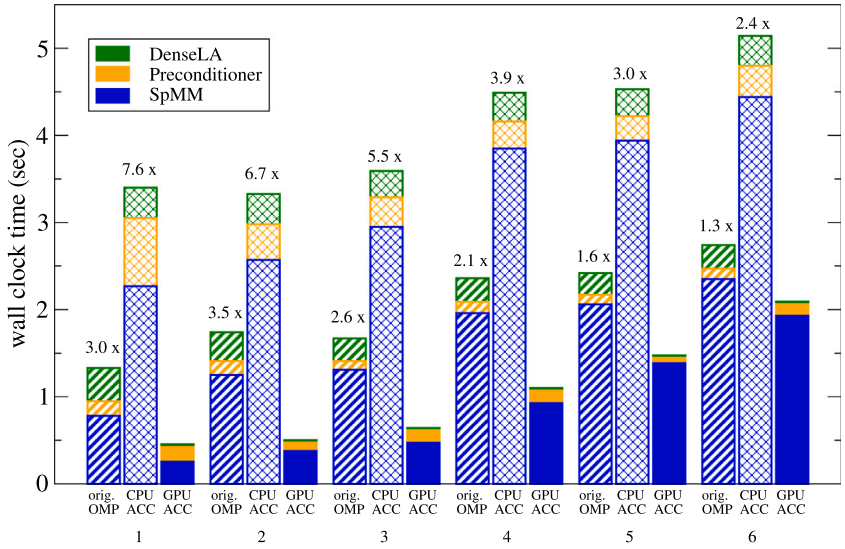


Fig. 15. Comparison between the overall performance of a LOBPCG iteration on GPUs (right bars), on CPUs using the OpenACC implementation developed for the GPUs (center bars), on CPUs using the original OpenMP implementation (left bars). The numbers above the left and center bars are the speedup achieved on the GPUs for each of the 6 test cases.

version, the speedup factor we observe ranges from 2.4 for the largest problem running on 120 MPI processes to 7.6 for the smallest problem running on a single MPI process; compared to the original OpenMP implementation, these speedup factors are between 1.3 for the largest case to 3.5 for the second-smallest case. Indeed, the difference in speedup between the OpenMP and the OpenACC implementations on the CPU is roughly a factor of two, coming from the difference in the SpMM implementation with the clause `schedule(dynamic, 1)` in the OpenMP version. Note that the increase of the wall clock time for the SpMM with the problem size is largely due to the increase in MPI communication and data transfer time indicating that reducing communication overhead will be crucial for large-scale computations utilizing thousands of GPUs. Of course, the MPI performance will also depend on the system; all our performance tests reported here were performed on the Cori GPU system, which is a small system intended for development of GPU-enabled codes, and not configured for optimal inter-node MPI communication.

6. Conclusion

In this paper, we showed how OpenACC can be used effectively to modify a previously developed MPI/OpenMP hybrid parallel iterative LOBPCG eigensolver for studying nuclear structure. The OpenACC directive-based modification enabled the solver to run efficiently on a distributed memory multi-GPU system. The use of OpenACC significantly reduces the amount of development effort for porting the existing eigensolver to an NVIDIA GPU system. However, due to the architecture difference between a general purpose GPU and a many-core CPU, care is needed to take advantage of the higher level of concurrency provided by SMs on a GPU, and high memory bandwidth of the device memory. In particular, we use loop fusion to reduce the granularity of the concurrency and increase the number of tasks that can be executed simultaneously. Although this approach leads to potential write conflicts, the extremely efficient atomic update in OpenACC on GPUs allows us to address this difficulty without much overhead.

We examined the parallelization of the SpMM operations which dominates the overall cost of the eigensolver, the preconditioning step, as well as some dense linear algebra operations required in the LOBPCG eigensolver. We demonstrated that significant speedup in SpMM can be achieved on a GPU device by using OpenACC. However, for large problems that must be distributed among hundreds of GPUs, the speedup factors are limited by data communication among different GPUs.

We found that the speedup in the preconditioning step, which consists of performing the multiplications of many small sparse matrices with several dense vectors (viewed as a dense matrix), is also somewhat limited. This limited speedup is likely due to the small dimensions of the matrices being multiplied. In future work we will investigate better strategies to improve the performance of the preconditioning step on GPUs.

With the OpenACC port described in this work, and implementations of the matrix initialization and of the evaluation of physical observables based on Ref. [33], we have an initial GPU-enabled port of our application MFDn for calculations of atomic nuclei on GPUs. Although the initial performance looks promising, further integration of the different parts of the code is needed in order to minimize data transfers, and to overlap data transfers, IO, and computation as much as possible. It remains to be seen what the performance will be of the fully ported code on a system like Perlmutter at NERSC [34], utilizing hundreds or even thousands of GPUs; at the moment we anticipate that MPI communication will become the bottleneck for runs at scale [35]. Assuming we can overcome this bottleneck, we envision that a GPU-enabled code for nuclear structure calculations will allow us to push the boundaries of our understanding of atomic nuclei utilizing the computing power of GPUs on current and future HPC platforms. Furthermore, we anticipate that the described LOBPCG solver implemented on GPUs will also be useful for other applications based on iterative sparse eigensolvers.

CRediT authorship contribution statement

Pieter Maris: Conceptualization, Software, Validation, Data collection, Writing and editing of the manuscript. **Chao Yang:** Conceptualization, Software, Validation, Data collection, Writing and editing of the manuscript. **Dossay Oryspayev:** Conceptualization, Software, Validation, Data collection, Writing and editing of the manuscript. **Brandon Cook:** Conceptualization, Software, Validation, Data collection, Writing and editing of the manuscript.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was initiated during an SDSC GPU Hackathon in 2020. We would like to thank Mathew Colgrove and Brent Leback from NVIDIA, as well as Ron Caplan from Predictive Sciences Inc. for their valuable suggestions on porting the LOBPCG solver to OpenACC. This work was supported in part by the U.S. Department of Energy (DOE) under grant No. DE-SC0018223 (SciDAC/NUCLEI) and the National Energy Research Scientific Computing Center (NERSC) Exascale Science Applications Program (NESAP). Computational resources were provided by NERSC, which is supported by the U. S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] B.R. Barrett, P. Navratil, J.P. Vary, Ab initio no core shell model, Prog. Part. Nucl. Phys. 69 (2013) 131–181, http://dx.doi.org/10.1016/j.ppnp.2012.10.003.

[2] J.P. Vary, R. Basili, W. Du, M. Lockner, P. Maris, D. Oryspayev, S. Pal, S. Sarker, H.M. Aktulga, E. Ng, M. Shao, C. Yang, Ab initio no core shell model with leadership-class supercomputers, in: A.M. Shirokov, A.I. Mazur (Eds.), Proc. Of Int. Conf. 'Nuclear Theory In The Supercomputing Era —2016', Khabarovsk, Russia, 2018, pp. 15–35, arXiv:1803.04101.

[3] M.A. Caprio, P.J. Fasano, P. Maris, A.E. McCoy, J.P. Vary, Probing ab initio emergence of nuclear rotation, Eur. Phys. J. A 56 (4) (2020) 120, http://dx.doi.org/10.1140/epja/s10050-020-00112-0, arXiv:1912.00083.

[4] Y. Saad, Numerical methods for large eigenvalue problems, Revised ed., in: Classics in Applied Mathematics, vol. 66, SIAM, Philadelphia, 2011.

[5] P. Sternberg, E.G. Ng, C. Yang, P. Maris, J.P. Vary, M. Sosonkina, H.V. Le, Accelerating configuration interaction calculations for nuclear structure, in: Proceedings Of The 2008 ACM/IEEE Conference On Supercomputing, SC08, IEEE, 2008, pp. 1–12, http://dx.doi.org/10.1109/SC.2008.5220090.

[6] M. Shao, H.M. Aktulga, C. Yang, E.G. Ng, P. Maris, J.P. Vary, Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver, Comput. Phys. Comm. 222 (2018) 1–13, http://dx.doi.org/10.1016/j.cpc.2017.09.004.

[7] A.V. Knyazev, Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method, SIAM J. Sci. Comput. 23 (2) (2001) 517–541, http://dx.doi.org/10.1137/S1064827500366124.

[8] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, J. Res. Nat'l Bur. Std. 45 (4) (1950) 255–282.

[9] B. Cook, P. Maris, M. Shao, N. Wichmann, M. Wagner, J. O'Neill, T. Phung, G. Bansal, High performance optimizations for nuclear physics code MFDn on KNL, in: M. Taufer, B. Mohr, J.M. Kunkel (Eds.), High Performance Computing, Springer International Publishing, Cham, 2016, pp. 366–377.

[10] NERSC, Cori KNL compute nodes, 2021, https://docs.nersc.gov/systems/cori/#knl-compute-nodes. (Accessed June 2021).

[11] G. Ruetsch, M. Fatica, CUDA Fortran For Scientists And Engineers: Best Practices For Efficient CUDA Fortran Programming, ELSEVIER SCIENCE & TECHNOLOGY, 2013.

[12] A. Munshi, B.R. Gaster, T.G. Mattson, J. Fung, D. Ginsburg, OpenCL Programming Guide, Addison–Wesley, 2012.

[13] H. Wang, S. Potluri, M. Luo, A. Singh, S. Sur, D. Panda, MVAPICH2-GPU: Optimized GPU to GPU communication for InfiniBand clusters, Comput. Sci. Res. Dev. 26 (2011) 257, http://dx.doi.org/10.1007/s00450-011-0171-3.

[14] K. Hamidouche, A.A. Awan, A. Venkatesh, D.K. Panda, CUDA M3: Designing efficient CUDA managed memory-aware MPI by exploiting GDR and IPC, in: 2016 IEEE 23rd International Conference On High Performance Computing, HiPC, 2016, pp. 52–61, http://dx.doi.org/10.1109/HiPC.2016.016.

[15] E. Otero, J. Gong, M. Min, P. Fischer, P. Schlatter, E. Laure, OpenACC acceleration for the PN–PN-2 algorithm in Nek5000, J. Parallel Distrib. Comput. 132 (2019) 69–78, http://dx.doi.org/10.1016/j.jpdc.2019.05.010.

[16] J.E. Stone, J.R. Perilla, C.K. Cassidy, K. Schulten, GPU-Accelerated molecular dynamics clustering analysis with OpenACC, in: R. Farber (Ed.), Parallel Programming With OpenACC, Morgan Kaufmann, Boston, 2017, pp. 215–240, http://dx.doi.org/10.1016/B978-0-12-410397-9.00011-1, (Chapter 11).

[17] A. Raj, S. Roy, N. Vydyanathar, B. Sharma, Acceleration of a 3D immersed boundary solver using OpenACC, in: 2018 IEEE 25th International Conference On High Performance Computing Workshops, HiPCW, 2018, pp. 65–73, http://dx.doi.org/10.1109/HiPCW.2018.8634138.

[18] H.W. Zhang, J. Zhu, Z.W. Ma, G.Y. Kan, X. Wang, W. Zhang, Acceleration of three-dimensional tokamak magnetohydrodynamical code with graphics processing unit and OpenACC heterogeneous parallel programming, Int. J. Comput. Fluid Dyn. 33 (10) (2019) 393–406, http://dx.doi.org/10.1080/10618562.2019.1683167.

[19] M. Martelli, C. Enderli, N. Gac, A. Vermesse, A. Mérigot, GPU acceleration: OpenACC for radar processing simulation, in: 2019 International Radar Conference (RADAR), 2019, pp. 1–6, http://dx.doi.org/10.1109/RADAR41533.2019.171296.

[20] E.R. Davidson, The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors large real-symmetric matrices, J. Comput. Phys. 17 (1975) 87–94.

[21] G.H. Golub, R. Underwood, The block Lanczos method for computing eigenvalues, in: J.R. Rice (Ed.), Mathematical Software III, 1977, pp. 361–377.

[22] D. Oryspayev, Experiences with porting a nuclear physics CI code to GPUs using OpenACC, 2020, https://www.openacc.org/events/openacc-summit-2020, presented at the annual OpenACC Summit.

[23] P. Maris, Eigensolvers for configuration interaction calculations in nuclear physics on GPUs, 2021, https://meetings.siam.org/sess/dsp_talk.cfm?p=109681, presented at the SIAM Conference on Computational Science and Engineering.

[24] B.N. Parlett, The Symmetric Eigenvalue Problem, Prentice Hall, Englewood Cliffs, 1980.

[25] H.M. Aktulga, C. Yang, E.G. Ng, P. Maris, J.P. Vary, Topology-aware mappings for large-scale eigenvalue problems, in: Euro-Par 2012 Parallel Processing, in: Lecture Notes in Computer Science, vol. 7484, Springer, 2012, pp. 830–842, http://dx.doi.org/10.1007/978-3-642-32820-6_82.

[26] H.M. Aktulga, A. Buluç, S. Williams, C. Yang, Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations, in: 2014 IEEE 28th International Parallel And Distributed Processing Symposium, IPDPS 2014, IEEE, 2014, pp. 1213–1222, http://dx.doi.org/10.1109/IPDPS.2014.125.

[27] H.M. Aktulga, C. Yang, E.G. Ng, P. Maris, J.P. Vary, Improving the scalability of a symmetric iterative eigensolver for multi-core platforms, Concurr. Comput. Prac. Exper. 26 (16) (2014) 2631–2651, http://dx.doi.org/10.1002/cpe.3129.

[28] Y. Saad, Iterative Methods For Sparse Linear Systems, Second ed., Society for Industrial and Applied Mathematics, 2003, http://dx.doi.org/10.1137/1.9780898718003.

[29] C.C. Paige, M.A. Saunders, Solution of sparse indefinite systems of linear equations, SIAM J. Numer. Anal. 12 (4) (1975) 617–629, http://dx.doi.org/10.1137/0712047.

[30] NERSC, Cori GPU nodes development system, 2021, https://docs-dev.nersc.gov/cgpu/hardware/.

[31] NVIDIA, NVIDIA Nsight compute, 2021, https://developer.nvidia.com/nsight-compute.

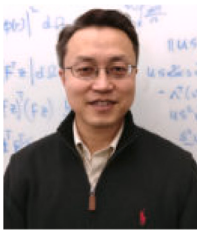
[32] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y.M. Tsai, E.S. Quintana-Ortí, Ginkgo: A modern linear operator algebra framework for high performance computing, 2020, arXiv:2006.16852.

[33] B. Cook, P. Fasano, P. Maris, C. Yang, D. Oryspayev, Accelerating quantum many-body configurationinteraction with directives, in: 2021 WACCPD, 2021, arXiv:2110.10765.

[34] NERSC, Perlmutter, 2021, https://docs.nersc.gov/systems/perlmutter/system_details/.

[35] M. Afibuzzaman, P. Maris, T. Groves, D. Oryspayev, B. Cook, C. Yang, H.M. Aktulga, Evaluation of the communication motif for a distributed eigensolver using the SST network simulation tool, in: 2020 IEEE/ACM Performance Modeling, Benchmarking And Simulation Of High Performance Computer Systems, PMBS, 2020, pp. 138–148, http://dx.doi.org/10.1109/PMBS51919.2020.00017.

Pieter Maris received his Ph.D. in theoretical physics from the University of Groningen in the Netherlands. After several postdoctoral positions in Japan and the USA, he is now a research professor at Iowa State University. He has more than 20 years experience using High-Performance Computing in his physics research. His current research areas are theoretical and computational nuclear physics, in particular ab initio calculations of the structure of atomic nuclei.



Chao Yang is a senior scientist in the Applied Mathematics and Computational Research Division at Lawrence Berkeley National Laboratory (LBNL). He received his Ph.D. from Rice University in 1998. He was a Householder fellow at the Oak Ridge National Laboratory from 1999 to 2000. He joined LBNL in 2000. His research interests include numerical linear algebra with applications in electronic structure calculations and quantum many-body problems, inverse problems, and high performance computing. He is a member of SIAM.

Dossay Oryspayev received his Ph.D. in Computer Engineering from Iowa State University. He is now with Computational Science Initiative of Brookhaven National Laboratory. His current research interests include a broad range of topics in high performance parallel computing area.

Brandon Cook received his Ph.D. in physics from Vanderbilt University. Before joining NERSC he was a postdoc at Oak Ridge National Laboratory where he developed and applied electronic structure methods to problems in material science. He now leads the simulations area of NERSC's application readiness program (NESAP) and works on understanding and analyzing performance on a system, workflow and application level, developing future benchmark suites, analyzing future architectures, developing productivity tools,and exploring future programming models.