Parallel Computing 109 (2022) 102856

Contents lists available at ScienceDirect

Parallel Computing

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

OpenMP application experiences: Porting to accelerated nodes

Seonmyeong Bak [c], Colleen Bertoni [a], Swen Boehm [f], Reuben Budiardja [f], Barbara M. Chapman [b], Johannes Doerfert [a], Markus Eisenbach [f], Hal Finkel [a], Oscar Hernandez [f], Joseph Huber [f], Shintaro Iwasaki [a], Vivek Kale [b,*], Paul R.C. Kent [f], JaeHyuk Kwack [a], Meifeng Lin [b], Piotr Luszczek [h], Ye Luo [a], Buu Pham [d], Swaroop Pophale [f], Kiran Ravikumar [c], Vivek Sarkar [c], Thomas Scogland [e], Shilei Tian [g], P.K. Yeung [c]

[a] *Argonne National Laboratory, Lemont, IL 60439, USA*
[b] *Brookhaven National Laboratory, Upton, NY 11973, USA*
[c] *Georgia Institute of Technology, Atlanta, GA 30332, USA*
[d] *Iowa State University, Ames, IA 50011, USA*
[e] *Lawrence Livermore National Laboratory, Livermore CA 94550, USA*
[f] *Oak Ridge National Laboratory, One Bethel Valley Rd., Oak Ridge, TN 37830, USA*
[g] *Stony Brook University, Stony Brook, NY, 11794, USA*
[h] *University of Tennessee, Knoxville TN, 37996, USA*

ARTICLE INFO

Keywords:
Application porting experiences
Accelerators
High performance computing
OpenMP implementations
GAMESS
GenASiS
GESTS
GridQCD
LSMS
QMCPACK
SLATE
RAJA

ABSTRACT

As recent enhancements to the OpenMP specification become available in its implementations, there is a need to share the results of experimentation in order to better understand the OpenMP implementation's behavior in practice, to identify pitfalls, and to learn how the implementations can be effectively deployed in scientific codes. We report on experiences gained and practices adopted when using OpenMP to port a variety of ECP applications, mini-apps and libraries based on different computational motifs to accelerator-based leadership-class high-performance supercomputer systems at the United States Department of Energy. Additionally, we identify important challenges and open problems related to the deployment of OpenMP. Through our report of experiences, we find that OpenMP implementations are successful on current supercomputing platforms and that OpenMP is a promising programming model to use for applications to be run on emerging and future platforms with accelerated nodes.

1. Introduction

OpenMP, the de facto directive-based standard for on-node programming, provides a convenient and flexible mechanism to exploit the substantial compute power within the nodes of today's high-performance supercomputer systems. OpenMP is one of the programming models that will be supported on exascale systems, and several applications from the United States Department of Energy (U.S. DoE) include OpenMP as part of their strategy to exploit the configured accelerators. OpenMP introduced support for accelerators in OpenMP version 4.0 via its device constructs, extended soon thereafter in OpenMP 4.5, e.g., to allow their asynchronous execution. OpenMP 5.0 introduced a range of features that address requirements of exascale application development, including the loop directive for performance portability, memory management APIs, meta-directives and variants, unified memory support and tools APIs. Many of these are partially based on proposals for extensions from the U.S. DoE's Exascale Computing Project (ECP), within which the SOLLVE team engages in a range

* Corresponding author.
E-mail addresses: sbak5@gatech.edu (S. Bak), bertoni@anl.gov (C. Bertoni), boehms@ornl.gov (S. Boehm), budiardjard@ornl.gov (R. Budiardja), barbara.chapman@stonybrook.edu (B.M. Chapman), jdoerfert@anl.gov (J. Doerfert), eisenbachm@ornl.gov (M. Eisenbach), hfinkel@anl.gov (H. Finkel), oscar@ornl.gov (O. Hernandez), huberjn@ornl.gov (J. Huber), siwasaki@anl.gov (S. Iwasaki), vkale@bnl.gov (V. Kale), kentpr@ornl.gov (P.R.C. Kent), jkwack@anl.gov (J. Kwack), mlin@bnl.gov (M. Lin), luszczek@icl.utk.edu (P. Luszczek), yeluo@anl.gov (Y. Luo), buupq@iastate.edu (B. Pham), pophales@ornl.gov (S. Pophale), kiran.r@gatech.edu (K. Ravikumar), vsarkar@gatech.edu (V. Sarkar), scogland1@llnl.gov (T. Scogland), shilei.tian@stonybrook.edu (S. Tian), pk.yeung@ae.gatech.edu (P.K. Yeung).

https://doi.org/10.1016/j.parco.2021.102856
Received 6 November 2020; Received in revised form 8 July 2021; Accepted 27 September 2021
Available online 23 October 2021

of activities with the goal of evolving OpenMP to meet the needs of exascale platforms and their users.

As new features are added to OpenMP [1,2], there is a strong need for the community to share early experiences with them, in order to understand their behavior in practice and learn how they may be effectively deployed in scientific codes. We also need to identify any pitfalls or deficiencies in either the features or their implementations, so that we can contribute to improvements in the specification and its compilers. Lastly, use cases are needed in order to train potential adopters in current best practices.

In this paper, we describe a set of applications and mini-apps that use OpenMP to program accelerated nodes, discuss how they do so, the successes and challenges experienced by the application programmers, and how they plan to use newly implemented (at the time of writing) features of OpenMP 5.0 and beyond to further optimize their codes.

2. Background

The applications that we describe in this paper and shown in Table 1 are written in Fortran and C++, and they have different computational motifs, application characteristics and requirements with respect to OpenMP support. While OpenMP has documentation for programming styles and methods to make use of OpenMP offload features [1], OpenMP must address a variety of application requirements if it is to be a broadly useful approach for node-wide application programming.

While most of the applications in Table 1 are deployed to further science in a specific domain, two of them, PLASMA/SLATE and RAJA, are libraries used to develop a wide range of science simulations; these two applications enable performance portability and offer a simple approach to offload application code, relying on OpenMP to provide this functionality. Some applications exploit advanced features of their base language (elements of Fortran2018, C++17) and require that OpenMP and its implementations support them. Applications such as GESTS require interoperability of OpenMP offload with libraries via asynchronous tasks. GenaSis expects to use OpenMP runtime library calls to manage data transfer between the accelerator and host. For some codes, e.g., PLASMA/SLATE, efficient tasking for load balancing and performance portability is a must.

2.1. OpenMP implementations

Many Fortran and C/C++ compilers support OpenMP, each with its own specific strengths and weaknesses. Given the rapid evolution of the OpenMP specification, support for recent features varies widely and is still gaining maturity. As part of the work described in this paper, we collaborated to identify and overcome shortcomings of LLVM's OpenMP support for accelerators. The SOLLVE team also created an OpenMP runtime that uses special parallel entities [3] and an MPI extension [4] to improve communication–computation overlap.

LLVM [5] has become a central part of the software development ecosystem for optimizing compilers and OpenMP implementations. Its frontend for C-like languages, clang, supports all OpenMP 4.5 features, most OpenMP 5.0 features, and some features that are expected to be part of OpenMP 5.1. An overview of the current status is available online [6]. Clang-11 defaults to OpenMP 5.0. A command-line flag is provided to base compilation on other versions of OpenMP. The LLVM OpenMP CPU runtime, libomp, originating from Intel, maintains ABI compatibility with the GCC OpenMP CPU runtime (libgomp). The user specifies each potential target architecture at build time through the `-fopenmp-targets=A,B,C` command-line flag. A list of supported targets is available online [6]. OpenMP offloading in LLVM is currently available for NVIDIA GPUs and CPU-based targets. Support for AMD GPUs and Intel GPUs, already available in the respective LLVM-based vendor compilers, will be available in the community version in the near future. Documentation on the OpenMP implementation in LLVM, including the runtimes, optimizations, frontend support, FAQ, and a setup guide is available online [6].

GCC currently supports OpenMP 5.0 with C/C++ and Fortran, except for the target directives on GPUs, which is an ongoing effort. Future versions will support the accelerator features. Intel compilers support OpenMP 4.5 and will support OpenMP 5.0 offloading capabilities targeting Intel's GPU, Xeon Phi, and CPUs. The IBM XL compiler version 16.1.6 supports OpenMP 4.5 for Power9 and NVIDIA GPUs, and there is a plan to support some OpenMP 5.0 features. PGI supports OpenMP 3.1 for the CPU and there are plans to support OpenMP 5.0 for NVIDIA GPUs in the future. Cray and AMD (AOMP) compilers leverage the LLVM infrastructure to provide OpenMP 4.5/5.0 support.

3. Application experiences

3.1. GAMESS

GAMESS (General Atomic and Molecular Electronic Structure System) [7,8] is a software package with a variety of electronic structure quantum chemistry methods, such as Hartree–Fock (HF) [9] and second-order Moller–Plesset perturbation theory with the resolution-of-the-identity approximation (RI-MP2) [10]. It is predominantly written in Fortran 77/90, with an optional C/C++ library which uses CUDA to offload code onto NVIDIA GPUs. GAMESS has traditionally used MPI together with OpenMP to run on multi-core CPUs. Several of the methods in GAMESS have been updated to optionally use OpenMP to offload computationally expensive regions to GPUs. We focus here on the GPU port of the HF and RI-MP2 methods using OpenMP.

The HF method solves a set of non-linear eigenvalue equations iteratively for the energy of a molecular system. It has two main bottlenecks: i) computation of a large number (on the order of N^4 where N is a measure of the molecular system size) of 4-index 2-electron repulsion integrals (4-2ERIs); and ii) forming an N^2 Fock matrix by contracting the N^4 4-2ERI tensor with a density matrix. HF is a fundamental method which is a starting point for many higher-accuracy methods, such as the RI-MP2 method. In the RI-MP2 method, 4-2ERIs are approximated as the product of 3-2ERIs. This simplifies the integral evaluation from 4-index to 3-index 2-electron repulsion integrals and allows the use of efficient matrix multiplication operations.

3.1.1. Implementation and optimization with OpenMP

For the HF code, we focused on the 4-2ERI evaluation. Parallelized previously with MPI+OpenMP threading on the CPU, the code contained multiple levels of conditional statements. Additionally, since computational work was not evenly assigned to threads, it suffered from load imbalance. To address this, we substantially reorganized the control flow and the order in which integrals are computed by putting conditionals into separate code blocks (see Fig. 1) and sorting the integrals ahead of time.

Just a few OpenMP directives were added to the reorganized code. As shown in Fig. 1, OpenMP directives were inserted to offload code to GPUs (and subroutines called from target regions were annotated with 'declare target'). Note that the routines to compute integrals, e.g., int1, were not modified at all. This version of GAMESS is in a development branch and supports one type of integral. For the RI-MP2 code, we focused on the computation of the perturbative correction, which is dominated by calls to a matrix multiplication routine (DGEMM). Here, the strategy to port from CPU to GPU was to merge sections of arrays so that inputs to the DGEMM call are larger, resulting in higher arithmetic intensity per DGEMM call, and fewer kernel launches. The resulting OpenMP code, discussed in detail in [11], was implemented and evaluated in a mini-app and in a development branch of GAMESS.

Table 1
Applications ported to accelerator architectures using OpenMP and their motifs and their functionality.

Applications	Motif(s)	OpenMP Features
QMCPACK	Monte Carlo	target nowait, `use_device_ptr`
GAMESS	Dense Linear Algebra	target teams distribute parallel do, target enter/exit data
GridMini	Stencil, Monte Carlo	C++17, teams distribute, thread_limit
PLASMA/SLATE	Dense Linear Algebra	task, task depend
RAJA	Stencil, all	teams distribute, reduction
LSMS	Dense Linear Algebra	declare variant, metadirective, complex numbers
GESTS	Spectral Methods	task detach, teams distribute, libraries, `omp_target_memcpy_rect`, asynch. and dep. objects
GENaSiS	Iterative Solvers	`ptr_associate`, dynamic metadirective

```
1  subroutine compute_integrals ( ... )
2    do i=1,N^4
3      ! create sorted array by method and type
4      if (method(i) .eq. 1 && type(i) .eq. 1)
5        store i in sorted (method=1,type=1)
6      endif
7      ...
8    enddo
9    ! integrals of different methods and types are evaluated separately
10   !$ omp target teams distribute parallel do map (...)
11   do i in sorted (method=1,type=1)
12     call int1 (i ,...)
13   enddo
14   !$ omp target teams distribute parallel do map (...)
15   do i in sorted (method=1,type=2)
16     call int2 (i ,...)
17   enddo
18    ...
19 end subroutine compute_integrals
```

Fig. 1. Pseudo-code for the GPU version of part of the HF algorithm.

3.1.2. Evaluation

A preliminary version of the restructured HF code was evaluated on a node of Summit with inputs of varying sizes (coronene and clusters of up to 64 water molecules, corresponding to a molecular system size N from 240 to 832). The overall speedup of the OpenMP GPU HF integral code using a single V100 GPU, relative to the OpenMP CPU HF code using two IBM Power9 CPUs, varied from $\frac{1}{5}$x to 5x with larger inputs resulting in larger speedups. In the next step, the integral sorting will be offloaded to GPUs and batching techniques will be applied to allow computation of larger molecular systems. For the RI-MP2 code, the restructuring described in [11] was evaluated on Summit. As discussed in [11], for large enough inputs, the OpenMP GPU version running on a single V100 GPU on a node of Summit was 6–7x faster than the CPU version running on two IBM Power9 CPUs of a node of Summit. This is near the best expected speedup for floating-point dominated code, since the theoretical ratio of the peak floating-point performance of one V100 to two Power9 CPUs is approximately 7x.

3.1.3. Challenges

While porting the code to GPUs, the major challenge was to rearrange it to i) increase the computation in the offloaded regions, and thus to reduce data transfer, and (ii) remove conditionals and load imbalance between GPU threads. The OpenMP implementation we used (IBM Fortran, version 16.1.1) provided the functionality required to port our code successfully.

Nevertheless, we encountered some performance challenges. For example, in our initial GPU port, we offloaded a region inside a host function which is called at each iteration of a solver. The offloaded region mapped private variables to the GPU and contained a function call which was marked as "declare target" to allow it to be called from an offloaded region. When we tested it using IBM Fortran OpenMP, the time per iteration of the solver increased. To work around this, we manually inlined the routines called in the offloaded region, which resulted in the time per call remaining constant, as expected. Unfortunately, the IBM compiler's performance did not allow us to use OpenMP's "declare target" and mapping of private variables effectively. This issue has been reported to IBM, and it does not occur with Cray Fortran.

3.2. GenASiS

GenASiS (*Gen*eral *A*strophysical *Si*mulation *S*ystem) is a code being developed for large-scale simulations of astrophysical phenomena that targets supercomputers. Currently, it is primarily intended for simulating and modeling core-collapse supernovae and the observable phenomena associated with a supernova event. Simulations using GenASiS have led to the discovery of magnetic field amplification by the Stationary Accretion Shock Instability (SASI) in a supernova environment [12,13]. Using GenASiS we have also performed an ensemble study of SASI- and convection-dominated regimes to study the nature and differences between these two types of explosions [14]. GenASiS is written in modern Fortran, leveraging the object-oriented features of the language to be an extensible and modular code.

GenASiS is organized into three main subdivisions [15–18]. Basics provides utilities generally needed by any large-scale simulation system. Mathematics provides classes for manifolds (i.e. meshes) and solvers that are agnostic of the physical system. In Physics, these solvers are endowed with the specific form of the stress–energy and theories of spacetime relevant to the physical problem at hand. Rather than a single application, GenASiS provides these classes to be instantiated by an application driver for a specific problem.

3.2.1. Implementation with OpenMP

The core of the data storage facility in GenASiS Basics is the `StorageForm` class, which consists of members and methods for handling generic data and metadata. Data are stored as a two-dimensional array where the first index typically enumerates the cells in the mesh and the second index enumerates the variables. The metadata includes units and variable names that can be used for I/O and visualization. Most solvers and storage needs of GenASiS are written using the `StorageForm` class, enabling a uniform, simplified code for functionality such as I/O and nearest-neighbor ghost cell exchange.

The OpenMP implementation of GenASiS [17] extends the `StorageForm` class with methods to mirror, associate, and synchronize the host (CPU) memory data allocation with a corresponding device (GPU) memory allocation. Listing 2 illustrates the use of the `StorageForm` class with methods relevant to manage the device memory allocation and association.

In Fig. 2, line 4 allocates the two-dimensional array class member `Value` on the host memory. Line 5 mirrors that allocation on the device and associates the memory location on the host with that on the device. This is accomplished by using the OpenMP library runtime routine `omp_target_associate_ptr()` under the hood. With this association, the next time the OpenMP runtime encounters that host variable within a `target` region, implicit mapping and data transfer are avoided since such mapping already exists. Line 7 initializes the member `Value` with problem-specific conditions. Line 8 updates the device copy of the variable with initial values, which can then be used inside the `AddKernel()` routine on the device. Suppose the result of this operation is needed back on the host. Line 14 accomplishes that by

```
1  type ( StorageForm ) :: &
2    Fields
3  ...
4  call Fields % Initialize ( [ nCells, nVariables ], ... )
5  call Fields % AllocateDevice ( )
6  ...
7  call  SetInitialConditions ( Fields % Value )
8  call Fields % UpdateDevice ( )
9  call AddKernel &
10          ( Fields % Value ( :, 1 ), &
11            Fields % Value ( :, 2 ), &
12            Fields % Value ( :, 3 ),
13            UseDevice = .true. )
14 call Fields % UpdateHost ( )
```

Fig. 2. An example of using `StorageForm` methods.

```
1  subroutine AddKernel ( A, B, C, UseDeviceOption )
2    real ( KDR ), dimension ( : ), intent ( in ) :: A, B
3    real ( KDR ), dimension ( : ), intent ( out ) :: C
4    logical ( KDL ), intent ( in ) :: UseDevice
5    integer ( KDI ) :: i
6    if ( UseDevice )
7      !$OMP target teams distribute parallel do &
8      !$OMP schedule ( static , 1 )
9      do i = 1, size ( C )
10       C ( i ) = A ( i ) + B ( i )
11     end do
12     !$OMP end target teams distribute parallel do
13   else
14     !$OMP parallel do &
15     !$OMP schedule ( runtime )
16     do i = 1, size ( C )
17       C ( i ) = A ( i ) + B ( i )
18     end do
19     !$OMP end parallel do
20   end if
21 end subroutine AddKernel
```

Fig. 3. A computational kernel with both offload and host multi-threading.

copying the data back to the host memory. Listing 3 shows the listing of the computational kernel `AddKernel ()` previously mentioned.

The OpenMP directive for offloading to the device differs minimally from OpenMP worksharing directives for the host with the addition of the `target teams distribute` directive and a different scheduling clause. We have found that a static schedule with a chunk size 1, using schedule(static , 1), gives the best performance on the device with the IBM XL and GNU compiler, while the `runtime` schedule performs the best for multi-threading on the host.

When the program enters the target region on line 10 in Listing 3, it uses the device location for the variable `A`, `B`, and `C`, which are previously mapped persistently inside the `StorageForm % AllocateDevice ()` method called on line 5 in Listing 2. Therefore, not only is there no need to have a mapping clause with the target directive, but the default mapping clause that often implies implicit data movement is also avoided since the reference count of these variables is one (already present on the device).

By hiding the implementation details of data mapping inside a method of `StorageForm` class, we simplify the directive needed to write a computational kernel. Persistent mapping and explicit control of data movement are also crucial to achieve the expected performance of offloaded computational kernels.

3.2.2. Evaluation

Fig. 4 shows the kernel timings of `RiemannProblem` implemented in GenASiS Basics The `RiemannProblem` is a multi-dimensional extension of the 1D Sod ShockTube problem. Although this problem can

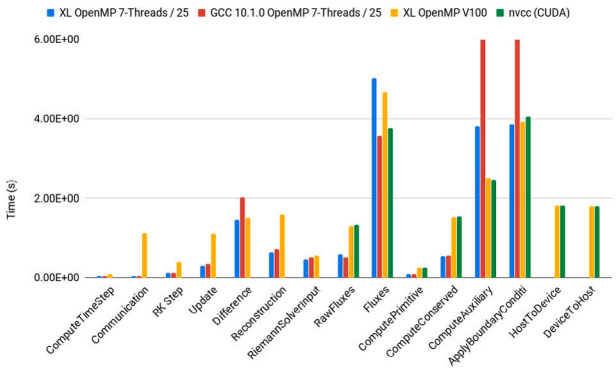

Fig. 4. Timings for computational kernels and data transfers in GenASiS Basics RiemannProblem 3D with 256^3 cells for 50 cycles. CPU multithreading timings factored by 25X to fit plot. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

be more naturally done with facilities provided by the Mathematics subdivision of GenASiS, here we implement simplified versions of the mesh infrastructure and solvers needed to solve this problem in GenASiS Basics which serves as a proxy application for the overall GenASiS fluid dynamics solvers. We have also ported six of the computational kernels to CUDA to allow us to investigate any performance differences to OpenMP offload.

We run this problem in 3D with 256^3 cells for 50 cycles. Timings for multithreading with 7 CPU threads (blue and red bars using IBM XL and GCC compilers, respectively), OpenMP offload with the XL compiler (yellow bars), and CUDA kernels with the `nvcc` compiler (green bars) are plotted. For all but one of the kernels, the OpenMP offload version obtains the same performance as the CUDA version. The CUDA version of the `Fluxes` kernel performs about 20% better than the corresponding OpenMP offload version. Although for this problem this particular kernel makes up about 16% of the total runtime, bringing the actual impact to performance closer to only $\sim$ 3%, this kernel provides a reproducer for better optimization of the OpenMP implementation that has been shared with compiler vendors.

3.2.3. Challenges using OpenMP 4.5

One challenge is that executing the computational loop in the code listing in Fig. 3 on the `host` when the device is not available requires redundant code. We would like to avoid writing the same code twice just because they need different directives for offload and CPU multithreading. This is not an artifact of GenASiS alone, as device offloading requires many other considerations with respect to data movement and updates.

A second challenge is using multiple streams with OpenMP. One of the desired features for GenASiS is the ability to call a kernel from within an OpenMP parallel loop so that all iterations execute in parallel and can ideally map to different `streams` within the GPU. Mapping to different `streams` provides another level of concurrency and, when adequate work is available per GPU, a higher occupancy can be achieved while reducing kernel launch time. This is an optimization an implementation can choose to do when the `nowait` clause is enabled in the target region. The implementation will create a target region that can be mapped to a stream if called by multiple CPU threads or if the target gets called many times in a loop.

3.2.4. Plans for OpenMP 5.0

The first challenge listed in Section 3.2.3 can be partly addressed using the `metadirective` directive introduced in OpenMP 5.0. It can specify multiple directive variants, one of which may be conditionally selected to replace the `metadirective` based on the enclosing OpenMP context. It would compress the code in Listing 3 to the code in Fig. 5.

```
1  !$OMP begin metadirective &
2  !$OMP& when(target_device={arch(nvptx)}: target teams loop )
     &
3  !$OMP& default( parallel loop )
4    do i = 1, size ( C )
5      C ( i ) = A ( i ) + B ( i )
6    end do
7  !$OMP end metadirective
```

Fig. 5. Metadirective directive for target offload and CPU multi-threading.

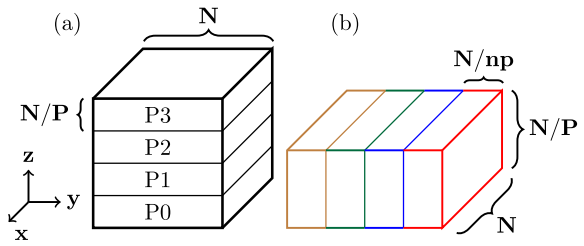

Fig. 6. Left: Decomposition of an N^3 solution domain among P MPI processes into *slabs* of data of size $N \times N \times N/P$. Right: Further decomposition of a slab into np smaller sub-volumes, each of size $N \times N/np \times N/P$.

The only drawback is that any conditional in the context selected with `metadirective` needs to be resolved at compile time, while the way GenASiS is currently structured requires runtime branch selection. OpenMP 5.1 now supports a metadirective that can select which version of the directive to use at runtime. This new feature will be beneficial here.

Another feature that we plan to employ is the OpenMP `allocate` directive which will be used in GenASiS to replace CUDA/HIP memory allocation routines, thus making it more portable. Once the compiler implementations for task parallelism and offloading are mature enough, we hope that using the `loop` construct inside the `target` region will greatly simplify multi-level device parallelism and provide a way to target multiple GPU streams, giving better performance.

3.3. GESTS

Three-dimensional (3D) fluid turbulence with disorderly fluctuations in space and time are a grand challenge in science and computing. A powerful tool for advancing understanding is direct numerical simulation [19] based on exact conservation laws in a simplified domain amenable to Fourier pseudo-spectral methods of high accuracy. High resolution is crucial, especially for studies of localized regions of high intensity [20]. Despite heavy communication requirements, a recent algorithm [21] using CUDA Fortran on Summit has allowed problem sizes as large as 6 trillion grid points. The objective of the GESTS (GPUs for Extreme Scale Turbulence Simulations) code is to advance towards exascale, preferably in a portable manner. The main task in the GESTS code is to take 3D Fast Fourier transforms (FFTs). Assuming a fat-node architecture with large CPU memory, to reduce communication costs we use a one-dimensional (*slabs*) domain decomposition as shown in Fig. 6. Within each plane in a slab, 1D FFTs in two directions (here x and y) are performed readily using highly optimized GPU libraries (cuFFT or rocFFT), while the FFT in the third (z) direction requires an all-to-all global transpose that re-partitions the data into, say, $x - z$ planes. However, if N is very large (up to 18,432 in [21]), a complete slab may not fit into the smaller GPU memory. We address this by dividing each slab into np smaller *sub-volumes*, as in Fig. 6b. In effect, batches of data formed from the sub-volumes are copied to the GPU, computed on, and copied back, while operations on different portions may overlap with one another.

3.3.1. Challenges porting to OpenMP

In the "batched" scheme above, each sub-volume of data to be copied consists of N/P strips of size $N \times N/np$, which are strided from one another. Efficient strided data transfer is thus critical. A simple packing on the host prior to transfer to device, or performing multiple copies one strip at a time, would have required an extra data-reordering operation on the CPU and the overhead of numerous smaller copies respectively [21]. In CUDA Fortran, the API call, `cudaMemcpy2D`, can be used to perform efficient strided copies directly.

OpenMP 5.0 allows strided `UPDATE`s but not non-contiguous array `TARGET DATA MAP` for array accesses like a(n/2:n,1:n). This suggests strided `UPDATE`s can serve our needs only if the entire slab is *mapped* to the GPU, for which sufficient memory may not be available. One possibility is to *map* only a sub-volume (a smaller buffer) to the GPU and *update* it after copying the necessary data from the larger buffer to the smaller buffer on the host. Unfortunately, the extra operations entailed on the host reduces performance, even if OpenMP *threads* are used to share the workload. It may be better to use the OpenMP 4.5 device memory routine `omp_target_memcpy_rect`, which allows copying of a specified sub-volume inside a larger array on the host to a smaller buffer on the device. This OpenMP routine is conceptually similar to `cudaMemcpy2d`, although currently it is Fortran-callable only through a C-FORTRAN interface that requires careful consideration of input parameters accounting for differences between C and Fortran in array structures. However, tests on Summit show slow performance compared to CUDA Fortran; thus, further studies are required.

As noted earlier, batched operation enables asynchronism among operations on different sub-volumes in the same slab. In OpenMP, this asynchronism can be achieved using the `TASK` clause for work on the host, `NOWAIT` for device kernels and data copies, and `DEPEND` to enforce the necessary synchronization between different *tasks*. However, a complication arises when a `TASK` with a `DEPEND(OUT:a)` calls a non-blocking library, such as *cufft* or *rocfft*. The OpenMP runtime appears to consider the *task* as "completed" once the binding thread calls the function, without the kernel having finished, or even started, running on the device. Further *tasks* with a `DEPEND(IN:a)` may proceed prematurely, leading to incorrect results. This issue can likely be fixed by using the `DETACH` clause introduced in OpenMP 5.0.

3.3.2. Implementation strategy with OpenMP

Fig. 7 compares pseudo-code segments in CUDA Fortran with OpenMP. Buffers labeled as NEXT, CURR and PREV for different sub-volumes in a slab are allocated on the GPU, and different operations are performed on them asynchronously. In a single loop iteration, a strided host-to-device copy of the $(ip+1)$th sub-volume to the NEXT buffer, computations on the CURR buffer (which holds the ipth sub-volume on the device), a strided device-to-host copy of the PREV buffer to $(ip-1)$th sub-volume, and all-to-all from the host on the $(ip-2)$th sub-volume are performed asynchronously. In CUDA Fortran, *events* are used to record and synchronize operations on different *streams* to ensure correct results. Computations on the CURR buffer (line 8) do not start before copy of the CURR buffer completes, as enforced by a `cudaStreamWaitEvent` call on line 7. In OpenMP, the `DEPEND` clause with dependency type `IN` ensures the task does not start before prior tasks using the same dependency variable with type `OUT` are completed.

The OpenMP version differs from the CUDA Fortran version in two key aspects. First, `omp_target_memcpy_rect` is called in place of `cudaMemcpy2D`. Second, a `DETACH` clause with an *event* handle is attached to the `TASK` construct launching the FFTs. This *task* also calls `hipStreamAddCallback`, on line 9, to insert a *callback* function into the *stream* in which the FFTs will execute. The *callback* function is executed once the FFTs complete on the GPU, which calls *omp_fulfill_event* to indicate the completion of the event passed to the `DETACH` clause. This satisfies the `OUT` dependency and allows further tasks to execute.

```
do ip=1,np
   NEXT = mod(ip+1,3); CURR = mod(ip,3);
   PREV = mod(ip−1,3); COMM = mod(ip−2,3);
   cudaStreamWaitEvent (trans_stream, DtoH(NEXT), 0)
   cudaMemCpy2DAsync (abuf(NEXT),a(ip+1),trans_stream)
   cudaEventRecord (HtoD(NEXT),trans_stream)
   cudaStreamWaitEvent (comp_stream, HtoD(CURR), 0)
   FFTExecute (abuf(CURR), comp_stream)
   cudaEventRecord (comp(CURR), comp_stream)

   cudaStreamWaitEvent (trans_stream, comp(PREV), 0)
   cudaMemCpy2DAsync (snd(ip-1), abuf(PREV), trans_stream)
   cudaEventRecord (DtoH(PREV), trans_stream)
   cudaEventSynchronize (DtoH(COMM))
   MPI_IALLTOALL (snd(ip-2))
end do
```

```
do ip=1,np
   NEXT = mod(ip+1,3); CURR = mod(ip,3);
   PREV = mod(ip−1,3); COMM = mod(ip−2,3);
   TASK DEPEND (IN:DtoH(NEXT), OUT:HtoD(NEXT))
   omp_target_memcpy_rect (abuf(NEXT), a(ip+1))

   TASK DEPEND (IN:HtoD(CURR), OUT:comp(CURR))
    DETACH(event)
   FFTExecute (abuf(CURR), comp_stream)
   hipStreamAddCallback (comp_stream, callback, event, 0)
   TASK DEPEND (IN:comp(PREV), OUT:DtoH(PREV))
   omp_target_memcpy_rect (snd(ip−1), abuf(PREV))

   TASK DEPEND (IN:DtoH(COMM))
   MPI_IALLTOALL (snd(ip-2))
end do
```

Fig. 7. Pseudo-codes from GESTS showing batched asynchronous transforms in one direction using CUDA Fortran (on the left) and OpenMP (on the right). The colors red, blue and green highlight events or dependencies corresponding to the NEXT, CURR and PREV buffers or sub-volumes as shown in Fig. 6b. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.3.3. Plans for OpenMP 5.0

As of September 2020, basic non-batched synchronous versions of our 3D FFT kernels on Summit have been tested, with comparable performance between CUDA Fortran and OpenMP, up to $12{,}228^3$ resolution, as shown in Fig. 8. The performance observed at large problem sizes shows less than ideal weak scaling of the code as MPI communication costs dominate performance. A speedup of 2.57X is observed for 3D FFT at the $12{,}228^3$ problem size. Larger speedups are expected in the full DNS code, which has more computations that can benefit from GPU acceleration. However, the full promise of OpenMP offloading still rests upon solutions to the challenges noted above, associated with strided copies (even using `omp_target_memcpy_rect`) and asynchronism (with the `DETACH` clause not yet fully supported). Progress in the near term will involve working closely with vendor experts, ideally with improved Fortran support for OpenMP 5.0 and higher. We remain hopeful for ultimate success in a portable OpenMP implementation (say, with the Cray compiler and AMD GPUs) of the successful batched asynchronism CUDA Fortran code in [21].

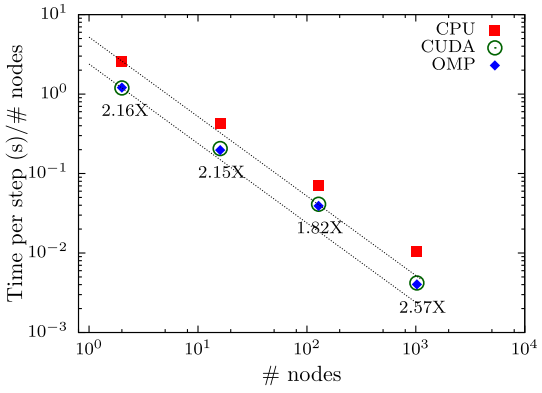

Fig. 8. Weak scaling performance of non-batched synchronous 3D FFT kernels on Summit. Performance of CUDA Fortran (green) and OpenMP (blue) versions are comparable. Speedup of OpenMP offload with respect to the CPU version is given as labels in the plot. Dashed lines indicate perfect weak scaling. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.4. GridMini

GridMini is a mini-application for Lattice Quantum Chromodynamics (QCD). Lattice QCD simulates the strong interactions of quarks and gluons on a four-dimensional discrete space–time grid, and provides crucial input to theoretical nuclear and particle physics. GridMini is a substantially reduced version of Grid [22], a new C++ lattice QCD library developed for highly parallel computer architectures. We use GridMini to assess whether OpenMP's target offloading can serve as a common portable solution across different GPU accelerators.

The main computational motif of lattice QCD is Markov Chain Monte Carlo simulations, with sparse matrix inversions. In both the Monte Carlo simulations and matrix inversions, the key computational kernel is the high-dimensional matrix–vector multiplication. LQCD calculations are memory bandwidth bound. For this reason, it is instructive for us to measure the sustained memory bandwidth with our code.

3.4.1. Implementation and optimization strategy

The following OpenMP implementation and performance evaluation are for the SU(3) matrix–matrix multiplication benchmark in GridMini, `Benchmark_su3`, to measure sustained memory bandwidth on the target device. It computes `z=x*y` many times, where `x`, `y` and `z` are all arrays of SU(3) matrices, and the bandwidth is calculated as the memory footprint divided by time spent in the calculation.

```
#define accelerator_for ( iterator ,num,nsimd, ... ) \
{                                                    \
        _Pragma("omp target teams distribute parallel for
 num_teams(nteams) thread_limit ( gpu_threads )") \
        naked_for( iterator , num, { __VA_ARGS__ }); \
 }
```

Fig. 9. C++ macros that define the loop-level computation in GridMini.

The loop-level computation over the arrays is done through an `accelerator_for` macro as defined in Fig. 9. In order to ensure that the functions are offloaded properly, all the potential inline functions are decorated with the `accelerator_inline` macro, which adds the `always_inline` attribute to the function definitions. These macros are consolidated in a header file, and will expand to different implementations/definitions for different compilers/architectures with `#ifdef` switches.

To simplify memory management for the deeply nested data structures in GridMini, we rely on unified virtual memory through explicit calls to `cudaMallocManaged()`.

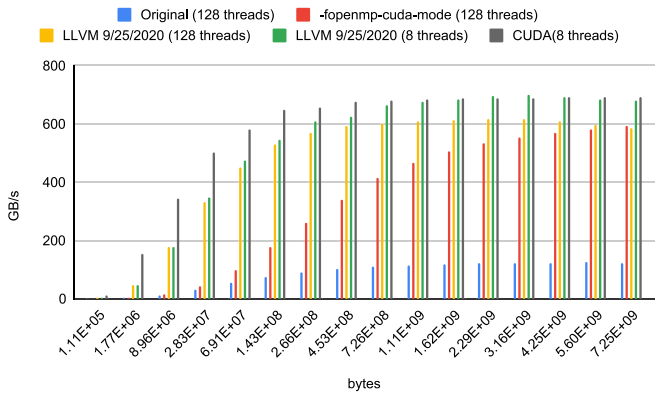

Fig. 10. Performance comparison of `Benchmark_su3` in GridMini among different LLVM versions and CUDA. Results labeled `Original` and `-fopenmp-cuda-mode` used the LLVM GitHub version dated 07/31/2020, `LLVM 09/25/2020` refers to the LLVM trunk version dated 09/25/2020, and `CUDA` uses cuda/10.1.243.

3.4.2. Performance evaluation

With the above implementation, we were able to run `Benchmark_su3` on NVIDIA GPUs after compiling it with LLVM/Clang. We ran into several issues with the above OpenMP offloading which have since been fixed in the LLVM/Clang compiler. Initially, the results were incorrect with `z` always returning 0. Upon further investigation, we found that the issue was related to the use of `struct` with short vectors (which is necessary for the vector layout in Grid). When a device function in the `target` region returned a value of this type, it was not copied back correctly. Next, we only obtained 125 GB/s on NVIDIA V100 GPUs on Cori, while the CUDA version achieved more than 600 GB/s sustained memory bandwidth. OpenMP performance dramatically improved when we compiled the code with an additional `-fopenmp-cuda-mode` option. The version of LLVM/Clang we used dated 07/31/2020 directly from the GitHub repository still gave much worse results than CUDA with small memory footprints due to frequent allocation and deallocation of short-lived data objects, which are expensive on GPUs. This was fixed by a patch (now in the main trunk of LLVM) that optimizes GPU allocations. The progress in performance of `Benchmark_su3` is shown in Fig. 10, where LLVM OpenMP is now comparable to CUDA for moderate to large memory footprints, but lags behind for small memory footprints, indicating possible overheads in the OpenMP runtime.

3.4.3. Plans for OpenMP 5.0

Since we are currently using `cudaMallocManaged` to explicitly manage memory, the code is not portable to other devices such as AMD or Intel GPUs. The most important OpenMP 5.0 feature to us is native support for unified shared memory. Once this feature is fully supported, we plan to test the performance and portability of our code on other architectures.

3.5. LSMS

LSMS (Locally Selfconsistent Multiple-Scattering) [23,24] is a first-principles density functional theory code for condensed matter and material sciences. It calculates the behavior of materials by solving the electron states in the materials in the framework of density functional theory [25,26] and solves the resulting Kohn–Sham equations using multiple scattering theory, also known as the Korringa–Kohn–Rostocker (KKR) method [27,28]. It solves the multiple scattering problem in real space and uses an approximation based on a finite distance cutoff for the scattering path of an electron before returning to its origin (Local Interaction Zone, LIZ). The solutions for atom-centered LIZs are combined to form the solution for the whole system, thus the computational effort scales linearly with the total number of atoms, as opposed to the usual cubic scaling of traditional methods to solve the Kohn–Sham equations.

LSMS calculations are dominated by complex dense linear algebra. The work is distributed at three parallelization levels that reflects the mapping of the physical subdivision of the LSMS methods onto the underlying computation. LSMS utilizes MPI at the highest level and uses OpenMP within each MPI rank for CPU multi-threaded parallelism. Each MPI rank can also utilize one accelerator. The calculation of electron densities within LSMS are naturally divided into the accumulation of contributions from each atom. Thus, the system is spatially subdivided into groups of atoms of approximately equal size. The LIZs (O(100) atoms/LIZ) define the template for the point-to-point communication to exchange the scattering matrices (approx. 500kB/atom) between the MPI ranks.

The on-node calculations for each atom local to the MPI rank are further parallelized using OpenMP on the CPU and are divided into a number of steps separated by MPI communication. The first step is the calculation of the scattering matrices which result from the numerical integration of systems of coupled ordinary differential equations on a discrete linear grid for a small set of initial values. After the communication of remote single scatterer matrices, these are combined based on the geometry of the physical system to form the KKR matrix, described in further detail below. This routine is the second most time-consuming operation. The scattering path matrix calculation is the most computationally costly part of LSMS, being responsible for over 9/10 of the floating point operations in a typical run. This calculation is accomplished by the inversion of the dense, complex, non-Hermitian KKR matrix. As further calculations only require a small (typically 32×32) diagonal block of the inverse matrix that has a dimension that is more than $100\times$ that block's size, the inversion is usually performed using an algorithm based on the Schur complement [24].

3.5.1. Parallelization with OpenMP

Performance portability is a central goal of LSMS parallelization. It will run on several different architectures and compilers as part of the SPEC HPC 2020 benchmark suite. This limits the ability to rely on optimized libraries for each architecture to perform many of the needed operations. OpenMP allows us to target several architectures while pushing compiler optimizations to the limit. This can be used to evaluate the performance gap between OpenMP and a library-based approach.

The KKR matrix needs to be built and solved with a standard matrix solver. This is done using the *getrf* and *getrs* routines from LAPACK. Since we must avoid relying on external packages, we had to find an alternative to the standard LAPACK and BLAS libraries. FLENS is a header-only library implementation of LAPACK routines with a high-level interface in C++. FLENS also offers a pure C++ implementation of the BLAS routines compiled along with FLENS called ulmBLAS.

Converting FLENS to run on the device required modifying the necessary BLAS routines in ulmBLAS to use OpenMP target offload. Some changes were made to improve GPU performance also. For many routines, this was as simple as applying a loop-level parallelism directive to the function. More complex level-3 routines such as the ubiquitous *gemm* operation required extensive changes. Others such as *iamax* could be efficiently offloaded with minor modifications, shown in Fig. 11. After the BLAS routines were parallelized, only a few adjustments had to be made to the FLENS LAPACK library. References to data on the device were updated from the device using the OpenMP update clause. Using the library now only required the data pointers to be mapped on the device via the same high level interface as shown in Fig. 12.

We ran a small benchmark on a single NVIDIA Volta GPU to compare the individual BLAS routines with a library implementation. Results were averaged over several tests. The performance of FLENS under OpenMP offload and using the *zgemm* and *izamax* BLAS routines

```
#pragma omp declare reduction (iamax :                                  \
      Pair<int, std::complex<double>> :                                 \
      omp_out = omp_in.val > omp_out.val ? omp_in : omp_out)            \
       initializer (omp_priv={-1, -1})
#pragma omp target teams distribute parallel for                        \
      reduction (iamax:absMax)
for (int i = 0; i < n; i++) {
      if (abs1(x[i*incX]) > absMax.val) {
           absMax.idx = i;
           absMax.val = abs1(x[i*incX]);
      }
}
return absMax.idx;
```

Fig. 11. Offloading the level 1 BLAS routine iamax which finds the index of the maximum value.

```
#pragma omp target data map(to:A[0:M*N]) map(tofrom:B[0:N])
{
     ZGeMatrix::View f_A = ZGeMatrix::EngineView(M, N, A, N, 1);
     ZGeVector::View f_B = ZGeVector::EngineView(N, B);
     IGeVector::View f_P = IGeVector::EngineView(N, piv);

     flens::lapack::trf(f_A, f_P);

     flens::lapack::trs(flens::NoTrans, f_A, f_P, f_B);
}
```

Fig. 12. Offloading matrix solvers once the BLAS routines were converted to use OpenMP offloading. Solves an $M \times N$ matrix A.

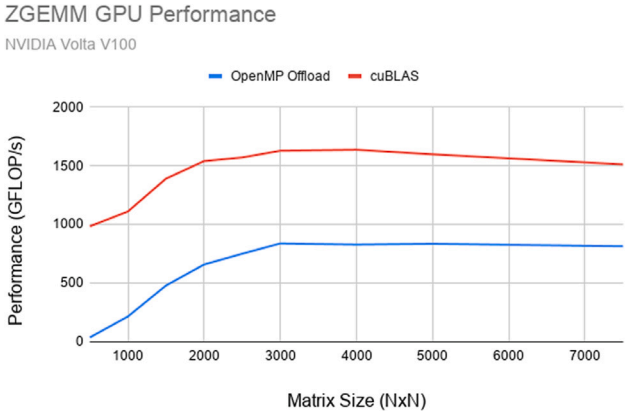

Fig. 13. Performance of selected BLAS routines implemented in FLENS using OpenMP offloading on a single Volta GPU versus cuBLAS performance.

is shown in Fig. 13. There is a noticeable performance gap. More compiler work is needed to optimize OpenMP offloading in order to complete with manually tuned libraries. In LSMS, cuBLAS was 2x faster than the FLENS OpenMP offload version.

3.5.2. Challenges

One of the initial challenges faced when moving LSMS to a device with OpenMP was the fact that not all compilers support complex numbers in target offload regions. Clang recently added such support for complex numbers and will soon support complex math functions.

3.5.3. Plans for OpenMP 5.0

OpenMP 5.0 introduced the allocator clause that allows the user to allocate specialized memory on the target device. The use of shared memory on the GPU is critical for optimized matrix–matrix multiplies using OpenMP 5.0. This can be done with per-team memory allocation.

```
#pragma omp parallel for  // high-level CPU threads
for(crowd : population) {  // over walker crowds
  #pragma omp target teams distribute \
    num_teams(crowd_size) // accelerator thread teams
  for(walker : crowd) {
    #pragma omp parallel for simd
      // accelerator threads/vector unit
    for(electron : walker electrons)
    { // some basic per electron computation }
  }
}
```

Fig. 14. QMCPACK multi-level parallelism.

3.6. QMCPACK

QMCPACK is a high-performance open-source Quantum Monte Carlo (QMC) simulation code [29,30]. It performs electronic structure calculations using a number of related highly accurate QMC algorithms. QMCPACK is written in C++ with extensive use of generic programming. For parallelization, it utilizes a hybrid MPI + (OpenMP, CUDA) approach to optimize memory usage and fully uses the growing number of cores or GPUs per node. High computational efficiencies are achievable, e.g. Ref. [31].

In QMC methods, wavefunctions are sampled by multiple Markov chains or "walkers". The walkers are updated using Metropolis-like algorithms and are loosely coupled to each other. A large population of walkers enables massive parallelism. Within a walker, compute kernels typically contain loops over electrons which are well-suited for fine level parallelism. QMCPACK distributes walkers first among MPI ranks and then among CPU threads. On GPU accelerators all the walkers held by a thread are advanced in lock-step for efficiency. Due to the random nature of the algorithms and slight workload imbalances that result, QMCPACK relies on multiple CPU threads independently offloading computation to GPUs asynchronously to maintain high occupancy.

3.6.1. Implementation and optimization strategy

QMCPACK targets maximal efficiency at 1 MPI rank per CPU socket or GPU. The coarse and fine level parallelism of QMCPACK can be abstracted as the pseudo code in Fig. 14. The parallelism leverages OpenMP threading and GPU offloading capabilities simultaneously.

3.6.2. Evaluation

Our study is based on the LLVM Clang compiler 11.0.0, although the features needed may be fully or partially supported by other compilers. e.g. OpenMP offload to NVIDIA GPUs is currently supported by IBM XL, Cray Clang, LLVM Clang and GNU GCC compilers. For simplicity, this study uses a mini-app, miniQMC, instead of the full QMCPACK application.

A typical OpenMP target region contains a compute kernel and host-to-device and device-to-host data transfers before and after the computation. The LLVM compiler front-end and OpenMP runtime transform the `target` construct with `map` clauses into `cuMemcpyHtoD`, `cuLaunchKernel`, `cuMemcpyDtoH` and `cuStreamSynchronize` on a single CUDA stream, see Fig. 15. Host arrays are pre-registered by `cudaHostRegister` for best transfer performance.

With an efficient single offload region implementation, multiple host threads are employed for independent offload. When compute kernels are small, NVIDIA GPUs may execute kernels from different streams concurrently and overlap them with data transfer operations. Fig. 16 shows 8 host threads offloading computation to the GPU via 8 streams. Support for these key features enables QMCPACK to efficiently leverage OpenMP.

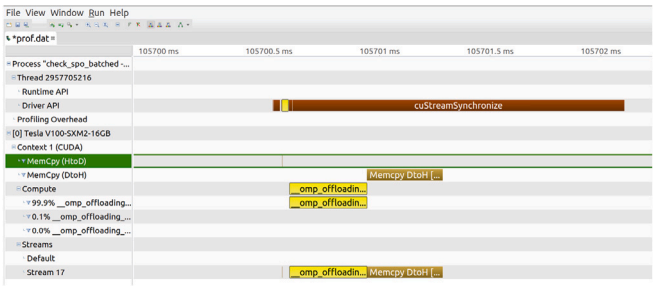

Fig. 15. Single OpenMP target offload region on a CUDA stream with minimal synchronization.

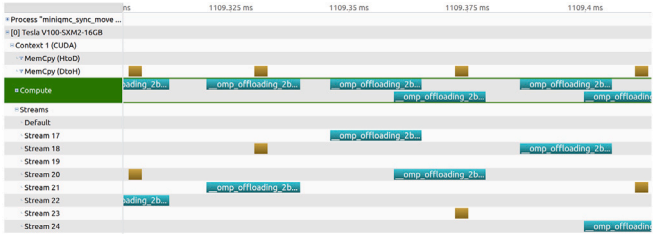

Fig. 16. Host OpenMP threads offload their own computation via independent CUDA streams in miniQMC. Time tracing by the NVIDIA Visual Profiler.

3.6.3. Challenges and developments

QMCPACK developers teamed up with the SOLLVE team to improve the quality of the Clang compiler and its runtime library to address features that are critical for production use of OpenMP in real applications. The following capabilities were added during the development of Clang 11: (1) math functions and complex algebra inside OpenMP offload regions (2) interoperability between offload regions and CUDA by exchanging memory pointers (needed for calling vendor linear algebra libraries) (3) optimized memory mapping lookup in the OpenMP offload runtime library (needed for applications that are sensitive to OpenMP runtime overhead). The remaining challenges using LLVM are (1) not being able to link static archives with device code (applications are forced to directly link all their object files); and (2) using `target nowait` without blocking the thread that launches the compute kernel is still in development.

3.6.4. Plans for OpenMP 5.0

As more OpenMP 5.0 features are enabled by compilers, QMCPACK will take advantage of (a) detached tasks for interoperability with other asynchronous runtime libraries; (b) the metadirective to reduce source code duplication and macros; (c) variant functions for clean specialization with high performance; and (d) tools extension for debugging and profiling.

3.7. PLASMA and SLATE libraries

PLASMA and SLATE are numerical linear algebra libraries that heavily rely on OpenMP to express runtime dependence between linear algebra kernels. They represent basic on-core or on-device units of work that were extracted from BLAS and LAPACK code to form a task-based representation of common linear algebra functions. These include one-sided factorizations (Cholesky, LU, QR, and LDLT), two-sided decompositions (eigenvalue, Schur form, and SVD), and the corresponding solvers and linear minimizers such as least-squares solve.

The prevailing motifs, in broad terms, are three kinds of kernels for dense linear algebra in the PLASMA and SLATE libraries: compute-bound, memory-bound, and communication-bound. The first kind are often encoded as a collection of loop nests of up to 3 levels that benefit from the volume-to-surface effect: computational complexity is higher order than storage complexity. While the majority of the loop nests are affine, some are not. The second kind of kernel performs data layout translations that drastically reduce adverse cache effects in computational kernels. Finally, the third kind of kernels encompass functions that perform any kind of data communication between host or device memories, either on-node or between nodes.

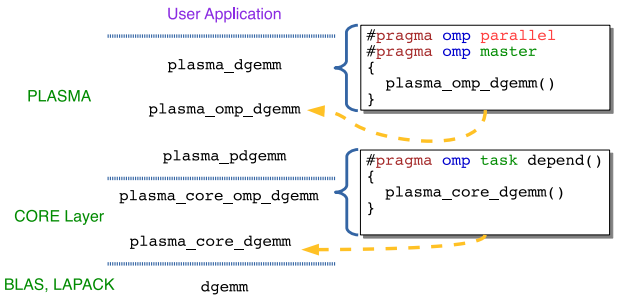

Fig. 17. Hierarchy of functions in PLASMA that represents various entry points into the libraries functionality.

The initial definition of tasking in OpenMP 3.0 was not sufficient to accommodate all these needs as the sibling tasks may be regarded as similar to the bulk-synchronous parallelization that is often applied in vendor libraries and in ATLAS. Instead, PLASMA and SLATE utilize the parallel region's task set and all of its sibling tasks spawned at runtime to match the computational need of the user problem. The number of tasks is increased for larger user problems and the tasks rely on the depend clause for runtime dataflow scheduling of the OpenMP implementation.

3.7.1. Implementation and optimization strategy

The task decomposition of the computational graph takes full advantage of cache or device residency of the user data that is decomposed in tiles, sets of tiles, and submatrices. The vast majority of the factorization routines are based on panel-update iteration while the decompositions often rely on panel-left–right-update scheme. The tile-based algorithms add the two-stage approach to the computational patterns that localizes data access and exposed much higher levels of parallelism to take advantage of increasing computing capability of modern multicore and accelerator devices. The use of MPI may introduce much higher overhead than is usually experienced on single-node codes. The generic approach of dealing with this is to employ a lookahead technique to overlap slow operations behind the fast ones.

3.7.2. Example of OpenMP code

The easiest representation is to look at the hierarchy of routines. Fig. 17 shows the three major layers of routines: (1) the main PLASMA entry points, (2) the CPU-core layer, and (3) the low-level legacy BLAS and LAPACK layer. There are three types of routines in the top-most layer: plasma_dgemm() is an entry point from a sequential application outside of any OpenMP region, plasma_omp_dgemm() is a sequential entry point from inside an active OpenMP region, either statically or dynamically scoped, and plasma_pdgemm() is a parallel entry point from within an active OpenMP region. In the middle layer, there are two types of routines: plasma_core_omp_dgemm() is a parallel entry point used from outside of the current tasks set in order to create a new task and specify its dependence set and plasma_core_dgemm() implements the task functionality and does not use any OpenMP pragmas. Finally, at the bottom-most layer, PLASMA makes calls to low-level implementations of BLAS and LAPACK subroutines such as Fortran's API dgemm_(), CBLAS API cblas_dgemm(), or LAPACKE API lapacke_dgetrf() – they can come from the reference Netlib implementation, open-source implementations in ATLAS or OpenBLAS, or vendor libraries such as Intel MKL currently available as oneMKL.

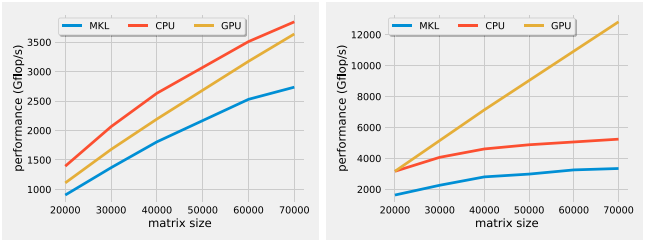

Fig. 18. Comparison of SLATE's performance on 4 nodes of the NERSC's supercomputer Cori against Intel's MKL ScaLAPACK implementation for LU factorization PDGETRF (left) and QR factorization PDGEQRF (right) in IEEE 64-bit floating-point precision.

3.7.3. Evaluation

Fig. 18 shows comparison of performance on 4 nodes of the Cori supercomputer located at NERSC between SLATE's and Intel's MKL ScaLAPACK implementation of LU factorization with partial pivoting (PDGETRF) and QR factorization (PDGEQRF), both in IEEE 64-bit floating-point precision and using CPU-only or the mixed CPU/GPU modes (the latter only available in SLATE). Note that each Cori node features two sockets each with Intel Xeon Skylake 6148 processor (20 cores and 40 hardware threads) and 8 NVIDIA Volta V100 GPUs. The interconnect cards are 4-way dual-port InfiniBand EDR by Mellanox. The MKL version was 2020.0.166, CUDA: 10.2.89, MPI: Open MPI 4.0.3. The SLATE code was from June 22, 2020 and was compiled with GCC compiler version 8.3.

3.7.4. Challenges and their resolution

Unlike PLASMA, SLATE routines require inter-node communication that is performed with MPI. However, the complexity of the modern hardware stack requires SLATE to employ a more complex interaction of software components than the often quoted MPI+OpenMP moniker. In fact, SLATE routines use MPI processes that spawn data-dependent OpenMP tasks that in turn invoke MPI functions to communicate across node boundaries which results in MPI+OpenMP+MPI with the majority of the MPI calls relying on a multi-threaded MPI implementation. One promising direction is to use an MPI implementation that is aware of OpenMP task implementations by, for example, combining a ULT-aware MPI implementation [32] with the BOLT OpenMP runtime [3], which employs ULT-based OpenMP tasks.

Unlike PLASMA which is 100% OpenMP-based, the SLATE implementation is based on the CUDA and HIP interface: the CPU part handles latency-bound compute and message-passing while the GPU part performs data transfers and a handful of calls to numerical kernels. Porting to Intel Xe Ponte Vechccio accelerators with OpenMP target offload became the natural choice to bring SLATE to the platform to make it 100% OpenMP.

3.7.5. Plans for OpenMP 5.0

OpenMP 5.0 offers new functionality, especially in the form of new clauses, that were originally implemented inside the QUARK runtime scheduling library. In particular, these include: specification of affinity of task and data for more efficient cache mapping; data-dependence iterators for multiple dependences between tasks that are not known until runtime and may differ between problem sizes. Finally, the extension to the offload support will offer less reliance on vendor-specific APIs.

3.8. RAJA

RAJA [33] is a cross-platform heterogeneous programming abstraction library in C++ developed by LLNL to help make scientific applications more portable and future-proof. It provides a model for loop-centric parallel programming that abstracts over OpenMP, TBB,

```
1  ReduceSum<double> sum_red;
2  forall <my_policy> (
3          RangeSegment(0,N),
4          [=] (int i) {
5    a[i] += c * b[i];
6    sum_red+=a[i];
7  });
8  double sum = sum_red.get();
```

(a) A basic RAJA daxpy loop with a sum reduction added

```
1  ReduceSum<double> sum_red;
2  forall <my_policy> (
3          RangeSegment(0,N),
4          [=] (int i, double&
   red) {
5    a[i] += c * b[i];
6    red+=a[i];
7  }, sum_red);
8  double sum = sum_red.get();
```

(b) RAJA daxpy using experimental OpenMP-friendly reduction

```
1  template <typename It, typename Fn>
2  RAJA_INLINE void forall_impl( const
       omp_target_parallel_for_exec_nt &, It&& iter, Fn&&
   loop_body) {
3    // ... prep body for offload and execute pre-plugins
4    auto i = distance_it ; auto N = iter.size(); using
   std::begin;
5    auto &begin_it = begin( iter );
6    #pragma omp target teams distribute parallel for map(to :
   body, begin_it ) // or firstprivate (body, begin_it )
7    for (i = 0; i < N; ++i) {
8      Body ib = body;
9      ib( begin_it[i]);
10   }
11   // ... execute post plugins
12 }
```

(c) A basic RAJA forall implementation for OpenMP offload loop

Fig. 19. Examples of RAJA usage and internals.

CUDA and HIP; more backends are in progress. It has some unique requirements that have driven some decisions in OpenMP's design for C++, especially helping support the construction of C++ abstractions over OpenMP.

A simple daxpy operation with an additional sum of the updated elements of a using a RAJA `forall` is presented in Listing 19(a). RAJA loops are written as function calls, taking a range of indices, or data elements, and a lambda to execute as the body of the loop. To run this on the host in an OpenMP `parallel for` loop, the user can define `my_policy` as `omp_parallel_for_exec`, or for an offload loop as `omp_target_parallel_for_exec`. Otherwise, the code for host or device execution is identical, and can easily be switched between the two or even multi-versioned by the user. This model provides advantages in portability and flexibility for user code, but it imposes significant constraints on the implementation of RAJA itself, and that is what we focus on in this section.

3.8.1. Implementation

The implementation of a RAJA backend for OpenMP has two major parts: (1) an OpenMP host backend that serves as the standard parallel host backend for RAJA, which has been tested and optimized extensively, (2) the OpenMP offload backend, which is almost entirely separate and has proven substantially more complicated to make both portable and integrate into an abstraction with the other models available. A minimal implementation of RAJA's offloading `forall` is given in Listing 19(c).

The offloaded loop shown is comparatively simple. Because of the abstraction, the loop is always over a range of integral values and is offloaded with all parallelism active. The challenges come from how to handle data in a consistent way through both RAJA and OpenMP. Note that in Line 14, we map the callable object passed in by the user onto the device. Different implementations of OpenMP handle pointers captured into lambdas differently, and users want to use differently managed pointers. The RAJA library does not handle data management,

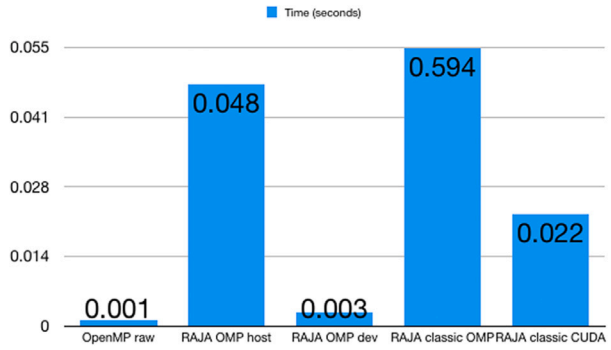

Fig. 20. Comparing reduction approaches on host and device with OpenMP.

but the RAJA Suite includes the Umpire and CHAI libraries [34,35] to help with this though it does not require their use. A user may decide to allocate their memory with `omp_target_alloc()`, or map memory that they allocated on the host, or even use unified memory if it is available.

When a lambda is mapped in OpenMP, all captured pointers are mapped and either translated or preserved. In principle, this means we should always be able to map the lambda and get correct behavior if the user's code is correct. In practice, we find that several implementations either fail to map the captures, allowing device addresses to work but causing mapped buffers to fail, or map them but on failing to find the corresponding storage on the device NULL the pointer, as would happen in a normal implicit map on a target region. This inconsistency means that for some compilers, RAJA uses the `map` clause for the lambda, others the `firstprivate` clause.

3.8.2. Optimization and evaluation

The other challenge has been supporting RAJA reductions. As shown in Listing 19(a), a reduction in RAJA is a separately declared variable captured by the lambda, and used in the body of the loop as though it were sequential. RAJA uses C++ copy construction and destruction to generate a reduction tree implicitly. This design helps keep user code as similar to the original as possible, making reductions blend in with loops almost like in serial code. However, it restricts RAJA from taking advantage of the OpenMP compiler's reduction facilities to implement RAJA's reductions. As part of the SOLLVE effort, we explored a new interface that would express the example in Listing 19(a) as in Listing 19(b). While they use similar types, passing the reduction through the `forall` arguments and passing a bare double reference to the body makes it possible to use OpenMP reductions underneath and have only a single reduction tree rather than one per variable being reduced. We produced a micro-benchmark to test the performance of reducing a large number of values into three variables in one loop with each of a raw OpenMP loop, RAJA OpenMP on the host, RAJA OpenMP offload with the new interface, RAJA's classic OpenMP offload reductions, and finally a RAJA CUDA reduction running on the same device as the OpenMP offload versions.

The results are presented in Fig. 20. Using our original reduction interface, the RAJA classic OMP item, the reduction kernel takes an extremely long time, more than 10× longer than on the host and more than 25× longer than the same interface implemented in CUDA. This proved extremely difficult to optimize, as it causes all implementations we tested to use their heavyweight runtime and generally deoptimized the kernel to the point that the cost was universally high. With the experimental interface however, the RAJA OMP dev column, the reduction kernel was 7× faster than even the highly optimized CUDA implementation. Adoption of the experimental interface into the RAJA API is a work in progress and likely will become the new standard way to perform reductions with RAJA given the substantial benefits found as part of this process.

3.8.3. Next generation OpenMP plans

Both OpenMP 5.0 and 5.1 have features designed to better support C++ in OpenMP offload. A major update that should help us implement our abstraction, especially, e.g., the old-style reductions and smart-pointers for memory management, is `declare variant` and `begin/end declare variant` allowing code specialization on a per-device or per-context basis. There are also a number of clarifications to the way `is_device_ptr` and similar clauses work to help deal with the data management issues mentioned above.

4. Conclusions and future work

The experiences that are described here to port applications to GPUs using OpenMP provided many insights and some directions for future work. All these applications and mini-apps written in C++ and Fortran were successful in their use of OpenMP offload features on different parts of the codes.

Applications needed significant restructuring to take advantage of the offload model for exploiting GPUs. For example, they had to make sure there is enough work to offload to the GPUs and that most of the computation is data parallel, e.g., they had to avoid conditionals or load imbalances across threads. Another big challenge in porting the applications with OpenMP is to map their data structures to the GPUs efficiently. Applications that mapped their data structures successfully in this study created high-level data structure abstractions to do so and used device runtime library routines or 'target enter/exit data' constructs to create storage or container classes in both Fortran, e.g., in GenASiS, and C++, e.g., in QMCPACK and GridMini. Once the restructuring is complete, additional code transformations were often necessary in order to improve the performance of the OpenMP version, such as tuning the number of threads per block, or inlining code.

OpenMP implementations have been improving at a fast rate, and the advancements resulted in higher application performance. The impact of compiler optimizations for the GPU, e.g., addressing memory management and kernel code optimizations, were important for application performance. In some cases, the offload support in OpenMP implementations achieved better performance than CUDA. OpenMP asynchronous target and tasking provides interoperability with asynchronous libraries, e.g., CUDA, HIP. The GESTS application used tasking to coordinate work between target regions and asynchronous CUDA/HIP libraries using the OpenMP 5.0 task detach feature to notify a task via a callback that dependences can be fulfilled when an asynchronous library completes its execution. Some applications, e.g., PLASMA and SLATE numerical libraries, continue using OpenMP to maintain performance-portable and load-balanced code using tasking with dependence tracking. Combining these established features with asynchronous operation and accelerator offload will extend these libraries' numerical facilities to the fully GPU-resident execution with near-native efficiency. Furthermore, the BOLT runtime has helped to enhance tasking performance and interoperability between MPI and OpenMP. QMCPACK took advantage of having OpenMP threads offload their own computation through independent CUDA streams; this control of CUDA streams through OpenMP was critical to attain high performance of their application on GPUs.

Nevertheless, there are some challenges that need to be addressed. One such challenge is that the status of OpenMP implementations [36] coupled with OpenMP application experiences shows the OpenMP implementations need to be tested on a larger variety of platforms. Another challenge is making more of the critical new OpenMP 5.0/5.1 features available to application teams in vendor OpenMP implementations, e.g., making the task detach and interop feature to query GPU streams available for QMCPACK; generally, features for memory management, tasking, and better language, e.g., C++, interfaces are needed for OpenMP applications and implementations to be successful.

At the time of writing, OpenMP 5.1 is about to be released, and work has begun on OpenMP 6.0. Guided by these and other application experiences, the SOLLVE team will continue to contribute to the

OpenMP specification and its implementation in LLVM. The SOLLVE fork of LLVM OpenMP as well as the main trunk of LLVM OpenMP is under active development by SOLLVE team members, and a lot of progress was made in the last 12 months. We plan to continue to work closely with applications, in particular ECP applications, to uncover and address areas for improvement. Finally, we will work with vendors to help them adopt SOLLVE's LLVM work into their own OpenMP implementations and to help test their products on multiple platforms.

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 funded in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, in particular its subproject on Scaling OpenMP with LLVM for Exascale performance and portability (SOLLVE). This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy 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). The development of some of the numerical software libraries tested in this work was supported by the National Science Foundation under OAC grant No. 2004541. This work was supported in part by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

References

[1] Openmp 5.0 reference guide, 2021, https://www.openmp.org/wp-content/uploads/OpenMPRef-5.0-1119-01-TSK-web.pdf.
[2] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998).
[3] S. Iwasaki, A. Amer, K. Taura, S. Seo, P. Balaji, BOLT: Optimizing OpenMP parallel regions with user-level threads, in: 2019 28th International Conference on Parallel Architectures and Compilation Techniques, PACT '19, 2019, pp. 29–42.
[4] J. Schuchart, C. Niethammer, J. Garcia, Fibers are not (P)threads: The case for loose coupling of asynchronous programming models and MPI through continuations, in: 27th European MPI Users' Group Meeting, EuroMPI/USA '20, 2020, pp. 39–50.
[5] The LLVM compiler infrastructure, 2020, http://llvm.org/.
[6] LLVM Developers, LLVM/Clang openmp support, 2020, https://clang.llvm.org/docs/OpenMPSupport.html.
[7] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A. Nguyen, S. Su, T.L. Windus, M. Dupuis, J.A. Montgomery, General atomic and molecular electronic structure system, J. Comput. Chem. 14 (11) (1993) 1347–1363.
[8] M.S. Gordon, M.W. Schmidt, Advances in electronic structure theory: GAMESS a decade later, in: C.E. Dykstra, G. Frenking, K.S. Kim, G.E. Scuseria (Eds.), Theory and Applications of Computational Chemistry, Elsevier, Amsterdam, 2005, pp. 1167–1189, (Chapter 41).
[9] V. Mironov, A. Moskovsky, M. D'Mello, Y. Alexeev, An efficient MPI/OpenMP parallelization of the Hartree-Fock-Roothaan method for the first generation of Intel(R) Xeon Phi(TM) processor architecture, Int. J. High Perform. Comput. Appl. 33 (1) (2019) 212–224.
[10] B.Q. Pham, M.S. Gordon, Hybrid distributed/shared memory model for the RI-MP2 method in the fragment molecular orbital framework, J. Chem. Theory Comput. 15 (10) (2019) 5252–5258, PMID: 31509402.
[11] J. Kwack, C. Bertoni, B. Pham, J. Larkin, Performance of the RI-MP2 fortran kernel of GAMESS on GPUs via directive-based offloading with math libraries, in: S. Wienke, S. Bhalachandra (Eds.), Accelerator Programming using Directives, WACCPD 2019, in: LNCS 12017, Springer International Publishing, Cham., ISBN: 978-3-030-49943-3, 2020, pp. 91–113, http://dx.doi.org/10.1007/978-3-030-49943-3_5.
[12] E. Endeve, C. Cardall, R. Budiardja, A. Mezzacappa, Generation of magnetic fields by the stationary accretion shock instability, Agron. J. 713 (2010) 1219–1243.
[13] E. Endeve, C.Y. Cardall, R.D. Budiardja, S.W. Beck, A. Bejnood, R.J. Toedte, A. Mezzacappa, J.M. Blondin, Turbulent magnetic field amplification from spiral SASI modes: Implications for core-collapse supernovae and proto-neutron star magnetization, Agron. J. 751 (1) (2012) 26.
[14] C.Y. Cardall, R.D. Budiardja, Stochasticity and efficiency in simplified models of core-collapse supernova explosions, Astrophys. J. Lett. 813 (2015) L6.
[15] C.Y. Cardall, R.D. Budiardja, GenASiS basics: Object-oriented utilitarian functionality for large-scale physics simulations, Comput. Phys. Comm. (ISSN: 0010-4655) 196 (2015) 506–534.
[16] C.Y. Cardall, R.D. Budiardja, GenASiS Basics: Object-oriented utilitarian functionality for large-scale physics simulations (Version 2), Comput. Phys. Comm. (ISSN: 0010-4655) 214 (2017) 247–248.
[17] R.D. Budiardja, C.Y. Cardall, GenASiS Basics: Object-oriented utilitarian functionality for large-scale physics simulations (Version 3), Comput. Phys. Comm. (ISSN: 0010-4655) 244 (2019) 483–486.
[18] C.Y. Cardall, R.D. Budiardja, GenASiS Mathematics : Object-oriented manifolds, operations, and solvers for large-scale physics simulations, Comput. Phys. Comm. (ISSN: 0010-4655) 222 (2018) 384–412.
[19] T. Ishihara, T. Gotoh, Y. Kaneda, Study of high Reynolds number isotropic turbulence by direct numerical simulations, Annu. Rev. Fluid Mech. 41 (2009) 165–180.
[20] P. Yeung, X. Zhai, K. Sreenivasan, Extreme events in computational turbulence, Proc. Natl. Acad. Sci. 112 (2015) 12633–12638.
[21] K. Ravikumar, D. Appelhans, P. Yeung, GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism, in: Proceedings of the International Conference for High Performance Computing, Networking and Storage Analysis, SC'19, Denver, CO, USA, ACM, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3295500.3356209.
[22] P.A. Boyle, G. Cossu, A. Yamaguchi, A. Portelli, Grid: A next generation data parallel C++ QCD library, PoS LATTICE2015 (2016) 023, http://dx.doi.org/10.22323/1.251.0023.
[23] Y. Wang, G.M. Stocks, W. Shelton, D. Nicholson, W. Temmerman, Z. Szotek, Order-N multiple scattering approach to electronic structure calculations, Phys. Rev. Lett. 75 (1995) 2867.
[24] M. Eisenbach, J. Larkin, J. Lutjens, S. Rennich, J.H. Rogers, GPU acceleration of the locally selfconsistent multiple scattering code for first principles calculation of the ground state and statistical physics of materials, Comput. Phys. Comm. 211 (2017) 2–7.
[25] P. Hohenberg, W. Kohn, Inhomogeneous electron gas, Phys. Rev. 136 (1964) B864–B871.
[26] W. Kohn, L. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. 140 (1965) A1133–A1138.
[27] J. Korringa, On the calculation of the energy of a Bloch wave in a metal, Physica 13 (1947) 392–400.
[28] W. Kohn, N. Rostoker, Solution of the Schrödinger equation in periodic lattices with an application to metallic lithium, Phys. Rev. 94 (1954) 1111–1120.
[29] J. Kim, et al., QMCPACK: an open source ab initio quantum Monte Carlo package for the electronic structure of atoms, molecules and solids, J. Phys.: Condens. Matter 30 (19) (2018) 195901.
[30] P.R. Kent, et al., QMCPACK: Advances in the development, efficiency, and application of auxiliary field and real-space variational and diffusion quantum Monte Carlo, J. Chem. Phys. 152 (17) (2020) 174105.
[31] A. Mathuriya, Y. Luo, R.C. Clay III, A. Benali, L. Shulenburger, J. Kim, Embracing a new era of highly efficient and productive quantum Monte Carlo simulations, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, NY, USA, ISBN: 978-1-4503-5114-0, 2017, pp. 38:1–38:12, http://dx.doi.org/10.1145/3126908.3126952.
[32] H. Lu, S. Seo, P. Balaji, MPI+ULT: overlapping communication and computation with user-level threads, in: 2015 IEEE 17th Int. Conf. on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th Int. Conf. on Embedded Software and Systems, 2015, pp. 444–454, http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.82.
[33] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, RAJA: portable performance for large-scale scientific applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC, P3HPC, IEEE, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019, pp. 71–81.
[34] CHAI, CHAI, 2020, https://github.com/LLNL/CHAI.
[35] Umpire, Umpire, 2020, https://github.com/LLNL/Umpire.
[36] OpenMP, OpenMP website, 2020, https://www.openmp.org/.