The Journal of Chemical Physics

RESEARCH ARTICLE | MARCH 29 2024

Hybrid programming-model strategies for GPU offloading of electronic structure calculation kernels

Special Collection: Modular and Interoperable Software for Chemical Physics

Jean-Luc Fattebert ; Christian F. A. Negre ; Joshua Finkelstein ; Jamaludin Mohd-Yusof ; Daniel Osei-Kuffuor ; Michael E. Wall ; Yu Zhang ; Nicolas Bock ; Susan M. Mniszewski

J. Chem. Phys. 160, 122501 (2024)
https://doi.org/10.1063/5.0198797

AIP Publishing

The Journal of Chemical Physics SOFTWARE pubs.aip.org/aip/jcp

Hybrid programming-model strategies for GPU offloading of electronic structure calculation kernels

Cite as: J. Chem. Phys. 160, 122501 (2024); doi: 10.1063/5.0198797
Submitted: 19 January 2024 • Accepted: 10 March 2024 •
Published Online: 29 March 2024

Jean-Luc Fattebert,[1,a] Christian F. A. Negre,[2] Joshua Finkelstein,[2] Jamaludin Mohd-Yusof,[3] Daniel Osei-Kuffuor,[4] Michael E. Wall,[3] Yu Zhang,[2] Nicolas Bock,[5] and Susan M. Mniszewski[3]

AFFILIATIONS

[1] Computational Sciences and Engineering Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, USA
[2] Theoretical Division, Los Alamos National Laboratory, Los Alamos, New Mexico 87545, USA
[3] Computer, Computational, and Statistical Sciences Division, Los Alamos National Laboratory, Los Alamos, New Mexico 87545, USA
[4] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94550, USA
[5] Canonical USA Inc., Eatontown, New Jersey 07724, USA

Note: This paper is part of the JCP Special Topic on Modular and Interoperable Software for Chemical Physics.
[a] **Author to whom correspondence should be addressed:** fattebertj@ornl.gov

ABSTRACT

To address the challenge of performance portability and facilitate the implementation of electronic structure solvers, we developed the basic matrix library (BML) and Parallel, Rapid O(N), and Graph-based Recursive Electronic Structure Solver (PROGRESS) library. The BML implements linear algebra operations necessary for electronic structure kernels using a unified user interface for various matrix formats (dense and sparse) and architectures (CPUs and GPUs). Focusing on density functional theory and tight-binding models, PROGRESS implements several solvers for computing the single-particle density matrix and relies on BML. In this paper, we describe the general strategies used for these implementations on various computer architectures, using OpenMP target functionalities on GPUs, in conjunction with third-party libraries to handle performance critical numerical kernels. We demonstrate the portability of this approach and its performance in benchmark problems.

I. INTRODUCTION

Performance portability is a significant challenge for application programs that are run on modern HPC resources. For example, software solutions targeting portability, such as OpenMP, can sometimes have a hard time delivering performance, either due to the lack of maturity of compilers or the due to fine granularity of control needed for some specific kernels. On the other hand, writing kernels in a vendor-specific language targeting one specific GPU may not be portable and can lead to software maintenance difficulties. Recent trends in HPC only make this problem more acute as many leadership computing facilities have adopted hardware composed of heterogeneous compute nodes containing CPUs and GPUs, where a majority of the acceleration is provided by the GPUs.[1–3]

In practice, it is usually best to use existing libraries when possible if those libraries implement the numerical kernels one needs. For electronic structure applications, linear algebra libraries are the most common dependency. For dense linear algebra on CPUs, standard interfaces developed for BLAS[4–6] and LAPACK[7] have facilitated the use and development of these libraries and several well optimized solutions exist. On GPUs, the situation is more complicated. Vendors offer optimized implementations of BLAS and LAPACK in platform-specific libraries, such as cuBLAS, cuSolver (NVIDIA), rocBLAS, rocSolver (AMD), and MKL (Intel). There is, however, no common interface to these libraries. As a consequence, application codes need to have platform-specific wrappers around the functions that they intend to use. In addition, application developers need to understand the details of all these interfaces. There are

several reasons why the GPU situation is not as user friendly as it is in CPUs. First, there is the choice of having two possible locations for data arrays passed as arguments, either allocated to the host or the device, as well as for the return value, when there is one. Then, there is also the option of enabling several kernels to execute asynchronously on the device using, for instance, GPU streams. When writing platform-specific GPU kernels, there also can be extra kernel arguments depending on the architecture to ensure optimality such as, for instance, different hardware/run time parameters, different thread-block grid sizes, or user controlled cache (shared memory) sizes. As a result of these various issues and GPU technology changes, user interfaces are not as stable as one would like. For sparse linear algebra, the situation is also complicated. Unlike the dense format, for which there are not too many ways of laying out the data, there are several sparse formats, including compressed sparse row (CSR), compressed sparse column (CSC), coordinate list (COO), and ELLPACK just to mention a few. In addition, even for a given format such as CSR, there are variants as some libraries may or may not expect data in a row to be ordered by column indexes.

Some open source alternatives have emerged in recent years, some of which are targeting multiple architectures and thus facilitating portability of application codes using those. At the node level, the MAGMA library[8,9] offers the functionalities of BLAS and LAPACK on GPUs and is already fully functional in NVIDIA and AMD GPUs. The SLATE project is implementing a distributed and scalable dense linear algebra library for distributed memory accelerator-based computer systems, aiming to provide performance and portability to various types of hardware (CPUs, GPUs, and accelerators).[10] On the sparse matrix side, Ginkgo[11] is being actively developed and already offers a lot of functionalities on GPU architectures.

With the development of the Parallel, Rapid O(N), and Graph-based Recursive Electronic Structure Solver (PROGRESS) and the basic matrix library (BML), our goal is to facilitate the development of performant and portable electronic structure solvers by providing the necessary linear algebra tools in a hardware agnostic way. Due to the diversity available in accelerator hardware, we qualitatively refer to performance portability as achieving a consistent and reasonable performance across computer platforms. By providing several matrix formats, specifically a dense format and several sparse formats, BML facilitates the development of reduced complexity algorithms that can exploit any possible sparsity of the Hamiltonian and density matrices. We reported on this concept and the BML a few years ago in Ref. 12, with a focus on CPU implementations. In this paper, we extend this concept to implementations on GPUs, demonstrating some of these ideas on various hardware, such as NVIDIA V100, AMD MI250X, and Intel GPUs. We describe, in particular, the implementation model used to offload calculations to GPUs, using OpenMP in combination with third-party libraries. Electronic structure calculations are an important class of applications that require heavy use of linear algebra kernels. Here, electronic structure calculations broadly refer to the many ways of numerically evaluating the state of electrons in a physical system (molecule and periodic solid), as necessary, to derive other physical quantities of interest. In this paper, we will restrict our discussion to mean-field models, such as density functional theory (DFT) or tight-binding methods. Some algorithms and implementations discussed are targeting large scale simulations and make use of matrix sparsity to reduce computational complexity to $\mathcal{O}(N)$. Moreover, fast time-to-solution is also of high interest in the community, specifically to speedup wall-clock times in quantum molecular dynamics (QMD) and enable better modeling with longer time scales for medium-sized systems on the order of 1000 electrons.

The idea of isolating all the linear algebra operations of an electronic structure code into a separate library is a natural design choice and, at the same time, allows for multiple application codes to share this implementation. Several other research groups have made efforts toward identifying and isolating software libraries and have made them available to the community. The DBCSR library,[13,14] which the CP2K simulation package[15] relies on, is designed to efficiently perform sparse matrix–matrix multiplication, among other operations. It provides a distributed implementation using MPI and runs on NVIDIA and AMD GPUs via CUDA and HIP, respectively. The ELectronic Structure Infrastructure (ELSI) project[16] provides an open-source software interface to facilitate the implementation and optimal use of high-performance solver libraries electronic structure codes, including traditional eigensolvers, $\mathcal{O}(N)$ complexity algorithms, and other reduced complexity methods. The Electronic Structure Library (ESL)[17] is a community-maintained library of software specific to electronic structure simulations, which includes, among others, the ELSI library just mentioned. On the application side, an example of an electronic structure code that recently embraced this separation of operations and the use of more libraries is SIESTA.[18]

After introducing the PROGRESS library and BML in Sec. II, we discuss in Sec. III the specific problem of the electronic structure that these libraries are targeting: computing the single-particle density matrix (DM). In Sec. IV, we discuss some GPU-friendly algorithms implemented in PROGRESS as possible alternatives to a direct dense diagonalization. In Sec. V, we describe our general strategy to offload computational kernels to the GPU using OpenMP. We then describe some more specific strategies used for the dense matrix format in Sec. VI (using the MAGMA and MKL libraries) and sparse matrix format in Sec. VII (using the AMD rocSPARSE library and the hypre library). Finally, in Sec. VIII, we discuss distributed memory solvers that leverage the shared memory solvers discussed in Secs. IV–VII.

II. PROGRESS AND BML LIBRARIES

The basic matrix library (BML) is designed to implement the linear algebra operations necessary to implement matrix-based electronic solvers. Its purpose is to hide all the implementation details of numerically intensive kernels, including architecture-specific code and interfaces with third-party libraries, from the user.

BML is written in C. This facilitates interoperability with other languages, such as Fortran and C++. BML supports four datatypes for all operations: single precision, double precision, single complex, double complex. To avoid writing essentially the same code for all the data types, C macros are used extensively for data types and in function names, and the C preprocessor is used to generate specifics code associated with each data type. To avoid naming conflicts with other packages, all BML function names use a prefix `bml_`.

BML supports four different matrix formats: dense, ELLPACK-R, CSR, and ELLBLOCK. ELLPACK-R is a sparse format with a fixed

memory allocation, allowing a pre-determined maximum number M of non-zero elements per row.[19] It is less adaptable than CSR since a growing number of non-zeroes may reach the limit M and lead to failure. However, it has the performance advantage of avoiding a lot of memory allocation/deallocation during routine usage. ELLBLOCK is a generalization of ELLPACK-R, in which each element is a block.[20] Note that the BML CSR format implementation is not exactly what is usually referred to as CSR. In the BML, rows of data and column indexes are stored independent to each other instead of being packed together into a single array. This facilitates memory reallocation when the number of non-zeroes in a row changes. In addition, BML supports a memory distributed format, "distributed2d," which builds on top of the four non-distributed formats.

BML matrices are stored as a C struct, which includes, besides a pointer to the data storage, other parameters necessary to fully describe that matrix, such as its type (e.g. dense or ELLPACK-R), its data type (float, double, complex), and the number of rows in the matrix. An example for an ELLPACK-R matrix is shown in Listing 1.

LISTING 1. C struct used for ELLPACK-R matrix storage.

```
struct bml_matrix_ellpack_t
{
    /* matrix type identifier. */
    bml_matrix_type_t matrix_type;
    /* data type */
    bml_matrix_precision_t matrix_precision;
    /* number of rows. */
    int N;
    /* number of columns in storage */
    int M;
    /* matrix elements */
    void *value;
    /* column indexes */
    int *index;
    /* number of non zeros per row */
    int *nnz;
    ...
}
```

The PROGRESS [Parallel, Rapid O(N), and Graph-based Recursive Electronic Structure Solver] library is a collection of algorithms used in electronic structure calculations, with a focus on iterative solvers based on matrix polynomials. It is mostly written in FORTRAN but also offers a C-interface for routines expected to be called directly by an application code. More specifically, it implements several versions of the "second-order spectral projection" (SP2) DM solver,[21] a Chebyshev polynomial expansion of the DM,[22] as well as some iterative methods to compute the inverse square root of an overlap matrix as is often necessary in a non-orthogonal basis set or a non-orthogonal tight-binding model. All these implementations rely on BML matrices and their functionalities. They are mostly matrix format agnostic and available for all data types in BML. Several of these algorithms show an $\mathcal{O}(N)$ computational complexity with matrix size N when sparse matrix formats are used, and an appropriate threshold is used to discard small matrix elements. Figure 1 shows the software stack for a typical application

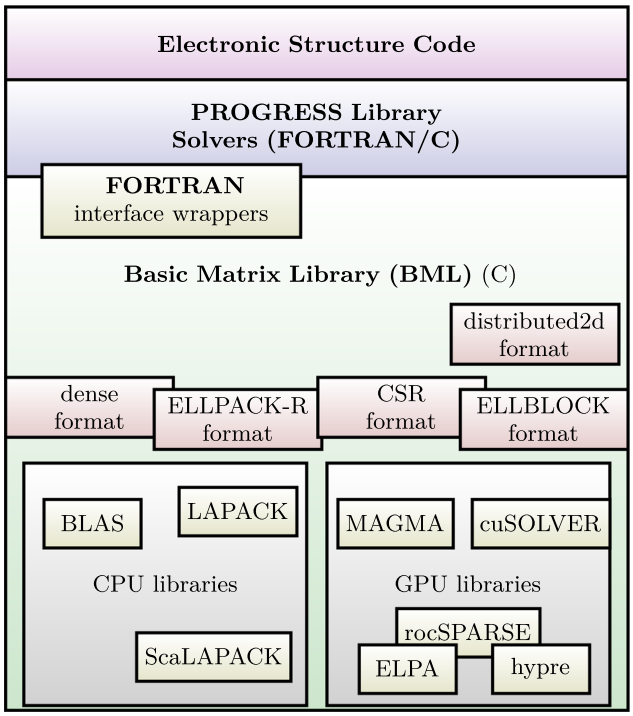

FIG. 1. Software stack showing the PROGRESS library and BML and their integration within an electronic structure application.

using PROGRESS and BML, including third-party dependencies. The PROGRESS library and BML are both open source, licensed under the BSD 3-clause license, and available on GitHub.[23,24]

To evaluate the solver's performance, we developed some benchmark drivers within PROGRESS. Our main benchmark test is based on a physical system, a small peptide chain solvated in water with periodic boundary conditions (Fig. 2). It consists of 523 atoms. From this atomic configuration, we build a tight-binding Hamiltonian represented by a matrix of size 1081×1081. To construct this Hamiltonian, we call a density functional based tight binding (DFTB)[25] Hamiltonian builder implemented in PROGRESS that uses DFTB parameters from the LATTE (Los Alamos Transferable Tight-binding for Energetics) code.[26] We build larger Hamiltonians by replicating this system by a factor of two or three in each direction. This gives us a series of Hamiltonians of increasing sizes to study computational cost, computational complexity, and parallel scaling. The sparsity of each DM for these benchmark problems is presented in Table I. This resulting "soft matter" system is what one would typically encounter in a biophysical molecular dynamics (MD) simulation.

PROGRESS and BML rely on the CMake build system.[28] To facilitate the development of these libraries and avoid breaking the code when changes are made by developers that are not familiar with all the functionalities and implementation details, an extensive suite of unit tests has been developed over time and continue to be enhanced. These tests cover all the matrix formats and matrix datatypes and are run through Ctest, the testing driver provided by CMake.

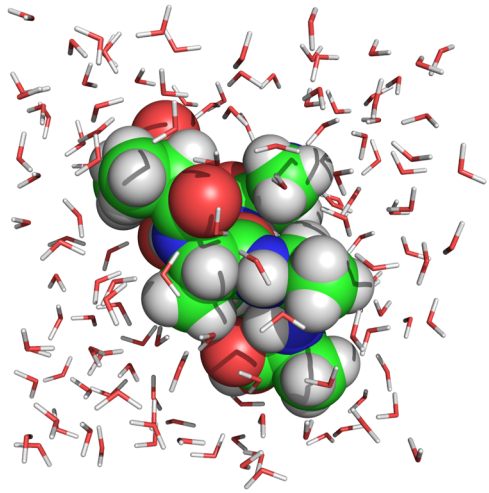

FIG. 2. Small peptide chain solvated in water, used for PROGRESS benchmarks. The figure was created using PyMOL.[27]

TABLE I. Percentage of zero-valued elements in DM in the PROGRESS benchmark problem, using a cutoff threshold of 10^{-6}.

Replicas	N	DM sparsity (%)
1 × 1× 1	1081	57.5
2 × 1× 1	2162	78.7
3 × 1× 1	3243	85.8
2 × 2× 1	4324	89.3
3 × 2× 1	6486	92.9
2 × 2× 2	8648	94.6
3 × 3× 1	9729	95.3
3 × 2× 2	12 972	96.4
3 × 3× 2	19 458	97.6
3 × 3× 3	29 187	98.4

III. SINGLE-PARTICLE DENSITY MATRIX AND THE ASSOCIATED EIGENVALUE PROBLEM

Given a symmetric (or Hermitian) $N \times N$ matrix H representing a Hamiltonian operator in a finite basis set, the task of computing the single-particle density matrix D in that same basis set can be accomplished by following a straightforward procedure, as follows:

1. Compute all the eigenvalues ϵ_i and eigenvectors $\mathbf{v}_i$ of H.
2. Given a chemical potential μ, the Fermi–Dirac distribution function is given by

$$f_\mu(\varepsilon) = \frac{1}{1 + \exp\left(\beta(\varepsilon - \mu)\right)}. \tag{1}$$

3. The single-particle density matrix is given by

$$D = V \begin{pmatrix} f_\mu(\varepsilon_1) & & & \\ & f_\mu(\varepsilon_2) & & \\ & & \cdots & \\ & & & f_\mu(\varepsilon_N) \end{pmatrix} V^T, \tag{2}$$

where V is the $N \times N$ matrix made of the ordered and orthonormal eigenvectors,

$$V = \left(\mathbf{v}_1 \mathbf{v}_2 \ldots \mathbf{v}_N\right).$$

Note that if there is a gap in the eigenvalue spectrum and no eigenvalue close to μ, $f_\mu(\varepsilon)$ takes either the value 1 ($\varepsilon < \mu$) or 0 ($\varepsilon > \mu$), provided that β is not too small. In this case, we have an insulator. The value μ in practice can be determined by setting an occupation (number of electrons) and finding μ, which results in this occupation through an iterative process.

In practice, the Hamiltonian matrix can come, for example, directly from a discretization of the problem in a small basis set (such as a set of Gaussian-shaped orbitals centered on the atoms), through a parameterized tight-binding (TB) approximation or from the projection of the Hamiltonian operator onto an auxiliary set of wave functions built iteratively during the search for the numerical solution in a larger *numerical* basis set (e.g. plane waves, finite elements, or finite difference cases). Depending on the discretization of the problem and the solver adopted, the size and degree of sparsity of this matrix H can vary significantly.

The matrix size also depends on the algorithm used to solve the electronic structure problem. When working with wave functions, N can be substantially smaller than in a TB or Gaussian-based approach. It can be as small as the number of occupied orbitals, as in a conjugate gradient solver,[29] in which case, no diagonalization is needed for the projected Hamiltonian (all states are fully occupied). However, wave function based solvers often use N larger than the number of occupied orbitals in order to speed up the solver or to solve for partial occupancy in metallic systems.[30] In addition, solvers such as block Davidson will involve solving a Rayleigh–Ritz problem often referred to, in the field of electronic structure, as subspace diagonalization for $2N \times 2N$ matrices.[31–33] A locally optimal block preconditioned conjugate gradient (LOBPCG) solver[34] will involve an even larger space, with a Rayleigh–Ritz procedure in the $3N \times 3N$ matrix space.

Note that for wave function based approaches, distribution of the wave function over nodes and cores can substantially reduce the time-to-solution. However, the Rayleigh–Ritz process used to compute the orbitals occupation does involve information from all the distributed parts, so that the resulting synchronization communication step is often the bottleneck in the strong scaling limit[35,36] and thus requires efficient algorithms to solve that problem.

IV. GPU FRIENDLY ALGORITHMS

Matrix multiplications have two advantageous properties when it comes to their implementations on GPUs: (i) very simple arithmetic operations and (ii) high arithmetic intensity (floating point operations per memory load). When compared to the operations involved in solving a dense eigenvalue problem on a GPU, the use

of standard dense diagonalization algorithms are typically not very efficient. Thus, solvers based on matrix–matrix multiplications are able to better utilize the massively parallel threads on a GPU and may offer a better performance.[20,37,38]

In the 1990s, significant development happened in the matrix–matrix multiplication based iterative solvers for the density matrix. Their primary purpose was to reduce algorithmic complexity from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$ by utilizing the sparsity in the Hamiltonian matrix and in functions of the Hamiltonian matrix (see Ref. 39 for a review). The key idea is to replace the diagonalization of the Hamiltonian matrix to evaluate the Fermi–Dirac function (see Sec. III) with a much cheaper polynomial approximation that one can easily apply to a Hamiltonian matrix. Often in these approaches, we define a shifted and rescaled Hamiltonian matrix $\widetilde{H}$ such that the eigenvalues of this matrix are all inside the interval $[a, b]$. A good polynomial approximation would then map the interval $[a, b]$ to a Fermi–Dirac function with the appropriate chemical potential μ so that

$$D \approx p_\mu(\widetilde{H}), \tag{3}$$

where the subscript of the polynomial p denotes the dependence on the chemical potential μ. Another type of approximate expansion is a recursive polynomial expansion,

$$D \approx p_n(p_{n-1}(\ldots p_2(p_1(\widetilde{H})))), \tag{4}$$

so that the polynomial p_μ is replaced by a composition of generally simpler or lower-order polynomials, $p_\mu = p_n \circ p_{n-1} \circ \cdots \circ p_2 \circ p_1$.

Although electronic structure problems can be quite large, domain scientists are often limited to solving problems of more modest sizes for which a very fast time-to-solution can be achieved. This is usually the case for quantum molecular dynamics, where an electronic structure problem needs to be solved at each time step to accurately compute atomic forces and propagate the atoms along the MD trajectories. In these more moderately sized problem cases, even matrix multiplications cannot always fully utilize the available resources on GPU devices and substantial portions of the GPU remain idle. Therefore, finding further parallelism in the evaluation of these polynomials is beneficial (see Sec. IV B).

A. SP2 solver

An example of a recursive expansion is the "second-order spectral projection" (SP2) algorithm,[21] as implemented in PROGRESS. In SP2, one starts with

$$D_0 \equiv \widetilde{H} = \frac{\varepsilon_N I - H}{\varepsilon_N - \varepsilon_1}, \tag{5}$$

the shifted and rescaled Hamiltonian, after which the density matrix is computed iteratively using the recursion

$$D_{m+1} = D_m^2 \tag{6}$$

if the trace of D_m is larger than the number of electrons and

$$D_{m+1} = 2D_m - D_m^2 \tag{7}$$

if the trace of D_m is smaller than the number of electrons. The density matrix D is then approximated by D_n for a sufficient number of iterations, n. Listing 2 shows a sketch of a Fortran code implementing an SP2 solver based on BML matrices and functionalities. We should emphasize that such an implementation is independent of the matrix format, the matrix data type, and the underlying computer architecture. Other variants of the SP2 algorithm have also been implemented in PROGRESS.

LISTING 2. Illustration of use case for BML matrices in the SP2 solver as implemented in the PROGRESS library.

```
!SP2 solver to compute DM d_bml for Hamiltonian h_bml
!using a cutoff threshold tau
subroutine prg_sp2(h_bml,d_bml,tau,...)
  use bml
  !Declare BML matrices ...
  type(bml_matrix_t),intent(in) :: h_bml
  type(bml_matrix_t),intent(inout) :: d_bml
  type(bml_matrix_t) :: x2_bml
  ...
  !initialize d_bml as shifted, rescaled Hamiltonian
  call bml_copy(h_bml, d_bml)
  call bml_add_identity(d_bml,-1.*emax)
  call bml_scale(-1.0/(emax-emin), d_bml)
  ...
  !SP2 loop
  do i=0,maxiter
    tr = bml_trace(d_bml)
    ! X2 <- X * X
    call bml_multiply_x2(d_bml,x2_bml,tau)
    if(tr-nel <= 0.) then
      call bml_add(2.,d_bml,-1.,x2_bml,tau)
    else
      call bml_copy(x2_bml, d_bml)
    end if
    !check for convergence
    ...
  end do
  call bml_deallocate(x2_bml)
end subroutine
```

We use SP2 in our PROGRESS benchmark, specifically to evaluate the performance of matrix-multiplication-based algorithms compared to dense diagonalization. For the bio-system benchmark described in Sec. II, SP2 converges to the specific tolerance used in that benchmark in 22 iterations, and its cost is typically dominated by the 22 matrix–matrix multiplications used in these iterations.

B. Chebyshev polynomial expansion

Recursive Fermi operator expansion techniques, such as SP2, have known challenges in the case of a vanishing electronic bandgap. In this case, a better technique is to use a serial Chebyshev expansion of the Fermi operator,[22]

$$f_\mu(H) \approx \sum_{n=0}^{L} c_n T_n(H), \tag{8}$$

where T_n is the nth Chebyshev polynomial and c_n is the nth Chebyshev expansion coefficient. Currently, in PROGRESS and in

some other codes,[40] this expansion is computed via the Chebyshev polynomial recursion property,

$$T_{n+1}(H) = 2HT_n(H) - T_{n-1}(H), \tag{9}$$

for $n > 0$, which results in a serialized summation—each term in the summation for f_μ needs to be computed in a sequence. Moreover, each recursion requires a single matrix multiplication so that for a Chebyshev expansion of polynomial order L, approximately L matrix multiplications are required. In other words, the number of matrix multiplications to approximate the Fermi operator and compute the density matrix scales linearly with the number of expansion terms.

In 1973, Paterson and Stockmeyer[41] showed that a generic polynomial P in powers of x of order L could be rewritten in such a way that only $\sim 2\sqrt{L}$ multiplications in x are needed to evaluate $P(x)$. Several decades later, Liang *et al.*[42,43] showed that this same idea could be applied directly to Chebyshev polynomial expansions of the Fermi–Dirac operator. Then, the Chebyshev expansion in Eq. (8) can be written as

$$\sum_{n=0}^{L} c_n T_n = \sum_{i=0}^{k-1} d_{i,0} T_i + T_k \left(\sum_{i=0}^{k-1} d_{i,1} T_i + T_k \left(\sum_{i=0}^{k-1} d_{i,2} T_i + \cdots + T_k \left(\sum_{i=0}^{k-1} d_{i,m-1} T_i \right) \cdots \right) \right), \tag{10}$$

for coefficients $d_{i,j}$, which can be determined from the Chebyshev coefficients c_n and positive integers k, m such that $L + 1 = km$. Not only did this algorithm replace a diagonalization with matrix–matrix multiplications, it also substantially reduced the *number* of multiplications needed to approximate the Fermi–Dirac operator for a given Chebyshev expansion size. Instead of the nearly L multiplications required for the serial recursion from Eq. (9), the number of multiplications now scaled with the square root of the polynomial order, $\sqrt{L}$. The large timing discrepancies between matrix–matrix multiplications and diagonalization previously mentioned then have an even more pronounced effect, making this approach very well-suited to modern GPU devices. The determination of the coefficients $d_{i,j}$ relies on the multiplicative recursion property of Chebyshev polynomials:[37] for n, m non-negative integers,

$$T_n T_m(H) = \frac{1}{2}(T_{n+m} + T_{|n-m|})(H). \tag{11}$$

Note that each sum, S_j, on the right-hand side of Eq. (10), where

$$S_j = \sum_{i=0}^{k-1} d_{i,j} T_i, \qquad 0 \le j \le m-1, \tag{12}$$

is completely independent of every other sum $S_{j'}$ for $j \neq j'$ once the first k Chebyshev polynomials $T_i, i = 0, \ldots, k$ are known. This introduces parallelism into Eq. (10) since each sum can be calculated concurrently with every other sum. In addition, the first k Chebyshev polynomials, T_1, T_2 up to T_k, can be calculated in a semi-parallel way too. The idea is to use the Chebyshev polynomial multiplication identity from Eq. (11) to calculate each $T_i, i = 2, \ldots, k$. Starting from T_0 and T_1, the second Chebyshev polynomial can be computed as

$$T_2 = 2T_1 T_1 - T_0, \tag{13}$$

and similarly, once T_0, T_1, and T_2 are known, the next two Chebyshev polynomials can be computed via

$$\begin{aligned} T_3 &= 2T_2 T_1 - T_1, \\ T_4 &= 2T_2 T_2 - T_0, \end{aligned} \tag{14}$$

using only previously computed T_i. Each Chebyshev polynomial on the left-hand side in Eq. (14) can, therefore, be computed in parallel. This parallelization on a single GPU is implemented using CUDA and HIP streams, on NVIDIA and AMD GPUs, respectively, through MAGMA queues. The combination of this parallelization along with the square root scaling Chebyshev expansion approach leads to orders-of-magnitude speedups over diagonalization when constructing the single-particle density matrix through a Chebyshev expansion. In Fig. 3, we show the speedup obtained on an AMD MI250X GPU.

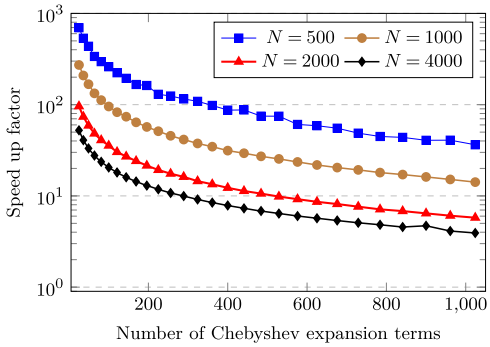

FIG. 3. Speedup over diagonalization of the density matrix construction (using a known chemical potential) on an AMD MI250X GPU with multiple GPU streams for $N = 500$ and $N = 1000$. Only a single stream is used for $N = 2000$ and N = 4000. The MAGMA library is used for diagonalization, which was found to be faster than the diagonalization routine in rocSolver.

Further details of this approach, and on how to compute the set of coefficients $\{d_{i,j}\}$, can be found in Ref. 37.

V. OFFLOADING WITH OPENMP

There are two components of GPU offloading, data movement and computation. Since most accelerated architectures have separate memory spaces for the host and device, with limited bandwidth between them, we attempt to minimize the data movement whenever possible. OpenMP[44,45] offers a portable, pragma-based framework for data movement between the host and device, as well as compute functionality on both. The OpenMP standard is supported by multiple compiler vendors,[46] with varying degrees of compatibility, especially with regard to offload support. GPU-offloading capabilities of OpenMP have been successfully used by various applications,[47–49] including some in the electronic structure community[50–52] and for large-scale sparse eigensolvers.[53] In this section, we outline the general strategy adopted in BML for using OpenMP to offload compute to accelerated devices.

A. Data allocation

The BML matrix format includes both the base matrix data (one or more data arrays, depending on the format) and certain metadata (e.g. matrix format, distribution mode, and local array bounds) in a C struct (see Listing 1). We only offload the data arrays to the device at allocation time, using persistent data allocation on the device. For example, in the ELLPACK-R format, we offload the values, index, and number of non-zero arrays using a combination of allocation and updates from the host to the device (see Listing 3). Subsequently, we assume that the data on the device are correct, and synchronization between the device and host is only performed when needed. We also modify the corresponding deallocation functions to ensure that the device-side memory is freed when host-side arrays are destroyed.

LISTING 3. GPU memory allocation and update for BML matrix data.

```
// allocate arrays on GPU
double* A_value=A->value;
#pragma omp target enter data map(alloc:A_value[:N*M])
int* A_index=A->index;
#pragma omp target enter data map(alloc:A_index[:N*M])
int* A_nnz=A->nnz;
#pragma omp target enter data map(alloc:A_nnz[:N])

// copy data from CPU to GPU
#pragma omp target update to(A_value[:N*M])
#pragma omp target update to(A_index[:N*M])
#pragma omp target update to(A_nnz[:N])
```

B. OpenMP compute

For simple computations or those computations that are not performance-critical, we can use OpenMP to perform device computations, for example, scaling an ELLPACK-R matrix (Listing 4).

LISTING 4. Example of a GPU-offloaded kernel using OpenMP directives.

```
  size_t MbyN = N * M;
#pragma omp target teams distribute parallel for
  for (size_t i = 0; i < MbyN; i++)
  {
    A_value[i] = scale * A_value[i];
  }
```

In practice, OpenMP may not offer the ability to generate kernels with optimal performance on some devices due to the inability to access certain fine-grained parallelisms that are only available via vendor-specific methods, such as CUDA.[54] For this reason, we generally utilize a mixture of OpenMP, for data movement and execution of simple or non-performance-critical compute with vendor-specific libraries (rocSparse and oneAPI) for performance-critical kernels (see Secs. VI B, VII A and VII B). Particularly for sparse matrices, we find that native OpenMP is unable to match the performance of vendor-optimized functions (see Fig. 6). Fortunately, it is relatively easy to pass data to vendor libraries through raw pointers in C as allocated by OpenMP on the GPU. In these cases, we utilize the OpenMP `use_device_ptr` functionality to perform compute on the previously allocated device-side data arrays, for example, Listing 5 shows the interface to call an eigensolver using Intel oneAPI MKL libraries.

LISTING 5. Code example illustrating a library call within an OpenMP region, in this case, calling MKL eigensolver.

```
#pragma omp target enter data \
  map(alloc:work[0:lwork])
#pragma omp target variant dispatch \
  use_device_ptr(evecs, evals, work)
  ssyev("V", "U", &N, evecs, &N, \
    evals, work, &lwork, &info);
```

In some cases, we have utilized entirely separate workflows, such as MAGMA, which encompass both data movement and compute bypassing OpenMP (see Sec. VI A).

For sparse matrix functionality, most vendor-supplied libraries only use the CSR format, both on CPU and accelerator. In contrast, BML uses the CSR (slightly modified), ELLPACK-R, and ELLBLOCK formats. In order to leverage the performance of the vendor-supplied libraries, we developed a set of functions to perform the appropriate data transformations. Here, we show a simple example of transforming data from a canonical CSR format to ELLPACK-R on the GPU (Listing 6);

LISTING 6. CSR to ELLPACK-R conversion using OpenMP target directives. Here, "ROWMAJOR" is a C macro that returns the 1-d index of an element (i,j) for a matrix of size A_M × A_N.

```
#pragma omp target teams distribute parallel for
  for (int i = 0; i < A_N; i++)
  {
    A_nnz[i] = csrRowPtr[i + 1] - csrRowPtr[i];
    for (int j = 0; j < A_nnz[i]; j++)
    {
      int idx = csrRowPtr[i] + j;
      A_value[ROWMAJOR(i, j, A_N, A_M)]
        = csrVal[idx];
      A_index[ROWMAJOR(i, j, A_N, A_M)]
        = csrColInd[idx];
    }
  }
```

Although these data transformations inevitably introduce some overhead, the cost is outweighed by the potential penalty of using sub-optimal compute for performance-critical functions, such as sparse matrix multiplication, in the native ELLPACK-R format using OpenMP.

VI. OFFLOADING DENSE LINEAR ALGEBRA SOLVERS

A. MAGMA on NVIDIA and AMD GPUs

The MAGMA library[8,55] offers most of the functionalities that we need for the dense matrix format. It essentially implements the whole set of functions typically available in BLAS and LAPACK for the GPU and more. It currently supports NVIDIA and AMD GPUs and is expected to support Intel GPUs in the near future. For offloading the dense matrix format to GPU in the BML, the choice was

made to rely heavily and, mostly, on MAGMA for memory allocation, data movement, and linear algebra operations on NVIDIA and AMD GPUs.

As an exception, we observed that the diagonalization function available in the cuSolver library "cusolverDnDsyevd" from NVIDIA significantly outperforms the MAGMA functions and have, thus, added an interface to it. Calling a cuSOLVER function from within a MAGMA code turns out to be easy since both codes work with simple C pointers, and a data array allocated by MAGMA can be directly used by any cuSolver function. The rocSolver library is the AMD equivalent of NVIDIA's cuSolver and is easy to integrate with MAGMA as well. However, we did not find the diagonalization function in that library to be more performant than the one in MAGMA.

In Fig. 4, we plot the time-to-solution for the SP2 solver (based on the "magma_dgemm" function) compared to dense diagonalization (using the "magma_dsyevd_gpu" function) for the PROGRESS 3D bio-benchmark, as measured on AMD MI250X GPU. SP2 significantly outperforms the dense diagonalization for all matrix sizes shown here (up to $N = 29\,187$) but more significantly for the smaller sizes.

Note that MAGMA provides a two-stage eigensolver that could potentially be faster than the divide and conquer version we are currently using. However, its current interface only supports data on the CPU, and thus, would require extra copies between the GPU and CPU in our implementation.

B. MKL on Intel GPUs

When offloading computation to Intel GPUs, we use the oneAPI MKL libraries.[56] At present, we have only offloaded the computations for BML dense format, but we anticipate that the methodology will be directly transferable to the appropriate sparse formats. The offloading of data to the Intel GPU is accomplished using the OpenMP functionality, as described in Sec. V. That is, we allocate the appropriate array, which in this case, is a single dense block, on the GPU when the BML matrix is initialized. Subsequently, the correct state of the matrix is assumed to be that on the GPU, and all computation is performed on the GPU when possible in order to minimize data movement and to maximally leverage the improved GPU compute relative to the CPU.

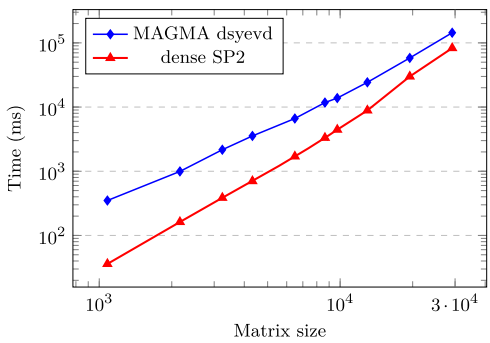

FIG. 4. Comparison of the time-to-solution between the SP2 solver and dense diagonalization using MAGMA functionalities on AMD MI250X GPU for the PROGRESS 3D bio-benchmark.

The Intel model for GPU offload is enabled via offloaded versions of their oneAPI MKL libraries. The call signature is largely the same as the corresponding CPU function, with the addition of an appropriate OpenMP pragma. For example, the call for matrix addition on CPU (computing $\alpha A + \beta B$) in double precision is given by using the BLAS C-interface (Listing 7):

LISTING 7. Computing $\alpha A + \beta B$ using the BLAS C-interface.

```
cblas_dscal(n, alpha, A->matrix, inc);
cblas_daxpy(n, beta, B->matrix, inc,
                A->matrix, inc);
```

In addition, the corresponding GPU offload is shown in Listing 8.

LISTING 8. Computing $\alpha A + \beta B$ using oneAPI MKL.

```
double* A_matrix=A->matrix;
double* B_matrix=B->matrix;
#pragma omp target variant dispatch \
    use_device_ptr(A_matrix)
  cblas_dscal (n, alpha, A_matrix, inc);
#pragma omp target variant dispatch \
    use_device_ptr(A_matrix, B_matrix)
  cblas_daxpy (n, beta, B_matrix, inc, A_matrix, inc);
```

Note that for complex data types, the MKL CBLAS calls on the GPU require an " & " in front of alpha and beta to get the address of a complex number. In addition, the `target variant dispatch` and `use_device_ptr` pragmas should also be noted. Otherwise, the call signature of the offloaded function appears identical to the corresponding host-side call.

With the appropriate functions offloaded, we can then compare the performance of the SP2 algorithm on the GPU with diagonalization, on both the CPU and GPU, for the PROGRESS 3D bio-benchmark. The resulting timings on the Sunspot testbed are shown in Fig. 5. Sunspot is a precursor to Aurora,[57] where each

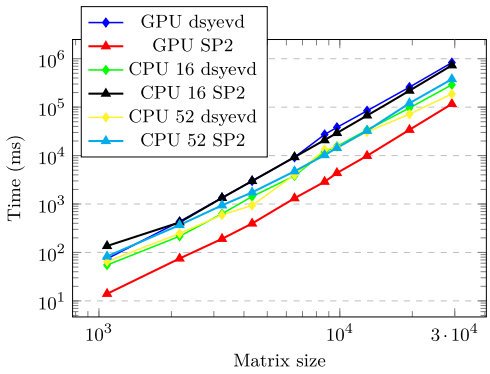

FIG. 5. PROGRESS benchmark timings measured on the Intel Sunspot system for dense matrices of sizes N = 1081 up to N = 29 187 using the oneAPI MKL library. Time is in milliseconds. The CPU runs used either 16 or 52 threads, as indicated in the legend. This work was performed on a pre-production supercomputer with early versions of the Aurora software development kit.

node consists of two Intel Xeon CPU Max Series (codename Sapphire Rapids or SPR) and six Intel Data Center GPU Max Series (codename Ponte Vecchio or PVC). Each Xeon has 52 physical cores supporting two hardware threads per core. We utilize 16 threads per process on the CPU, to simulate using 1/6 of the CPU available per GPU. We also repeated the test with 52 CPU threads, equivalent to a full socket, which shows some improvement at larger sizes. As seen in other systems, the SP2 solver is performing well compared to the dense diagonalization on the GPU. On the other hand, MKL "dsyevd" on CPU using 16 threads already outperforms the same function on the GPU.

Although the choice to implement the offloaded functions with (almost) identical signatures to the CPU version makes the implementation straightforward from a coding standpoint, the lack of maturity in the software stack is an issue. Functions are being ported to the GPU gradually, and compiler updates may break working builds so frequent regression testing is needed. Overall, this is to be expected with a software stack at this stage of development.

VII. LEVERAGING THIRD-PARTY LIBRARIES FOR $\mathcal{O}(N)$ SPARSE LINEAR ALGEBRA SOLVERS

When it comes to iterative sparse solvers, such as SP2, the key kernel needed is a sparse–sparse matrix multiply. Compared to others, such as sparse–dense matrix multiply, this function is not implemented in many libraries. Previously, threaded sparse matrix methods in BML were implemented using the ELLPACK-R sparse matrix format. Our initial approach to GPU acceleration used OpenMP offload. Unfortunately, this led to bottlenecks in matrix–matrix multiplications and additions—the performance of the offloaded versions of these functions was lacking. Thus, we switched to the use of third-party libraries for these numerical kernels.

A. AMD rocSparse library

As faster methods were needed for AMD GPUs to prepare for the availability of Frontier at the Oak Ridge Leadership Facility,[3] we sought to replace these functions with AMD rocSPARSE methods. The CSR data used by rocSPARSE are obtained using an ELLPACK-R to CSR translation code within BML (see Sec. V). Calls to rocSPARSE methods are then made within an omp target data region to enable host or device pointers to be passed to rocSPARSE methods as appropriate. The code block in Listing 9 illustrates the general approach of calling a rocSPARSE function using the GPU pointer to a BML data array (the approach is similar to the way the MKL methods are used on Intel GPUs, as described in Sec. VI B). In this code block, the function `f()` performs a computation on `A_matrix` on the GPU. The `use_device_ptr(A_matrix)` clause instructs the compiler to pass the GPU pointer for `A_matrix` to `f()`. The variable `A_N` is not included in the `use_device_ptr()` clause, indicating that the value on the host will be used.

LISTING 9. Using OpenMP offload for a rocSPARSE function "f" call and a BML matrix "A."

```
double* A_matrix=A->matrix;
#pragma omp target data use_device_ptr(A_matrix)
{
  f(..., A_matrix,A_N,...)
}
```

The most computationally intensive kernel in SP2 is the sparse matrix multiplication; therefore, we first developed an interface to use the `rocsparse_spgemm()` function inside BML for this kernel. Several other bottlenecks were subsequently identified. In particular, the addition of two matrices in BML was accelerated using the function `rocsparse_csrgeam()`.

As of at least ROCm 5.3, the rocSPARSE methods require sorted column indices. This is important for our applications as the ELLPACK-R format is unsorted, and matrices can become unsorted during the course of calculations. For example, computing the matrix transpose leads to unsorted column indices. Moreover, even though the rocSPARSE methods require sorted matrices on input, they can produce unsorted matrices on output. Therefore, we changed the BML rocSPARSE code to sort the matrices as needed. This was accomplished using the rocSPARSE `rocsparse_csrsort` function. In addition, to ensure that the resulting sparse matrices satisfy the thresholding criterion for including matrix elements, our implementation uses the rocSPARSE function `rocsparse_xprune_csr2csr`.

Figure 6 compares the performance of the SP2 solver for the PROGRESS 3D bio-benchmark using the rocSPARSE (red triangles) vs MAGMA (blue diamonds) solvers. The builds used the Cray CCE compilers version 15 and AMD ROCm version 5.1. The rocSPARSE method shows approximate linear scaling, as expected for the sparse SP2 density matrix algorithms. Due to the approximate linear scaling, the rocSPARSE timings for large matrix sizes are smaller than the MAGMA timings (up to more than 10x). Two OpenMP offload timings for small matrices are shown for comparison (orange squares shown in Fig. 6). These data points show that the time required for the density matrix build using the rocSPARSE method is orders of magnitude smaller than that achieved for the OpenMP offload code. Overall, the figure shows the substantial performance increase achieved in GPU when taking advantage of sparsity using rocSPARSE compared to the dense format using MAGMA for large matrix sizes. It also shows that rocSPARSE overcomes the performance limitations of the OpenMP offload methods.

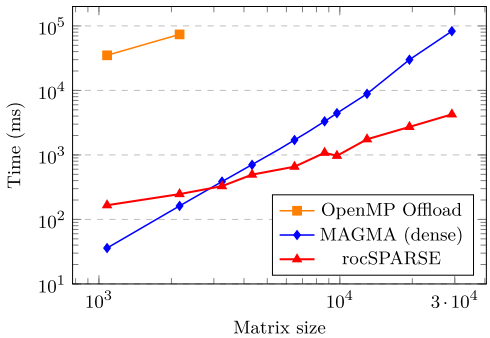

FIG. 6. Comparison of the time-to-solution of the SP2 solver between sparse matrices using rocSPARSE and dense matrices using MAGMA on AMD MI250X GPU for the PROGRESS 3D bio-benchmark. OpenMP offload timings for small matrix sizes are also shown for comparison.

B. Hypre library

As mentioned above, functionalities for multiplying two sparse matrices by each other are not found in many linear algebra libraries. The focus is often on solving sparse linear systems using iterative solvers, where multiplying a sparse matrix by a vector is the key ingredient. That being said, electronic structure solvers are not the only ones using sparse–sparse matrix multiplications. Another area where those are heavily used is in the algebraic multigrid community. There, coarse grid operators are often computed as the product $A_L = RA_lP$, where A_l is the discretized operator at the fine level, A_L is the discretized operator at the coarse level, and R and P are restriction and prolongation operators, respectively, which are used to coarsen or refine the data between levels. All these operators are represented as sparse matrices, and thus, the computation of A_L is typically the result of two consecutive sparse–sparse matrix multiplications.

The hypre linear solver library[58,59] is a well-known open-source library with scalable algebraic multigrid solver capabilities, among other solvers. The sparse matrix–matrix multiplication routines in hypre provide access to both vendor library routines and internal algorithms independent of the vendor. These internal algorithms are ported to the HPC hardware with NVIDIA, AMD, and Intel GPUs based on their respective native programming paradigms. Thus, through the same interface to hypre, the BML could access and utilize this performance critical sparse matrix functionality on different HPC platforms, leading to a performant and portable alternative to vendor-specific libraries. Additionally, leveraging open-source libraries, such as hypre, provides security through access to software capabilities and personnel with technical expertise and a shared interest in addressing software stack issues and challenges in emerging HPC hardware.

Accessing sparse matrix–matrix multiplication routines through hypre follows a similar approach to integrating BML with vendor libraries. First, matrix data are converted from the ELLPACK-R format to the standard CSR format on the device. The internal CSR matrix data structure of hypre uses raw C array pointers to store the matrix data. This interface makes it convenient to directly pass device pointers to standard CSR data, allocated by OpenMP, into hypre's internal CSR matrix data structure without incurring additional overhead. Next, with the data on the device, the sparse matrix–matrix multiplication is performed using the functionality provided by hypre. Finally, the result is converted back from CSR to ELLPACK-R using the same approach as for vendor libraries. Listing 10 provides a code snippet of how the integration with hypre is realized. Note that unlike rocSPARSE, hypre's interface does not put any requirement on the order of the elements in a CSR row.

LISTING 10. Code snippet showing the use of hypre for sparse matrix–matrix multiplication in the BML. The assignment operations in the round brackets show how data pointers are passed to hypre's internal data structure on the device.

```
    /* create hypre csr matrix */
    matA = hypre_CSRMatrixCreate( A_N,A_N,nnzA );
    matB = hypre_CSRMatrixCreate( B_N,B_N,nnzB );
#pragma omp target data use_device_ptr \
    (csrRowPtrA,csrColIndA,csrValA, \
    csrRowPtrB,csrColIndB,csrValB)
    {
       hypre_CSRMatrixI(matA) = csrRowPtrA;
       hypre_CSRMatrixJ(matA) = csrColIndA;
       hypre_CSRMatrixData(matA) = csrValA;

       hypre_CSRMatrixI(matB) = csrRowPtrB;
       hypre_CSRMatrixJ(matB) = csrColIndB;
       hypre_CSRMatrixData(matB) = csrValB;
    }
    /* perform matrix multiplication */
    matC  = hypre_CSRMatrixMultiplyDevice(matA, matB);
```

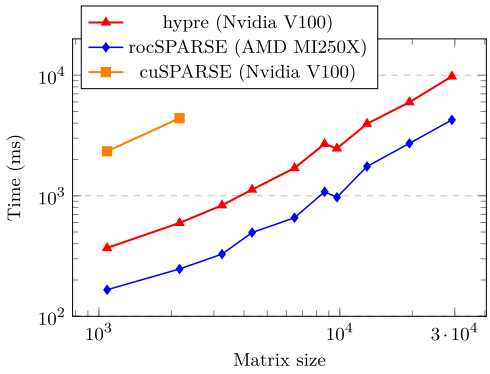

FIG. 7. Time-to-solution of the sparse $\mathcal{O}(N)$ SP2 solver using the hypre library on NVIDIA V100 GPU for the PROGRESS 3D bio-benchmark. For comparison, we also plot two data points when using the NVIDIA cuSparse library on the same V100 GPU, as well as the results using the rocSparse SP2 library on the AMD MI250X GPU.

Initial challenges to this effort came from having a consistent compiler stack to enable an interoperable build of hypre with BML, with OpenMP offload and vendor-specific compiler constraints. These challenges were eventually resolved. The performance results for the PROGRESS SP2 benchmark using the sparse–sparse matrix multiplication from the hypre library are shown in Fig. 7 for an NVIDIA V100 GPU. Two data points using the NVIDIA cuSPARSE library are also shown. These correspond to the two smallest matrix sizes in the PROGRESS benchmark—the only two that we were able to complete using the NVIDIA CUDA 11 toolkit due to the large memory requirements for the cuSPARSE sparse–sparse matrix multiplication. They show a significantly better performance using hypre (factor 6-7X). The results from Sec. VII A using rocSPARSE on AMD MI250X GPU, are also shown for reference. Taking the differences in hardware performance into account —about 3X more flops and 1.8X better memory bandwidth for the AMD—GPU—they indicate a comparable performance for hypre and rocSPARSE sparse–sparse matrix multiplications.

VIII. DISTRIBUTED MEMORY SOLVERS

Distributed memory approaches are very attractive for large problems that do not fit in the memory of a single node. However, they can also be used to speed up time-to-solution. There is obviously always a cost to distributing a problem across multiple nodes due to communication and possibly extra computation for overlapping work. However, in many cases, these costs can be managed to a

reasonable fraction of the compute time, and calculations can benefit from the distributed resources.

Here, we distinguish between two general ways of distributing an atomistic simulation: distributing the linear algebra problem of computing a DM or dividing the physical system into a set of (potentially overlapping) subsystems. The ScaLAPACK library[60] is probably the most well-known library implementing distributed linear algebra operations. In BML, we implemented a distributed matrix format that leverages the operations implemented for the various shared memory formats in BML. This approach is described in Sec. VIII A. Partitioning the DM computation at the physical level was proposed by Yang and Lee in 1995 using a divide and conquer approach.[61] In Sec. VIII B, we present a related approach implemented in PROGRESS. It relies purely on matrix elements to determine the partitioning of the system and is thus, very appropriate for a library implementation totally independent of any specific electronic structure code. It also leverages shared memory solvers implemented in BML, which are used for the linear algebra operations performed for each physical sub-system.

A. Distributed linear algebra

In order to leverage the implementation performed for all the shared memory matrix formats in BML, we introduced a distributed matrix format, in which each block owned by an MPI task is a BML matrix in a non-distributed (shared memory) format. This allows us to have a distributed format for all the shared memory matrix formats already implemented in BML. We introduce the following restrictions with this format: (i) the sub-matrices owned by each MPI task have to be square matrices and (ii) the number of MPI tasks used has to be a squared integer. We named that new format "distributed2d." This format is built with the BML library when BML is configured to be built with MPI.

Our implementation is non-intrusive, leaving the shared memory matrix formats untouched. It consists mainly of "wrapper" functions calling sub-matrix operations when possible, implementing the "distributed2d" matrix operations as combinations of "shared memory" matrix operations. Some operations, such as the Frobenius norm for instance, need an MPI reduction at the end to get the global values. Some operations (multiplication, transpose, ...) require more substantial communications. Our current implementation of the distributed matrix–matrix multiplication is based on Cannon's algorithm. Some operations are intrinsically more intrusive, for instance, computing the bounds on the eigenvalues of a matrix using Gershgorin circles. In this case, the strategy is to add functionalities to the basic formats to be used by "distributed2d" implementation. Some operations are beyond the scope of this project. For instance, implementing a distributed eigensolver would require a lot of work beyond the resources of this project. Fortunately, other libraries offer good distributed eigensolvers. Thus, in BML, our eigensolver is simply interfacing with an existing solver. For CPU, we have implemented an interface to ScaLAPACK.[60] For GPU, we have implemented an interface to the ELPA library.[62]

B. Graph-based divide electronic structure

Recent findings[63] by Niklasson and co-workers have demonstrated, both theoretically and practically, that there exists a bijective correspondence between matrix functions of sparse matrices and the same functions applied to only certain graph-restricted domains (parts) of the matrices. This theory enables the decomposition of the problem of computing a sparse matrix function into sub-problems involving much smaller dense matrices. The technique achieves near-perfect parallelism, in which computations can be executed with distributed memory and minimal communication.[64,65] The significance is twofold: it not only offers a systematic approach to addressing matrix function calculations by breaking them down into manageable components but also capitalizes on the power of parallel computing to handle these computations concurrently.

The original article used electronic structure calculations (the application of the Fermi–Dirac function) to validate this technique and was coined "graph-based linear-scaling electronic structure theory." The utilization of electronic structure concepts to verify this theoretical result underscores its practical relevance in real-world applications, particularly in the domain of large-scale scientific computing. Other researchers embraced this concept and translated it into libraries designed for the parallel application of matrix functions on massively large scales.[66]

Within the context of the PROGRESS library, we apply the Fermi–Dirac function to compute the system's DM. This technique, however, can be generalized to other functions. Given a Hamiltonian matrix H and a threshold τ, one can create a graph G by constructing its adjacency matrix A defined as

$$A_{ij} = \begin{cases} 1 & \text{if } |H_{ij}| \geq \tau \text{ and } i \neq j, \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

This graph can be partitioned into several components (or parts) using various methods referred to as graph-partitioning techniques.[67] We define a partition Π of graph G into m parts, as a collection of m sets of nodes of G, $\Pi = \{\pi_1, \ldots \pi_i, \ldots \pi_m\}$.

The PROGRESS library provides several options for graph partitioning. Perhaps the most straightforward one is to simply divide the matrix index list into relatively regular segments. PROGRESS also offers partitioning through the use of the METIS library, which implements various algorithms based on multiple constraint partitionings.[68] In addition, PROGRESS provides a partitioning algorithm with the objective of minimizing the total number of arithmetic operations if a $\mathcal{O}(N^3)$ complexity would dominate the mathematical operations performed over each set of nodes. PROGRESS also implements several variations of graph partitioning inspired by the Kernighan–Lin method.[69] Finally, there is a method in which a METIS partitioning is refined using a simulated annealing technique to minimize the number of operations in an $\mathcal{O}(N^3)$ complexity algorithm.

Within the context of electronic structure, two concepts immediately follow the idea of a partition: *core* partition and *core-halo* partition. A partition is called a *core* partition, $\Pi_c = \{\pi_1^c, \ldots \pi_i^c, \ldots \pi_m^c\}$, if every node in G belongs to one and only one of the $\pi_k^c \in \Pi_c$. If we extend each core component $\pi_k^c \in \Pi_c$ to also include the neighboring nodes of every node in π^c, this defines a *core-halo* partition, $\Pi_{ch} = \{\pi_1^{ch}, \ldots \pi_i^{ch}, \ldots \pi_m^{ch}\}$. Thus, if a node l belongs to π_k^c, then every directly connected neighboring node o (where $A_{l,o} = 1$) belongs to π_k^{ch}.

Given a partition Π of G, for every element $\pi_k \in \Pi$, we can extract a submatrix $h_{\alpha\beta}^k := H_{i \in \pi_k, j \in \pi_k}$. Then there is a one-to-one

mapping between indices of h and the indices of H, that is, if s_k is the size (number of nodes) of π_k, $\alpha \in [1, s_k] \leftrightarrow i \in \pi_k$. BML implements the functionalities to extract a submatrix h from a global matrix H and to map data from a submatrix h into a global matrix H.

To prevent discontinuities, our approach involves extracting submatrices from core-halo partitions, applying the necessary functions to these submatrices and then subsequently constructing the final matrices. This is accomplished by first constructing a core-halo partition from a core partition. The extension is based on an "auxiliary" matrix that defines the connectivity of the orbitals. This auxiliary matrix can be the Hamiltonian itself, the overlap matrix, or a previously computed DM. A previously computed DM can be obtained from previous time steps (or a previous SCF step) during a geometry optimization and/or a QMD simulation. From this extension, we can extract the submatrix $h^{k_{ch}}$ for each part k and apply a matrix function to it. Once the function has been evaluated for every part k, we can then extract the submatrices corresponding to the "core" partition and map them back to a full system density matrix P, where $P \approx f(H)$.

In Listing 11, we present a general algorithm that applies this graph-based approach to computing the density matrix. The inputs are the Hamiltonian (`h_bml`) matrix and the auxiliary matrix (`g_bml`) that serves as a guess for the connectivity of the orbitals. In our PROGRESS implementation, all the matrices in the interface are sparse matrices. The construction of the graph is handled using the BML ELLPACK-R format given the sizes of the adjacency matrices that are typically involved in large systems that need memory distributed techniques. The auxiliary matrix is converted into a weighted adjacency matrix by taking the absolute values of every entry and a thresholding operation is applied to control the extent of the resulting graph. The resulting graph is then used in a graph partitioning algorithm to get the parts. The graph parts are then used to define the Hamiltonian sub-matrices (dense BML matrices), which are then solved for independently and concurrently using, for example, a regular diagonalization method (`prg_build_DM()`). From the several DM sub-matrices obtained, each MPI task will locally reconstruct a "partially filled" full DM, before the full DM is assembled by summing up all contributions using an MPI reduction operation. Figure 8 shows the average DM element error and the average CH size as a function of the threshold value picked to build the adjacency matrix. We see that we get a well-controlled error (Frobenius norm of the difference between the graph-based DM and the DM obtained using the dense diagonalization) by modifying the thresholding parameter used to construct the adjacency matrix. The average element error follows a linear function of the threshold on a log–log plot, indicating that the error is a polynomial of the threshold (error $\sim \tau^{2.2}$ in this case). When the submatrices are extracted, they contain a halo region (extra layer of surrounding orbitals), which is an extension from the cores (extracted set of atoms) arising from the overlapping graph-partitioning process. The smaller the threshold, the larger the overlap between the different parts and the smaller the error committed. On the other hand, the smaller the threshold, the larger the individual Hamiltonians that need to be solved independently and the higher the computational cost.

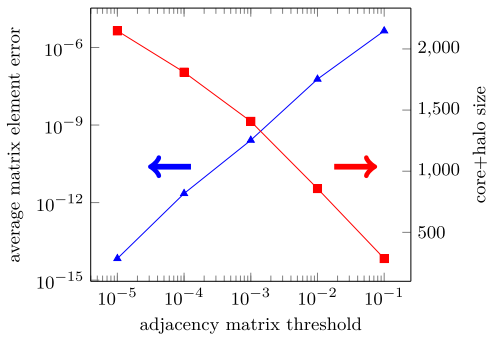

FIG. 8. Error vs thresholding parameter used to construct the adjacency matrix (blue) with average sizes of core + halo parts (red). These results were obtained from the PROGRESS benchmark Hamiltonian for $N = 2162$ and a partition into eight parts.

LISTING 11. General density matrix graph-based distributed solver routine as implemented in PROGRESS.

```
!given a Hamiltonian matrix h_bml and a matrix g_bml
!describing the graph, compute DM d_bml
!tau: threshold for graph partition
subroutine prg_graphSolver(h_bml,g_bml,d_bml,tau,...)
  !Constructing the graph
  call prg_get_graph(g_bml,tau,...)
  !Partitioning the graph
  call prg_metisPartition(g_bml)

  !loop over local parts
  do i= localPartMin(myRank), localPartMax(myRank)
    !Extracting Hamiltonian sub-matrices
    call bml_matrix2submatrix(h_bml,s_h_bml(i),...)
    !Solving the sub-problem
    call prg_build_DM(s_h_bml(i), s_d_bml(i)...)
    !Add solution of sub-problem to DM
    call bml_submatrix2matrix(s_d_bml(i),d_bml,...)
  end do
  !Collect local and remote DM parts
  prg_collectMatrixFromParts(d_bml,...)
end program
```

We tested this graph-based approach performance by evaluating the wall-clock time for the construction of DM as a function of the problem size. In Fig. 9, we show the timings obtained and compare those to a standard dense diagonalization. The computational cost as a function of problem size is remarkably promising. On a single Intel CPU node, the graph-based approach has a very low prefactor scaling as compared to the regular DM construction method (see Fig. 9) and becomes competitive for matrix size $N = 2000$ and beyond. We also see a larger speedup for larger adjacency matrix threshold (Fig. 10).

With modern hybrid architectures with GPU-accelerated nodes, speedups can be obtained only for larger problems. The overhead associated with this distributed graph-based approach includes (i) partitioning the global matrix into "core + halo" parts, (ii) extracting the dense submatrices associated with each part from large sparse matrices, and (iii) communications between MPI tasks to gather the calculated submatrices into the resulting global DM. In Table II, we present the matrix and submatrix sizes used by the graph-solver, as well as the performance numbers obtained on NVIDIA GPUs using

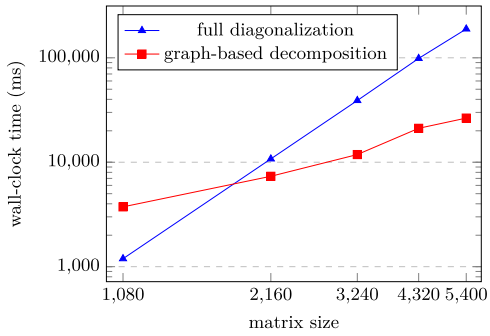

FIG. 9. Wall-clock time vs problem size for a fixed threshold of 0.01 for the graph-based solver using eight parts (red). This is compared to the wall-clock time of a dense diagonalization of the entire problem (blue). Runs on a single Intel Core i7 CPU node with 12 OpenMP threads.

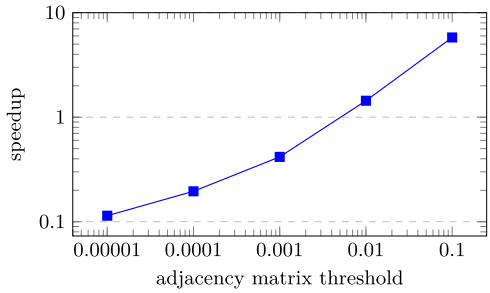

FIG. 10. Speedup with the distributed graph solver compared to serial global solver as a function of the threshold used for graph partitioning. The results obtained for the PROGRESS benchmark with $N = 2162$.

TABLE II. Performance of the graph-based density matrix construction for a Hamiltonian matrix of size 12 972. Run on one node of Summit at OLCF (NVIDIA V100 GPUs). The local, single GPU solver is NVIDIA cuSolver "cusolverDnDsyevd." The graph was constructed from the (precomputed) DM itself with a threshold of 10^{-2}.

Number of parts	4
Number of MPI tasks	4
Number of GPUs	4
Matrix size	12 972
Number of core nodes/part	3 149–3 314
Number of core + halo nodes/part	5 657–5 810
Wall-clock time for local solver (s)	2.0
Total wall-clock time distributed solver (s)	5.6
Wall-clock time single GPU solver (s)	8.7
Speedup	1.55X

a molecular system from the PROGRESS benchmark described in Sec. II. In this case, the difference between the time spent in the local solver and the total time for the distributed solver shows the time spent in the overhead operations (3.6 s). While it cannot be eliminated totally, we expect future code optimization to reduce it significantly.

IX. CONCLUDING REMARKS

Even though the software development environment using high-performance computing resources with GPU accelerators has improved substantially in recent years, it is still a challenge to produce software that is performant, portable, and maintainable.

With modern hybrid architectures, where more than 90% of the flops are those in the GPU, more and more scientific code needs to be developed and optimized for GPU execution. From a numerical point of view, maximal utilization of a GPU is both a lot of work and technically challenging and may even require algorithm redesign. Dense eigenvalue solvers routinely used in the electronic structure community do not get the speedup that one might expect on a GPU based on flops specifications, while other solvers based on matrix multiplications perform much better.

In this paper, we demonstrated some ideas on how to address some of these issues and described libraries (PROGRESS and BML) where these techniques are implemented. From a performance point of view, when comparing the dense matrix-multiplication-based iterative solver, SP2, with traditional dense diagonalization, we showed some significant speedups using AMD and Intel GPUs. For $\mathcal{O}(N)$ solvers based on sparse SP2, focusing on NVIDIA and AMD GPUs, we demonstrated how to leverage third-party libraries for core numerical kernels within an OpenMP offload implementation to achieve a better performance than dense solvers for matrix sizes beyond $N = 3000$. We also showed how some distributed memory solvers can be implemented, leveraging the performance of the shared memory operations implemented in BML. For these, the challenge remains to keep the data transfer overhead low in comparison to on-GPU operations, which are extremely fast on high-end devices.

From a practical point of view, the software stack can still be quite unstable and building a set of working libraries and code together remains a challenge. In addition, OpenMP offload support differs from compiler to compiler, which, with fewer debugging options than in CPUs, can require significant code development efforts. While we expect software stack stability and reliability to improve with maturity of the technology used in today's largest HPC resources, we also expect some challenges to persist for quite some time.

ACKNOWLEDGMENTS

This manuscript has been authored in part by UT-Battelle, LLC, under Contract No. DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This work was also performed by members of the Los Alamos National Laboratory. The Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy (Contract No. 89233218NCA000001). Portions of this work were

performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344.

This work was performed as part of the Co-design Center for Particle Applications, supported by the Exascale Computing Project (No. 17-SC-20-SC), a collaborative effort of the DOE Office of Science and the National Nuclear Security Administration (NNSA). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

The authors thank Rui Peng Li, LLNL, for his insightful comments and assistance in the integration of hypre into the BML library. The authors also thank Colleen Bertoni, ANL, for her assistance while working on Aurora and Sunspot.

AUTHOR DECLARATIONS

Conflict of Interest

The authors have no conflicts to disclose.

Author Contributions

Jean-Luc Fattebert: Conceptualization (lead); Formal analysis (equal); Investigation (equal); Methodology (equal); Software (lead); Supervision (lead); Visualization (equal); Writing – original draft (lead); Writing – review & editing (lead). **Christian F. A. Negre**: Conceptualization (equal); Investigation (equal); Methodology (equal); Software (equal); Supervision (equal); Writing – original draft (equal). **Joshua Finkelstein**: Methodology (equal); Software (equal); Visualization (equal); Writing – original draft (equal); Writing – review & editing (equal). **Jamaludin Mohd-Yusof**: Investigation (equal); Software (equal); Writing – original draft (equal). **Daniel Osei-Kuffuor**: Investigation (equal); Methodology (equal); Software (equal); Writing – original draft (equal). **Michael E. Wall**: Investigation (equal); Methodology (equal); Software (equal); Visualization (equal); Writing – original draft (equal). **Yu Zhang**: Software (equal). **Nicolas Bock**: Software (equal). **Susan M. Mniszewski**: Conceptualization (equal); Funding acquisition (lead); Methodology (equal); Project administration (lead); Software (equal); Supervision (equal); Writing – review & editing (equal).

DATA AVAILABILITY

The PROGRESS and BML software libraries discussed in this paper are publicly available under open source licenses at https://github.com/lanl/qmd-progress and https://github.com/lanl/bml, respectively. The data that support the findings of this study are available from the corresponding author upon reasonable request.

REFERENCES

[1] D. Tom, C. James, W.-C. Lin, and S. McIntosh-Smith, "Heterogeneous programming for the homogeneous majority," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (IEEE, 2022), pp. 1–13.

[2] L. Luo, T. P. Straatsma, L. E. Aguilar Suarez, R. Broer, D. Bykov, E. F. D'Azevedo, S. S. Faraji, K. C. Gottiparthi, C. De Graaf, J. A. Harris, R. W. A. Havenith, H. J. Aa. Jensen, W. Joubert, R. K. Kathir, J. Larkin, Y. W. Li, D. I. Lyakh, O. E. B. Messer, M. R. Norman, J. C. Oefelein, R. Sankaran, A. F. Tillack, A. L. Barnes, L. Visscher, J. C. Wells, and M. Wibowo, "Pre-exascale accelerated application development: The ORNL Summit experience," IBM J. Res. Dev. **64**(3/4), 11:1–11:21 (2020).

[3] A. Scott, C. Zimmer, J. Lange, D. Bernholdt, V. M. Vergara, T. Beck, M. Brim, R. Budiardja, S. Chandrasekaran, M. Eisenbach, T. Evans, M. Ezell, N. Frontiere, A. Georgiadou, J. Glenski, P. Grete, S. Hamilton, J. Holmen, A. Huebl, D. Jacobson, W. Joubert, K. Mcmahon, E. Merzari, S. Moore, A. Myers, S. Nichols, S. Oral, T. Papatheodore, D. Perez, D. M. Rogers, E. Schneider, J.-L. Vay, and P. K. Yeung, "Frontier: Exploring exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'23* (Association for Computing Machinery, New York, 2023).

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," ACM Trans. Math. Software **5**(3), 308–323 (1979).

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of Fortran basic linear algebra subprograms," ACM Trans. Math. Software **14**(1), 1–17 (1988).

[6] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," ACM Trans. Math. Software **16**(1), 1–17 (1990).

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999).

[8] D. Jack, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with GPUs," Numerical Computations with GPUs (2014), pp. 1–26.

[9] C. Brown, A. Ahmad, S. Tomov, and D. Jack, "Design, optimization, and benchmarking of dense linear algebra algorithms on AMD GPUs," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)* (IEEE, 2020), pp. 1–7.

[10] M. Gates, A. YarKhan, S. Dalal, K. Akbudak, S. Cayrols, D. Bielich, A. Ahmad, M. Al Farhan, and D. Jack, "Portable and efficient dense linear algebra in the beginning of the exascale era," in *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (IEEE, 2022), pp. 36–46.

[11] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," ACM Trans. Math. Software **48**(1), 2 (2022).

[12] N. Bock, C. F. A. Negre, S. M. Mniszewski, J. Mohd-Yusof, B. . Aradi, J.-L. Fattebert, D. Osei-Kuffuor, T. C. Germann, and A. M. N. Niklasson, "The basic matrix library (BML) for quantum chemistry," J. Supercomput. **74**(11), 6201–6219 (2018).

[13] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," Parallel Comput. **40**(5-6), 47 (2014).

[14] O. Schütt, P. Messmer, J. . Hutter, and J. VandeVondele, *GPU-accelerated Sparse Matrix–Matrix Multiplication for Linear Scaling Density Functional Theory* (John Wiley & Sons, Ltd, 2016), Chap. VIII, pp. 173–190.

[15] T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann, D. Golze, J. Wilhelm, S. Chulkov, M. H. Bani-Hashemian, V. Weber, U. Borštnik, M. Taillefumier, A. S. Jakobovits, A. Lazzaro, H. Pabst, T. Müller, R. Schade, M. Guidon, S. Andermatt, N. Holmberg, G. K. Schenter, A. Hehn, A. Bussy, F. Belleflamme, G. Tabacchi, A. Glöß, M. Lass, I. Bethune, C. J. Mundy, C. Plessl, M. Watkins, J. Van de Vondele, M. Krack, and J. Hutter, "CP2K: An electronic structure and molecular dynamics software package—Quickstep: Efficient and accurate electronic structure calculations," J. Chem. Phys. **152**(19), 194103 (2020).

[16] V. W. Z. Yu, C. Campos, W. Dawson, A. García, V. Havu, B. Hourahine, W. P. Huhn, M. Jacquelin, W. Jia, M. Keçeli, R. Laasner, Y. Li, L. Lin, J. Lu, J. Moussa, J. E. Roman, Á. Vázquez-Mayagoitia, C. Yang, and V. Blum, "ELSI—An open infrastructure for electronic structure solvers," Comput. Phys. Commun. **256**, 107459 (2020).

[17]M. J. T. Oliveira, N. Papior, Y. Pouillon, V. Blum, E. Artacho, D. Caliste, F. Corsetti, S. de Gironcoli, A. M. Elena, A. García, V. M. García-Suárez, L. Genovese, W. P. Huhn, G. Huhs, S. Kokott, E. Küçükbenli, A. H. Larsen, A. Lazzaro, I. V. Lebedeva, Y. Li, D. López-Durán, P. López-Tarifa, M. Lüders, M. A. L. Marques, J. Minar, S. Mohr, A. A. Mostofi, A. O'Cais, M. C. Payne, T. Ruh, D. G. A. Smith, J. M. Soler, D. A. Strubbe, N. Tancogne-Dejean, D. Tildesley, M. Torrent, and V. W. Z. Yu, "The CECAM electronic structure library and the modular software development paradigm," J. Chem. Phys. **153**(2), 024117 (2020).
[18]I. V. Lebedeva, A. García, E. Artacho, and P. Ordejón, "Modular implementation of the linear- and cubic-scaling orbital minimization methods in electronic structure codes using atomic orbitals," R. Soc. Open Sci. **10**(4), 230063 (2023).
[19]F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón, "Improving the performance of the sparse matrix vector product with GPUs," in *2010 10th IEEE International Conference on Computer and Information Technology* (IEEE, 2010), pp. 1146–1151.
[20]S. M. Mniszewski, J. Belak, J.-L. Fattebert, C. F. A. Negre, S. R. Slattery, A. A. Adedoyin, R. F. Bird, C. Chang, G. Chen, S. Ethier, S. Fogerty, S. Habib, C. Junghans, D. Lebrun-Grandié, J. Mohd-Yusof, S. G. Moore, D. Osei-Kuffuor, S. J. Plimpton, A. Pope, S. T. Reeve, L. Ricketson, A. Scheinberg, A. Y. Sharma, and M. E. Wall, "Enabling particle applications for exascale computing platforms," Int. J. High Perform. Comput. Appl. **35**(6), 572–597 (2021).
[21]A. M. N. Niklasson, C. J. Tymczak, and M. Challacombe, "Trace resetting density matrix purification in O(N) self-consistent-field theory," J. Chem. Phys. **118**(19), 8611–8620 (2003).
[22]S. Goedecker and M. Teter, "Tight-binding electronic-structure calculations and tight-binding molecular dynamics with localized orbitals," Phys. Rev. B **51**, 9455–9464 (1995).
[23]C. F. A. Negre, N. Bock, and S. M. Mniszewski, *BML, version 2.4*, 2023, see https://github.com/lanl/bml.
[24]A. M. Niklasson, S. M. Mniszewski, F. Christian, A. Negre, M. E. Wall, M. J. Cawkwell, and N. Bock, *PROGRESS, Version 1.3*, 2023, see https://github.com/lanl/qmd-progress.
[25]Th. Frauenheim, G. Seifert, M. Elsterner, Z. Hajnal, G. Jungnickel, D. Porezag, S. Suhai, and R. Scholz, "A self-consistent charge density-functional based tight-binding method for predictive materials simulations in physics, chemistry and biology," Phys. Status Solidi B **217**(1), 41–62 (2000).
[26]N. Bock, M. J. Cawkwell, J. D. Coe, A. Krishnapriyan, M. P. Kroonblawd, A. Lang, C. Liu, E. Martinez Saez, S. M. Mniszewski, C. F. A. Negre, A. M. N. Niklasson, E. Sanville, M. A. Wood, and P. Yang, *LATTE: Developer repository for the LATTE code*, 2023, see https://github.com/lanl/LATTE.
[27]Schrödinger, LLC, *The PyMOL molecular graphics system, version 1.8*, 2015.
[28]See https://cmake.org for CMake; accessed 19 December 2023.
[29]I. Štich, R. Car, M. Parrinello, and S. Baroni, "Conjugate gradient minimization of the energy functional: A new method for electronic structure calculation," Phys. Rev. B **39**, 4997–5004 (1989).
[30]G. Kresse and J. Furthmüller, "Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set," Comput. Mater. Sci. **6**(1), 15–50 (1996).
[31]A. R. Tackett, N. A. W. Holzwarth, and G. E. Matthews, "A projector augmented wave (PAW) code for electronic structure calculations, Part II: Pwpaw for periodic solids in a plane wave basis," Comput. Phys. Commun. **135**(3), 348–376 (2001).
[32]C. Yang, J. C. Meza, and L.-W. Wang, "A constrained optimization algorithm for total energy minimization in electronic structure calculations," J. Comput. Phys. **217**(2), 709–721 (2006).
[33]J.-L. Fattebert, "A robust solver for wavefunction-based density functional theory calculations," Electron. Struct. **4**(1), 015002 (2022).
[34]F. Bottin, S. Leroux, A. Knyazev, and G. Zérah, "Large-scale ab initio calculations based on three levels of parallelization," Comput. Mater. Sci. **42**(2), 329–336 (2008).
[35]A. Levitt and M. Torrent, "Parallel eigensolvers in plane-wave density functional theory," Comput. Phys. Commun. **187**, 98–105 (2015).
[36]M. Lupo Pasini, B. Turcksin, W. Ge, and J.-L. Fattebert, "A parallel strategy for density functional theory computations on accelerated nodes," Parallel Comput. **100**, 102703 (2020).
[37]J. Finkelstein, C. F. A. Negre, and J.-L. Fattebert, "A fast, dense Chebyshev solver for electronic structure on GPUs," J. Chem. Phys. **159**(10), 101101 (2023).
[38]S. Khadatkar and P. Motamarri, "Subspace recursive Fermi-operator expansion strategies for large-scale DFT eigenvalue problems on HPC architectures," J. Chem. Phys. **159**(3), 031102 (2023).
[39]S. Goedecker, "Linear scaling electronic structure methods," Rev. Mod. Phys. **71**, 1085–1123 (1999).
[40]S. Mohr, W. Dawson, M. Wagner, D. Caliste, T. Nakajima, and L. Genovese, "Efficient computation of sparse matrix functions for large-scale electronic structure calculations: The CheSS library," J. Chem. Theory Comput. **13**(10), 4684–4698 (2017).
[41]M. S. Paterson and L. J. Stockmeyer, "On the number of nonscalar multiplications necessary to evaluate polynomials," SIAM J. Comput. **2**(1), 60–66 (1973).
[42]W. Z. Liang, C. Saravanan, Y. Shao, R. Baer, A. T. Bell, and M. Head-Gordon, "Improved Fermi operator expansion methods for fast electronic structure calculations," J. Chem. Phys. **119**(8), 4117–4125 (2003).
[43]W. Z. Liang, R. Baer, C. Saravanan, Y. Shao, A. T. Bell, and M. Head-Gordon, "Fast methods for resumming matrix polynomials and Chebyshev matrix polynomials," J. Comput. Phys. **194**(2), 575–587 (2004).
[44]See https://www.openmp.org/spec-html/5.2/openmp.html for OpenMP Application Programming Interface.
[45]B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The ongoing evolution of OpenMP," Proc. IEEE **106**(11), 2004–2019 (2018).
[46]See https://www.openmp.org/resources/openmp-compilers-tools for OpenMP Compilers and Tools.
[47]S. Bak, C. Bertoni, S. Boehm, R. Budiardja, B. M. Chapman, J. Doerfert, M. Eisenbach, H. Finkel, O. Hernandez, J. Huber, S. Iwasaki, V. Kale, P. R. Kent, J. Kwack, M. Lin, P. Luszczek, Y. Luo, B. Pham, S. Pophale, K. Ravikumar, V. Sarkar, T. Scogland, S. Tian, P. Yeung, and P. K. Yeung, "OpenMP application experiences: Porting to accelerated nodes," Parallel Comput. **109**, 102856 (2022).
[48]J.-L. Fattebert, S. DeWitt, A. Perron, and J. Turner, "Thermo4PFM: Facilitating phase-field simulations of alloys with thermodynamic driving forces," Comput. Phys. Commun. **288**, 108739 (2023).
[49]A. S. Sabau, L. Yuan, J.-L. Fattebert, and J. A. Turner, "An OpenMP GPU-offload implementation of a non-equilibrium solidification cellular automata model for additive manufacturing," Comput. Phys. Commun. **284**, 108605 (2023).
[50]Y. Luo, P. Doak, and P. Kent, "A high-performance design for hierarchical parallelism in the QMCPACK Monte Carlo code," in *2022 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)* (IEEE, 2022), pp. 22–27.
[51]D. Datta and M. S. Gordon, "Accelerating coupled-cluster calculations with GPUs: An implementation of the density-fitted CCSD(T) approach for heterogeneous computing architectures using OpenMP directives," J. Chem. Theory Comput. **19**(21), 7640–7657 (2023).
[52]B. Q. Pham, M. Alkan, and M. S. Gordon, "Porting fragmentation methods to graphical processing units using an OpenMP application programming interface: Offloading the Fock build for low angular momentum functions," J. Chem. Theory Comput. **19**(8), 2213–2221 (2023).
[53]F. Rabbi, C. S. Daley, H. M. Aktulga, and N. J. Wright, "Evaluation of directive-based GPU programming models on a block eigensolver with consideration of large sparse matrices," in *Accelerator Programming Using Directives*, edited by S. Wienke and S. Bhalachandra (Springer International Publishing, Cham, 2020), pp. 66–88.
[54]J. Mohd-Yusof and N. Sakharnykh, "Fast sparse matrix multiplication for QMD using parallel merge," in GPU Technology Conference (GTC), 2015.
[55]See https://icl.utk.edu/magma/index.html for MAGMA: Matrix Algebra on GPU and Multicore Architectures.
[56]See https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html for Intel oneAPI Base Toolkit.
[57]See https://www.alcf.anl.gov/aurora-media-kit for Aurora Media Kit.

[58]See https://llnl.gov/casc/hypre, https://github.com/hypre-space/hypre for hypre: High Performance Preconditioners.
[59]R. D. Falgout, R. Li, B. Sjögreen, L. Wang, and U. M. Yang, "Porting hypre to heterogeneous computer architectures: Strategies and experiences," Parallel Comput. **108**, 102840 (2021).
[60]L. Blackford, J. Choi, A. Cleary, J. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide* (SIAM, 1997).
[61]W. Yang and T.-S. Lee, "A density-matrix divide-and-conquer approach for electronic structure calculations of large molecules," J. Chem. Phys. **103**(13), 5674–5678 (1995).
[62]V. W. Z. Yu, J. Moussa, P. Kůs, A. Marek, P. Messmer, M. Yoon, H. Lederer, and V. Blum, "GPU-acceleration of the ELPA2 distributed eigensolver for dense symmetric and hermitian eigenproblems," Comput. Phys. Commun. **262**, 107808 (2021).
[63]A. M. N. Niklasson, S. M. Mniszewski, C. F. A. Negre, M. J. Cawkwell, P. J. Swart, J. Mohd-Yusof, T. C. Germann, M. E. Wall, N. Bock, E. H. Rubensson, and H. Djidjev, "Graph-based linear scaling electronic structure theory," J. Chem. Phys. **144**(23), 234101 (2016).
[64]H. N. Djidjev, G. Hahn, S. M. Mniszewski, C. F. A. Negre, A. M. N. Niklasson, and V. B. Sardeshmukh, *Graph Partitioning Methods for Fast Parallel Quantum Molecular Dynamics* (Society for Industrial and Applied Mathematics, 2016), pp. 42–51.
[65]P. Ghale, M. P. Kroonblawd, S. Mniszewski, C. F. A. Negre, R. Pavel, S. Pino, V. Sardeshmukh, G. Shi, and G. Hahn, "Task-based parallel computation of the density matrix in quantum-based molecular dynamics using graph partitioning," SIAM J. Sci. Comput. **39**(6), C466–C480 (2017).
[66]W. Dawson and T. Nakajima, "Massively parallel sparse matrix function calculations with NTPoly," Comput. Phys. Commun. **225**, 154–165 (2018).
[67]H. W. Y. Adoni, T. Nahhal, M. Krichen, B. Aghezzaf, and A. Elbyed, "A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems," Distrib. Parallel Databases **38**(2), 495–530 (2020).
[68]See http://glaros.dtc.umn.edu/gkhome/metis/metis/overview for METIS—Serial graph partitioning and fill-reducing matrix ordering; accessed 3 May 2022.
[69]B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," Bell Syst. Tech. J. **49**(2), 291–307 (1970).