2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC) | 978-1-6654-6021-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/P3HPC56579.2022.00009

Portable and Efficient Dense Linear Algebra in the Beginning of the Exascale Era

Mark Gates*, Asim YarKhan*, Dalal Sukkari*, Kadir Akbudak*, Sebastien Cayrols*, Daniel Bielich*,
Ahmad Abdelfattah*, Mohammed Al Farhan†, Jack Dongarra*
*Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
{mgates3, yarkhan, sukkari, kadir, sebastien.cayrols, drbieli, aahmad2, dongarra}@icl.utk.edu
†KAUST, Thuwal, Saudi Arabia
mohammed.farhan@kaust.edu.sa

***Abstract*—The SLATE project is implementing a distributed dense linear algebra library for highly-scalable distributed-memory accelerator-based computer systems. The goal is to provide a library that can be easily ported to different hardware (CPUs, GPUs, accelerators) and will provide high performance for machines into the future. Current ports include CPUs, CUDA, ROCm, and oneAPI. We achieve both performance and portability by leveraging several layers and abstractions, including OpenMP tasks to track data dependencies, MPI for distributed communication, and the BLAS++ and LAPACK++ libraries developed as a portable layer across vendor-optimized CPU and GPU BLAS and LAPACK functionality. We rely on the C++ standard library and templating to reduce code duplication for better maintainability. The few kernels not present in BLAS are implemented in CUDA, HIP, and OpenMP target offload, and are easily ported to new platforms.**

***Index Terms*—numerical linear algebra, distributed computing, GPU computing**

I. Introduction

The SLATE project [1] is implementing a distributed dense linear algebra library for highly-scalable distributed-memory accelerator-based computer systems. We seek to provide a replacement for ScaLAPACK, with similar functionality including parallel BLAS, norms, solving linear systems, least squares, eigenvalue problems, and the singular value decomposition (SVD), as well as expanding coverage to new algorithms. The goal is to provide a library that can be easily ported to different hardware (CPUs, GPUs, accelerators) and will provide high performance for machines into the future. We achieve both portability and performance by leveraging several layers and abstractions, illustrated in Figure 1.

To gain more parallelism and overlap communication, recent linear algebra libraries have leveraged runtime systems to schedule tasks. SLATE uses the OpenMP runtime [2] to track dependencies between tasks and schedule task execution on each node. To keep the size of the task graph manageable, we aggregate operations on individual tiles of the matrix into larger super-tasks, and track dependencies between super-tasks, as discussed in Section III-A. For execution on the GPU, many of these super-tasks can be mapped to batched BLAS operations, which are a key for SLATE to achieve portable performance.

LAPACK and ScaLAPACK have been the de facto standard linear algebra libraries for decades, and this success can largely

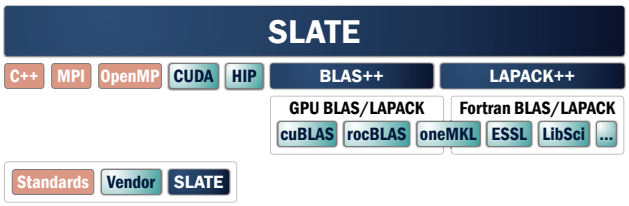

Fig. 1. SLATE software stack.

be attributed to ways that they layered the software using vendor-optimized BLAS libraries to achieve good performance in a portable fashion. Inspired by this, we developed the BLAS++ and LAPACK++ libraries to serve both as a C++ interface to BLAS and LAPACK, and as a portability layer across different vendor-provided GPU linear algebra libraries, including the batched BLAS, as described in detail in Sections III-B to III-D. Current ports are to the NVIDIA CUDA, AMD ROCm, and Intel oneAPI platforms.

A significant risk in writing a portable GPU accelerated library is having GPU kernels written in proprietary GPU computing languages. While several languages and libraries exist for writing portable code, at the moment there is not one dominant, portable technique in the industry. SLATE maintains relatively few of its own GPU kernels, primarily relying on the vendors' BLAS. These few, simple kernels are easily ported between platforms, and we are keeping options open to a variety of implementations for specific platforms. Our kernels were initially implemented in CUDA, which are then translated to HIP. For the Intel platform, our strategy is to implement kernels using OpenMP target offload, as discussed in Section IV-D, which may also provide portability to a variety of architectures.

In a break from the traditional Fortran and more recent C based linear algebra libraries, SLATE uses standard C++, currently relying on C++17 features including templates and the C++ standard library. Using templates and object-oriented abstractions reduces the amount of code for better maintainability and productivity. We template for both data types and implementation targets on the CPU or GPU, as described in Section III-E. The standard library provides us with portable

2831-3909/22/$31.00 ©2022 IEEE
DOI 10.1109/P3HPC56579.2022.00009

and efficient implementations of many data structures, algorithms, and threading utilities. Using these C++ features increases developer productivity by reducing the code that needs to written and maintained.

II. Related Work

A. ScaLAPACK

To draw inspiration during the design of SLATE, we looked at the ScaLAPACK library [3] and examined what led to its long-term viability. ScaLAPACK has been the de facto standard distributed dense linear algebra library for over two decades, being first released in 1995. It has remained portable and highly productive over multiple generations of computer architectures and network designs. This degree of longevity is due to a dependence on a set of core features and underlying libraries:

a) BLAS: The *Basic Linear Algebra Subprograms* specify operations on vectors and matrices to provide a foundation for building higher level mathematical routines. There are 3 levels of BLAS routines: Level 1 BLAS work on vectors only [4], Level 2 BLAS contain matrix-vector operations [5], and Level 3 BLAS contain matrix-matrix operations [6]. The BLAS routines can be very highly tuned on the underlying single-node architectures, whether they are CPU, multicore, GPU, or other hardware devices. Particularly, the Level 3 BLAS routines, of which general matrix-matrix multiply (`gemm`) is the canonical example, have a high computational intensity with $O(n^2)$ memory accesses for $O(n^3)$ flops, yielding a surface-to-volume effect, and can be optimized to approach the theoretical peak on many devices.

b) LAPACK: The *Linear Algebra Package* [7] builds on top of the BLAS to provide routines for factoring matrices (e.g., LU, QR, Cholesky), solving systems of linear equations and linear least squares, eigenvalue problems and the SVD. Earlier linear algebra libraries such as EISPACK and LINPACK used Level 1 and 2 BLAS, which have $O(1)$ memory accesses per operation, resulting in memory-bound algorithms. In comparison to these, LAPACK reformulated algorithms by blocking in order to extensively use Level 3 BLAS. LAPACK routines are defined in shared-memory nodes and depend on the underlying BLAS implementation for high performance.

c) PBLAS: The *Parallel Basic Linear Algebra Subprograms* [8] are an implementation of the BLAS for distributed memory architectures. These routines were intended to support a distributed equivalent of LAPACK, so that the LAPACK algorithms could be translated to a distributed-memory infrastructure with high productivity.

d) BLACS: The *Basic Linear Algebra Communication Subprograms* are a linear algebra-oriented communication interface designed to be implemented on multiple distributed memory machines in a portable and efficient manner. BLACS can be built on top of the PVM (Parallel Virtual Machine) [9] or MPI (Message Passing Interface) [10] communication libraries. Today, MPI dominates HPC communication interfaces, although alternatives do exist.

ScaLAPACK built on these underlying components that provide abstract matrix operations. Implementations of high performance BLAS are provided by vendors or open source projects as new hardware became available. Support for newer network infrastructures was left to other projects such as the MPI implementations. The PBLAS routines made the transition from shared memory LAPACK algorithms to distributed memory ScaLAPACK implementations a relatively direct translation operation. The longevity of ScaLAPACK is in a great part due to the use of an appropriate set of lower level building blocks, which were then implemented on newer architectures by vendors.

B. Task-based libraries

While (Sca)LAPACK[1] blocks the matrix to achieve high performance, it uses a "fork-and-join" model, where the main thread issues a large BLAS operation to be done in parallel ("fork"), then the threads or MPI ranks synchronize ("join"). This creates unnecessary artificial synchronization points, reducing the achievable performance.

The PLASMA [11] and FLAME [12] libraries popularized linear algebra algorithms where the data is represented as collections of contiguous tiles, using a runtime scheduler to track data-tile dependencies between tasks and schedule tasks when their data-tiles are ready. This eliminates artificial synchronization points inherent in (Sca)LAPACK's fork-and-join model. FLAME uses its own SuperMatrix runtime. Originally PLASMA used the home-grown QUARK runtime scheduler, which was later replaced by OpenMP tasks when OpenMP 4.5 added data dependencies [13]. This works well on a variety of systems, but the large task graph was found to overwhelm OpenMP schedulers on heavily accelerated systems using Intel Xeon Phi, which can only be exacerbated in distributed environments with multiple accelerators per node.

DPLASMA [14] extended the PLASMA algorithms to distributed environments via the PaRSEC runtime system [15], with GPU accelerator support using CUDA. In DPLASMA, a number of algorithms use a Parameterized Task Graph (PTG) to describe the tasks and dependencies [16], where each task internally knows all its successor tasks via a formal state-space description. In this representation, the task dependencies do not need to be inferred at runtime and a PTG can be efficiently executed. However, algorithms that make runtime decisions based on data, such as LU with partial pivoting, are easier to express using the less efficient but more intuitive Insert Task interface, which is similar to OpenMP tasks and has similar scalability concerns.

Chameleon [17] extended PLASMA algorithms differently to support various runtime systems: QUARK, OpenMP, PaRSEC, or StarPU, with PaRSEC and StarPU supporting distributed environments and GPU accelerators using CUDA. Use of advanced runtimes in DPLASMA and Chameleon has been a successful strategy. The main difficulty is that it is tied to

[1]We use "(Sca)LAPACK" for discussion that applies to both LAPACK and ScaLAPACK, and "(D)PLASMA" for both PLASMA and DPLASMA.

the runtime, which often requires close collaboration with the runtime developers. For instance, adding ROCm or oneAPI support likely requires first adding this support to the runtime.

MAGMA [18] is a library for single-node GPU-accelerated linear algebra, with current ports to CUDA and ROCm. While it does not use a runtime scheduler, it uses similar techniques such as a lookahead update to overlap tasks on the CPU with tasks on the GPU and remove artificial synchronizations.

C. GPU platforms

NVIDIA was the first vendor to offer general-purpose GPU programming via their CUDA language [19], as well as BLAS and LAPACK-style linear algebra routines in their cuBLAS and cuSOLVER libraries. They were also among the first to offer batched BLAS routines.

As a counterpart to NVIDIA's CUDA platform, AMD developed the Radeon Open Compute (ROCm) GPU ecosystem [20]. ROCm is open-source and has several math libraries including rocBLAS, rocSOLVER, and rocSPARSE. AMD also developed the HIP (Heterogeneous Interface for Portability) layer on top of the ROCm and CUDA ecosystems. For instance, the hipBLAS library wraps either cuBLAS or rocBLAS. The HIP programming language closely follows CUDA, so that it can be compiled for either platform, allowing developers to write portable code to run on AMD and NVIDIA GPUs.

A third platform is provided by SYCL, which is an open standard for GPU computing in C++ [21] managed by the Khronos Group and originally targeting OpenCL, but now supporting multiple backends. Intel adopted SYCL for their GPUs and extended it in their DPC++ (Data Parallel C++) compiler. Building on DPC++, Intel developed the oneAPI libraries [22], including the oneMKL (Math Kernel Library).

These three ecosystems comprise the U.S. Department of Energy's current and upcoming top systems: Summit and Perlmutter using NVIDIA GPUs, Frontier using AMD GPUs, and Aurora using Intel GPUs. Therefore, these platforms have been the current focus of SLATE's porting efforts.

III. Designing for the Future

The SLATE project has the goal of creating a high performance library that is portable to new hardware platforms. To achieve this, SLATE builds on a number of layers and abstractions, as shown in Figure 1, much as ScaLAPACK did.

A. OpenMP tasking and MPI communication

SLATE uses OpenMP as its runtime scheduler, as PLASMA does, which provides a portable standard to rely on. In the PLASMA algorithms, tasks operate on a tile-by-tile basis, each task updating usually one or two tiles. This results in an $O((n/n_b)^3)$ task dependency graph, for a matrix of dimension n with block size n_b, which can be daunting for large matrices in a distributed environment. For instance, in the Cholesky factorization diagrammed in Figure 2, PLASMA would have a task for each colored tile. In contrast, in SLATE these individual tile operations are aggregated into super-tasks, yielding

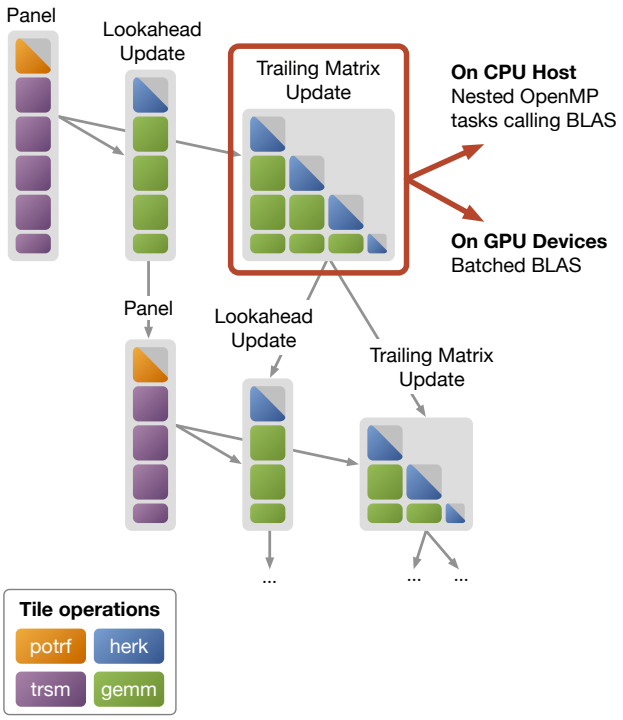

Fig. 2. Super-tasks with dependencies in SLATE, for the Cholesky factorization. Super-tasks are dispatched to either the CPU or GPU.

a much coarser granularity that tracks dependencies between block columns and block updates, resulting in an $O(n/n_b)$ task dependency graph as illustrated by the large gray blocks in Figure 2. These super-tasks can then schedule fine-grained tasks on tiles in a variety of ways, including OpenMP tasks *without* dependencies, OpenMP nested parallelism, or batched BLAS on the CPU or the GPU. Scheduling super-tasks this way yields many of the advantages of finer grained tasks – reducing artificial synchronizations, overlapping tasks, and overlapping communication with computation – in an easy-to-program manner with the convenience of OpenMP compiler pragmas.

Using a node-level runtime instead of a distributed runtime means that SLATE must manage its own communication. Due to its ubiquity and familiarity to developers, SLATE directly uses the MPI standard for communication, unlike ScaLAPACK which had the BLACS layer.

B. BLAS++ and LAPACK++ CPU portability layer

A key layer for portability in SLATE are the new BLAS++ and LAPACK++ libraries developed to wrap around the vendor optimized BLAS and LAPACK libraries. These are distributed as independent C++ libraries that can be used by applications other than SLATE. Major goals include making code more portable by hiding differences between Fortran compilers, enabling template code by smoothing differences between real-valued and complex-valued routines, and simplifying the use of BLAS and LAPACK routines to improve productivity.

The Fortran BLAS standard continues to provide portable performance across different CPUs, with both sequential and multi-threaded versions available. With some caveats, it is portable across different CPU platforms. The caveats are mostly related to things unspecified in the Fortran standard regarding the application binary interface (ABI). Fortran 2003 can be used to export a C-compliant interface, resolving these issues, with a C++ interface layered upon that. We chose to handle all the issues in C++ to avoid depending on the Fortran compiler used for the BLAS library.

A primary role of BLAS++ is to provide C++ calling conventions, instead of the C interface to BLAS provided by CBLAS, or the Fortran conventions in directly calling BLAS. Minimally, this means scalars are passed by value instead of by pointer, and the spelling of routines in the Fortran ABI is hidden, whether the convention is `dgemm_`, `DGEMM`, or `dgemm`. BLAS++ deals with any peculiarities of the Fortran BLAS implementation, such as whether single-precision routines like `sdot` return a single-precision or double-precision result (an oddity in f2c, CLAPACK[2], and macOS Accelerate), or how string values are passed to Fortran, a long unappreciated issue in CBLAS and LAPACKE [23].

BLAS++ goes further, removing the initial letter in BLAS routines signifying the data type ("d" in `dgemm`), and overloading `blas::gemm` for different data types that dispatch to `sgemm`, `dgemm`, `cgemm`, or `zgemm` for single (float), double, complex-single, or complex-double, respectively. It can potentially support other precisions such as half and quad. This overloading is critical to enable templating in SLATE, by making SLATE's code identical for all data types.

A second goal is to handle real and complex values in an identical fashion. For instance, in the traditional BLAS, there are `[cz]herk` routines[3] for a complex-Hermitian rank-k update, and `[cz]syrk` routines for a complex-symmetric rank-k update, but only `[sd]syrk` routines for real valued matrices, since a real-Hermitian matrix is typically just called symmetric. Instead, BLAS++ provides both `blas::herk` and `blas::syrk` for all data types; in the real case both `blas::herk` and `blas::syrk` map to `[sd]syrk`. Again, this is critical for templating in SLATE. For instance, Cholesky can call `blas::herk` for all data types, whereas in non-templated libraries like LAPACK and (D)PLASMA, routine names need to be translated from `[cz]herk` to `[sd]syrk` when translating code from complex to real-valued.

BLAS++ rectifies other examples where BLAS or CBLAS treats real and complex values differently. CBLAS passes real scalars by value, but complex scalars by pointer; BLAS++ always passes scalars by value:

```
// CBLAS: real alpha is passed by value.
cblas_sscal( n,  alpha, x, incx );

// CBLAS: complex alpha is passed as pointer.
cblas_cscal( n, &alpha, x, incx );

// BLAS++: alpha is always passed by value.
blas::scal( n, alpha, x, incx );
```

CBLAS returns real values, but passes complex values as output arguments. In Fortran BLAS, whether complex values are returned or passed as a hidden output argument depends on the Fortran compiler. BLAS++ always returns complex values the same as real values, which primarily affects the dot product:

```
// CBLAS: real result is returned from dot.
result = cblas_sdot( n, x, incx, y, incy );

// CBLAS: complex result is output argument.
cblas_cdotc_sub( n, x, incx, y, incy, &result );

// BLAS++: result is always returned from dot.
// Also, blas::dot means conjugated, x^H y;
// use blas::dotu for unconjugated, x^T y.
result = blas::dot( n, x, incx, y, incy );
```

For asynchronous GPU routines, it is advantageous for the result to be an output argument, but again real and complex are treated the same:

```
// BLAS++ GPU routine: result is output argument.
blas::dot( n, x, incx, y, incy, &result, queue );
```

BLAS++ uses enum values for options such as uplo, side, and transpose, similar to enums in CBLAS, which in traditional BLAS are characters. This provides a measure of type safety, consistency, and clarity to the code. With a call like:

```
dtrsm( "L", "L", "N", "U",
       m, n, alpha, A, lda, B, ldb );
```

it is not immediately obvious what the options mean without remembering the exact order of arguments, while with:

```
blas::trsm( Side::Left, Uplo::Lower,
            Op::NoTrans, Diag::Unit,
            m, n, alpha, A, lda, B, ldb );
```

the arguments are obvious. Internally these enums are mapped to characters for calling Fortran BLAS, and to the appropriate enums for calling cuBLAS, rocBLAS, and oneMKL.

BLAS++ uses the standard C++ `std::complex` data type, rather than a library-specific data type such as `lapack_complex_double`, or type stripping using `void*` as CBLAS does. This provides type safety and avoids the need for casting data types. One oddity in the C++ standard library is that `std::conj` applied to a real value returns a complex value, even though the imaginary part will be zero. BLAS++ provides `blas::conj` that returns a real value in this case, making it a no-op, again to simplify templating.

All of this applies equally to the LAPACK++ library. Additionally, LAPACK routines often requires workspaces, with a query for the optimal workspace size. LAPACK++ hides the workspaces, doing the query and allocating memory internally as needed, simplifying application code. If needed for performance reasons, it would be easy to add versions that take workspaces using C++ overloading.

[2]CLAPACK is a translation of LAPACK into C using the Fortran to C translator f2c, but keeping Fortran calling conventions. LAPACKE is a library of C wrappers around LAPACK, similar to CBLAS.

[3]Henceforth, we will use `[..]` regular expression syntax as shorthand for multiple names, in this case, `cherk` and `zherk`.

Thus, the BLAS++ and LAPACK++ libraries greatly simplify development in SLATE and other applications. They make code both more portable and enable maintaining a single templated version rather than separate versions for each data type. Our strategy in SLATE is to code for the complex-valued version, and the real-valued template instantiations generally work automatically.

C. BLAS++ and LAPACK++ GPU portability layer

In addition to wrapping the Fortran BLAS and LAPACK API, the BLAS++ and LAPACK++ libraries also provide wrappers around the various GPU BLAS and LAPACK APIs available in the CUDA cuBLAS/cuSOLVER, ROCm rocBLAS/rocSOLVER, and oneAPI oneMKL libraries. This is a more difficult porting challenge than the CPU BLAS, which already follow the Fortran BLAS standard. While the various GPU BLAS libraries loosely follow the BLAS standard, they all have different routine names, different data types, different enum constants, and take different arguments for streams or queues. Again, BLAS++ aims to smooth over these differences, providing a single interface for all the various platforms. This was critical to enable SLATE to be easily ported to different platforms.

Other libraries also aim to be portability layers for math libraries. MAGMA provides a portability layer across cuBLAS and rocBLAS, which served as a model for BLAS++ and LAPACK++. However, MAGMA lacks C++ features such as overloading for different precisions. AMD's HIP (Heterogeneous Interface for Portability) libraries form a layer across CUDA and ROCm libraries, including the HIP language for writing kernels and the hipBLAS, hipSOLVER, etc. libraries that sit on top of cuBLAS, cuSOLVER, etc. for CUDA GPUs, and on top of rocBLAS, rocSOLVER, etc. for ROCm GPUs. However, HIP lacks support for Intel oneAPI. Intel's oneAPI libraries also have a goal of being a portable library across various platforms, with ROCm and CUDA builds available, but currently have a dependency on Intel's DPC++ compiler and library [24]. Thus it is unfortunately still unclear if true portability across all three major GPU hardware architectures (NVIDIA, AMD, Intel), as well as future platforms, will be achieved by either of these projects.

In contrast, BLAS++ and LAPACK++ are designed to have minimal dependencies and work with any recent C++ compiler that the underlying vendor libraries supports. They are also easily ported to new GPU BLAS libraries.

To support GPUs, BLAS++ introduces a Queue class that wraps around either a CUDA stream and cuBLAS handle for CUDA, HIP stream and hipBLAS handle for ROCm, or SYCL queue for oneAPI. When calling GPU routines in BLAS++, the main difference from CPU routines is the addition of this queue argument at the end, and that pointers point to GPU device memory:

```
// BLAS++ call on CPU. x is in CPU memory.
blas::scal( n, alpha, x, incx );

// BLAS++ call on GPU. dx is in GPU device memory.
blas::scal( n, alpha, dx, incx, queue );
```

Having one queue object, instead of a separate BLAS handle and associated stream, simplifies the use compared to cuBLAS and rocBLAS. Unlike cuBLAS and rocBLAS, a BLAS++ queue knows what device it is for, so there is no need to call `cudaSetDevice` or `hipSetDevice` before calling a device routine. This matches SYCL and oneAPI, which do not have the concept of a "current device". It also makes programming simpler and safer, as calls always correctly execute on the device associated with the queue. Initially, BLAS++ also provided a `set_device` function, which affected routines that previously did not take a queue such as `device_malloc`; however this use is being phased out.

Due to differences in the vendor libraries, several design choices need to be made. cuBLAS and rocBLAS take scalars by reference as pointers, to provide an extra level of efficiency when the scalars reside in GPU memory, while oneMKL takes scalars by value. This unfortunately forces a portable interface to use a mode that all three platforms support, namely BLAS++ takes scalars by value on the CPU. It then passes these scalars to cuBLAS or rocBLAS via CPU pointers, or passes them by value to oneMKL. This can reduce performance, for instance in QR factorization, passing by value forced us to make an extra CPU copy of the tau vector, which could otherwise be accessed directly on the GPU using pointers. We can add overloaded versions of routines that takes scalars as GPU pointers, but it does not appear to be as portable and might introduce the need for `#ifdef` or `if constexpr` conditions into SLATE's code, something we try to minimize.

CUDA and ROCm both index GPU devices as non-negative integers. SYCL does not index GPU devices; the user must query for a list of devices, which returns objects representing each device. For multi-GPU codes, this makes iterating over GPU devices cumbersome, and porting across platforms more difficult. Therefore, BLAS++ enumerates SYCL devices internally, and presents them as indexed by a non-negative integer the same as in CUDA and ROCm.

Recently we expanded LAPACK++ to cover some LAPACK-style routines on the GPU, namely Cholesky, LU, and QR factorizations (`potrf`, `getrf`, and `geqrf`, respectively). These are provided in cuSOLVER, rocSOLVER, and oneMKL. In cuSOLVER, these take a workspace. GPU memory allocation is relatively expensive, often imposing extra synchronization. Therefore, unlike in the LAPACK++ CPU implementation that hides workspaces, for GPU routines the workspace and workspace query is exposed in the LAPACK++ interface. If workspaces are not needed for an implementation, the query simply returns zero. cuSOLVER also requires a cuSOLVER handle, separate from the cuBLAS handle, whereas rocSOLVER uses the rocBLAS handle, and oneMKL uses just the SYCL queue. To support cuSOLVER, LAPACK++ subclasses the BLAS++ Queue to add the cuSOLVER handle. On ROCm and SYCL platforms, the LAPACK++ Queue adds nothing to the BLAS++ Queue. This solver handle is allocated on its first use, so it adds no overhead if cuSOLVER routines are never called.

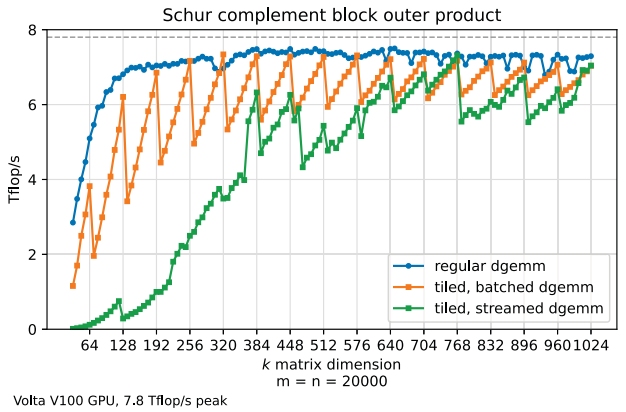

Fig. 3. Block outer-product matrix multiply, implemented as single gemm, batched gemm on tiles, or individual tile gemms in separate streams, on NVIDIA V100 GPU.

D. Batched BLAS

In recent years batched routines [25], which take a set of matrices and perform the same operations on all of them in parallel, have become popular. The cuBLAS, rocBLAS, and oneMKL vendor libraries supply some batched BLAS routines, but not necessarily all the Level 3 BLAS routines (i.e., batched interfaces to `gemm`, `hemm/symm`, `herk/syrk`, `her2k/syr2k`, `trsm`, and `trmm`). For the routines where the vendor has not provided a batched interface, BLAS++ creates batched implementations that call the vendor's non-batched routines in parallel using multiple streams or queues. A BLAS++ Queue can actually have multiple underlying streams or queues that it uses internally; externally, it operates as an in-order queue.

The motivation for using batched BLAS in SLATE comes from updates in typical matrix factorizations. In LU factorization, the trailing matrix update is a matrix-multiply of the panel block-column and a block-row to update the trailing matrix (similar to Figure 2). In ScaLAPACK, this is accomplished with a call to `pdgemm`, which in turn calls `dgemm`. In SLATE, we divide the matrix into tiles and perform a `gemm` for each output tile. Since all output tiles are independent here, we can batch the `gemm` operations together for increased performance. Figure 3 compares the performance of doing this update as a single gemm (blue line), as a batched gemm on tiles (orange line), or as individual tile gemms in multiple streams (green line). We see that at well chosen block sizes, which for cuBLAS are multiples of 64 starting at 192, the batched gemm matches the performance of the single gemm. Thus we gain the advantages of storing the matrix by tiles [1] that can be individually allocated, copied, and operated on, without sacrificing performance. For tile-based frameworks, multi-streaming individual gemms works well for large block sizes but has significantly lower performance for small block sizes.

E. C++ Templates and Abstractions

SLATE uses C++ templates extensively in its algorithms and code, in support of productivity and portability. Using templates improves productivity by reducing the amount of code that needs to be written and maintained. And templates improve portability by enabling high level algorithms to be targeted to various implementations under the covers.

The obvious use of templates is to support the four classic floating point data types – float, double, complex-float, and complex-double – with a single templated code. Today, this can be extended to other precisions such as 16-bit half precision (IEEE or bfloat "brain float" [26]), double-double, and quad precision. SLATE algorithms are written assuming complex values, with operations like conj automatically simplifying when the code is instantiated for real values.

SLATE also templates its algorithms on the target implementation, whether that is on the CPU host using OpenMP tasks, on GPU devices using batched BLAS, or another implementation. This makes the top-level algorithm largely independent of the lower-level implementation, and allows adding new implementations without changing application code that uses SLATE.

A user can further select among different algorithms by passing an option to SLATE's routines, in contrast to other libraries where each new variant adds a new routine to the public API. For instance, LAPACK has four least squares routines, depending on the implementation:

- QR (`dgels`),
- SVD using divide-and-conquer (`dgelsd`),
- SVD using QR iteration (`dgelss`),
- complete orthogonal factorization (`dgelsy`).

SLATE has only one top-level least squares routine (`gels`), which can call various implementations (currently Householder QR or Cholesky QR) based on an option passed to the routine. MAGMA has different routines depending on whether the matrix originates in CPU memory (e.g., `magma_zpotrf`) or GPU memory (`magma_zpotrf_gpu`), and for multi-GPU variants (`magma_zpotrf_m`, `magma_zpotrf_mgpu`). In SLATE, the matrix object tracks where tiles are distributed across multiple nodes in CPU host memory or GPU device memory, and all algorithms are built to be multi-GPU, so there is only one version to maintain. This also makes SLATE much easier for an application to use.

In addition to templates, SLATE uses C++ object-oriented abstractions, such as Matrix and Tile classes, that also greatly reduce the amount of code and simplifies programming. Instead of the 19 parameters of the PBLAS `pdgemm` routine, SLATE's `gemm` routine takes just 5 parameters: scalars alpha and beta, and matrices A, B, and C. The transposition operations, dimensions, offsets, data pointers, and context are all encapsulated in the Matrix objects. The top-level code can even ignore the transposition operation, which is handled at a low level when calling BLAS++.

IV. Portability

A. OpenMP Issues

Following the OpenMP standard gives us a portable runtime to work with. We did find that different compilers or compiler versions seemed to make different implicit choices about whether variables are private or shared. For instance, in:

```
for (int i = 0; i < n; ++i) {
    #pragma omp task
    my_function( i );
}
```

whether `i` is considered private (desired) or shared (causes errors) seems to vary, which could be an OpenMP compliance issue. The recommendation is to add `default( none )` and always declare whether variables are private or shared:

```
#pragma omp task default( none ) \
        firstprivate( i ) shared( A )
```

However, on some platforms adding `default( none )` causes compile errors. Using an `assert` inside an OpenMP task required declaring the C++ constant `__func__` for g++ versions $\leq$ 9. Referencing an `MPI_Datatype` required declaring Open MPI internal globals such as `ompi_mpi_double`. Our solution is to define a macro, `slate_default_none`, that we can set to `default( none )` for test builds to check that variables have explicit `firstprivate` and `shared` clauses, but the macro is empty for general builds:

```
#pragma omp task slate_default_none \
        firstprivate( i ) shared( A )
```

B. SLATE Port to AMD Platform

SLATE was initially written to support NVIDIA's CUDA and cuBLAS for GPU acceleration. As the ROCm software stack will be used on upcoming exascale systems, such as Frontier, we expanded SLATE's GPU support to include AMD GPUs.

The strategy we employed in SLATE is to leverage the BLAS++ library as a portability layer to run on AMD and NVIDIA GPUs. We replaced all the CUDA and cuBLAS function calls by portable abstractions in the BLAS++ library. For example, `cudaStream_t stream` and `cublasHandle_t cublas_handle` are replaced by the BLAS++ wrapper `blas::Queue queue`, which provides CUDA and ROCm backends. BLAS++ also handles setting the device as needed based on the queue, eliminating uses of `cudaSetDevice`. Because of the close similarity between CUDA and ROCm semantics, few issues arose in porting to ROCm.

C. SLATE Port to Intel Platform

SLATE has an experimental port to Intel accelerators, which is well underway and expected to be released when Intel's HPC GPUs become available. Intel provides the oneAPI set of tools and libraries as a unified, multi-architecture programming model path for developing high-performance, data-centric applications across diverse architectures. For the purposes of SLATE, we are using the SYCL interfaces to manage GPU access and memory allocation and using the oneMKL math library for optimized BLAS and LAPACK routines.

There are some differences between SYCL and the earlier approaches that needed to be addressed. For SYCL, all calls need to be provided with a SYCL queue, which implies the device to execute on, whereas in the earlier CUDA and ROCm implementation, SLATE used a `set_device` call to set the current device, notably before GPU memory allocations. To accommodate SYCL, a queue argument was added to the BLAS++ memory allocation interface, and the `set_device` call moved into BLAS++. The BLAS++ `set_device` function is deprecated.

For SYCL, the sequence of calls submitted to a SYCL queue will (by default) execute asynchronously and out-of-order. Each oneAPI call generates a SYCL event, which may be used for synchronizing enqueued tasks. However, the SLATE programming style expects the device calls to execute in the order that they are submitted. In order to match that expected behavior, the SYCL queues are initialized using the `in_order` property to provide in-order execution.

Beyond these changes, which are contained mostly in the BLAS++ layer, the SLATE algorithms did not require any major changes to adapt to the Intel oneAPI platform. This demonstrates the careful design in SLATE that allows the platform porting changes to be very localized to specific layers.

D. Portability of Device Kernels in SLATE

SLATE has made a substantial effort to minimize the GPU accelerator specific code, obtaining a majority of accelerator functionality from the BLAS++ and LAPACK++ libraries. However, a small set of useful routines remain that are not contained in BLAS++ and LAPACK++ and need to run on the accelerators. This includes routines for setting initial matrix values, adding, copying, scaling, and transposing matrices, and computing various matrix norms. These are all memory-bound kernels that simply loop once over the matrix entries, and so are relatively easily ported. In SLATE, all these routines were initially implemented in CUDA. ROCm/HIP support for AMD GPUs was added to SLATE by using the `hipify` tool to automatically translate the CUDA code. This is integrated into SLATE's Makefile and CMake build systems, so any updates to the CUDA code are automatically translated to HIP, significantly reducing the effort to maintain two versions.

Only minor differences between CUDA and HIP have arisen. One is the device implementation of complex `abs`, which in ROCm was not numerically robust for numbers that could have intermediate overflow. We substituted an implementation based on LAPACK, which is the only case where our kernel code has an `#ifdef __NVCC__` to use different code for CUDA and HIP. Another is that the `x *= y` operator generated a compiler error for complex numbers in ROCm; simply substituting `x = x * y` solved the issue.

To provide oneAPI support, SLATE has implemented OpenMP device-offload kernels for these routines. Currently,

these OpenMP device-offload kernels are functionally equivalent to the other implementations, though some do not yet achieve the same performance levels. Much of the performance difference is because the CUDA and ROCm kernels use high-speed GPU local shared-memory regions to hold buffers for accumulating partial results, which in CUDA equate to a user-managed L1 cache. These memory regions are difficult to specify and access under OpenMP 4.5. However, the OpenMP 5.0 specification includes predefined memory allocators, so that different kinds of memories can be supported. As full OpenMP 5.0 compiler implementations become more common, SLATE should be able to use these new memory allocators (e.g., `omp_pteam_mem_alloc`) to improve the performance. These OpenMP device-offload kernels may eventually reach sufficient performance to allow SLATE to remove the remaining CUDA and ROCm vendor-specific code, making SLATE more device independent and easily portable to new platforms.

V. Performance Results

A. Experimental Setup

To validate the portability of SLATE, we performed experiments on two systems: the NVIDIA-based system Summit [27] and the AMD-based system Crusher [28], both located at the Oak Ridge Leadership Computing Facility (OLCF). Each node of Summit has two 22-core IBM POWER9 CPUs, six NVIDIA V100 GPUs, and 512 GiB of DDR4 memory. A V100 GPU has 16 GiB HBM2 memory and a theoretical peak performance of 7.8 Tflop/s for double-precision arithmetic.

Crusher is an early-access system with the same node hardware as the upcoming Frontier exascale system. Each node of Crusher has one 64-core AMD EPYC 7A53 CPU, four AMD MI250X GPUs, and 512 GiB DDR4 memory. An MI250X GPU has two Graphics Compute Dies (GCDs); each GCD has 64 GiB HBM2e memory and a theoretical peak performance of 26.5 Tflop/s for double-precision arithmetic. An MI250X GPU has a network interface chip (NIC) that connects with Infinity Fabric so that MPI messages can be sent directly between GPUs over the network without passing through the host CPU memory.

One MPI process per GPU for Summit and per GCD for Crusher is used in all experiments. Table I displays the modules that are explicitly loaded and the software environment used on these systems.

TABLE I
Software Environment for Each System.

	Summit	Crusher
Modules (explicitly loaded)	gcc/9.1.0 essl cuda/11.5.2 spectrum-mpi netlib-lapack netlib-scalapack python	rocm/5.1.0 craype-accel-amd-gfx90a
Compiler	GCC v9.1	Cray clang v14
MPI Lib.	IBM Spectrum v10.4	Cray MPICH v8.1
Math Lib. (CPU)	IBM ESSL v6.3	Cray LibSci v21.08

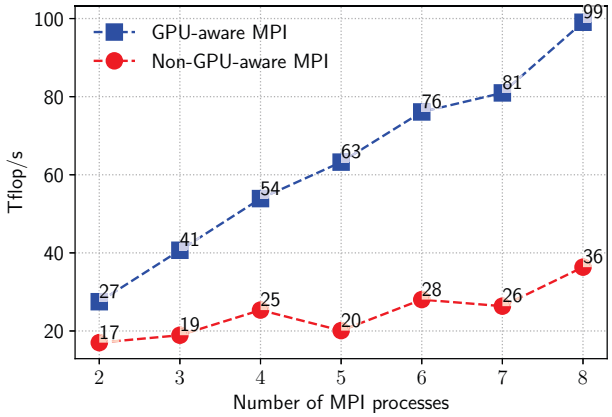

Fig. 4. Performance impact of using GPU-aware MPI on one node of Crusher. `gemm` operation is performed. One MPI process per GCD is used. Using GPU-aware MPI provides significant speedup.

B. GPU-Aware MPI

Due to the integrated NIC on the AMD MI250X GPU, it is essential to store data in the GPU memory and perform communications without including the host CPU. The GPU-aware MPI library on the Crusher system provides such functionality. We investigate the performance impact of exploiting this property in the parallel `gemm` operation. Figure 4 displays the performance effect of using GPU-aware MPI during parallel `gemm` operations. Weak scalability experiments are performed for varying number of MPI processes running on one node of Crusher. The largest square matrices of the same size are used for each number of MPI processes. As seen in the figure, communicating matrices directly from the device memory instead of communicating using host memory results in significant performance improvements. This avoids allocating temporary tiles on the CPU, copying data to the CPU, then after the MPI send/recv, copying data back to the GPU. Using the Matrix and Tile abstractions in SLATE, enabling use of GPU-aware MPI was a simple code change, essentially just specifying the device ID of the tile used for MPI communication, rather than using the default host ID. GPU-aware MPI is used for communication operations in `gemm` and `potrf` experiments on Crusher in the rest of the paper.

However, enabling GPU-aware MPI does raise portability issues. Enabling it may require setting an option in the job scheduler (such as adding `--smpiargs="-gpu"` to jsrun on Summit) or setting an environment variable (such as `MPICH_GPU_SUPPORT_ENABLED=1` on Crusher). There is not currently a standard mechanism in MPI to query whether it is GPU-aware or not, and a non-GPU-aware MPI generally segfaults if given pointers to GPU memory. Hence we will need to add a compile-time or run-time mechanism for the user to instruct SLATE to assume use of GPU-aware MPI.

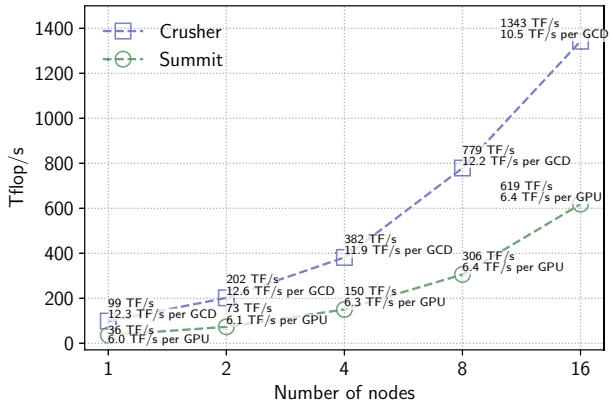

Fig. 5. Weak scaling performance of SLATE's double-precision `gemm` operation on Summit and Crusher.

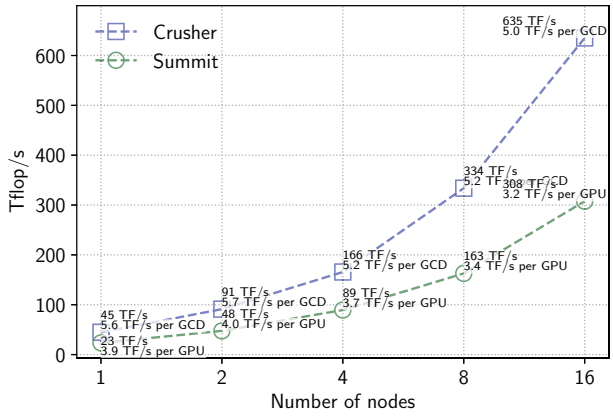

Fig. 6. Weak scaling performance of SLATE's double-precision `potrf` operation on Summit and Crusher.

C. Performance Portability Results

We conduct experiments on Summit and Crusher to empirically show that SLATE obtains decent performance on both systems with completely different architectures and software stacks. Figures 5 and 6 present the performance of the `gemm` and `potrf` operations, respectively, on 1, 2, 4, 8, and 16 nodes of both systems. We report weak scaling results to show the best performance that SLATE can achieve. The largest square matrices of the same size are used for `gemm` and similarly the largest matrix is used for `potrf` for each number of nodes. The commands for running SLATE on one node of the respective system are shown in Figure 7, showing example block sizes n_b and grids. These commands consist of a command for the batch job scheduler software and another command to run the SLATE tester. The best tile size n_b that is found empirically is used for each linear algebra operation and system. Since the mix of communication and tile BLAS operations changes for different matrix operations, the optimal block size can vary, typically in the range $n_b = 256$ to 1024, though it may be larger for some operations such as `gemm`. Compared to a typical ScaLAPACK block size of $n_b = 64$, these larger block sizes provide a higher compute intensity (flops per memory access) required for good GPU performance.

Figure 5 shows the performance of `gemm` on Summit and Crusher. On Summit the single-GPU sustained peak performance of the double-precision `gemm` operation in the CUDA library on one V100 is 7.57 Tflop/s. For different number of nodes, SLATE's multi-node `gemm` performance per V100 varies between 79% and 85% of the sustained single-gpu peak `gemm` performance. Regarding Crusher, the sustained peak performance of the `gemm` operation in the rocBLAS library on one GCD of a MI250X highly depends on the tile size n_b and the transposition of the input matrices. In the case when A is non-transposed and B is transposed, with $n_b = 2048$, the single-GCD sustained peak performance of the double-precision `gemm` operation in the rocBLAS library on one GCD of one MI250X is 24.5 Tflop/s. In the case when both A and B matrices are non-transposed, the peak performance drops to 18 Tflop/s. For different number of nodes, SLATE's multi-node `gemm` performance per GCD varies between 58% and 68% of the single-GCD sustained peak `gemm` performance.

Figure 6 shows the performance of `potrf` on Summit and Crusher. For different number of nodes, SLATE's double-precision `potrf` performance per device varies between 42% and 53% of the sustained peak `gemm` performance on Summit. SLATE's double-precision `potrf` performance per device varies between 28% and 32% of the sustained peak `gemm` performance on Crusher. We expect as the ROCm software stack matures, improving the performance of batched `gemm` and other routines, and as we continue to tune and optimize the SLATE code for this new architecture, that the performance will improve. We have already seen significant improvements since earlier versions of ROCm.

VI. Conclusion

SLATE follows in the footsteps of LAPACK and ScaLAPACK in using blocked algorithms and leveraging vendor-optimized BLAS to achieve high performance in a portable fashion. It differs from these previous efforts in the level of complexity of today's machines: multi-core CPUs, multiple GPUs per node, complex memory hierarchies, machines over-provisioned for floating point computation compared to memory and network bandwidth, and a variety of vendor and community languages and libraries for programming GPUs. We apply the same portability techniques as in the past, updated for today's ecosystems: using industry standards where available in C++, OpenMP, and MPI; minimizing vendor-specific code in CUDA and HIP; and developing a portability layer in the BLAS++ and LAPACK++ libraries around vendor-optimized BLAS and LAPACK, which now includes both CPU and GPU implementations. Early results on forthcoming AMD systems show the promise that we can achieve both portability and good performance. The porting effort to Intel systems shows the robustness of the design, and we look forward to

Commands run on Summit for one node (42 cores, 6 GPU devices, 6 MPI ranks):

```
jsrun --nrs 6 --tasks_per_rs 1 --cpu_per_rs 7 --gpu_per_rs 1 -brs
./tester --dim 16384:524288:16384 --nb 896 --grid 3x2 --target d --repeat 3 gemm
./tester --dim 16384:524288:16384 --nb 896 --grid 3x2 --target d --repeat 3 potrf
```

Commands run on Crusher for one node (64 cores, 8 GPU devices, 8 MPI ranks):

```
srun --nodes 1 --ntasks 8 --cpus-per-task 8 --gpus-per-node=8 --gpu-bind=closest
./tester --dim 16384:524288:16384 --nb 2048 --target d --repeat 3 gemm
./tester --dim 16384:786432:16384 --nb 320 --target d --repeat 3 potrf
```

Fig. 7. Weak scaling performance of gemm on Summit and Crusher.

the availability of Intel's high-performance GPUs. The modern templating and abstractions in C++ and the task scheduling in OpenMP allow us to achieve these goals in a relatively compact code, avoiding significant code complexity and code duplication for ease of maintenance and high developer productivity.

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative. 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.

References

[1] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra, "SLATE: Design of a modern distributed and accelerated linear algebra library," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC19. Denver, CO, USA: ACM New York, NY, USA, Nov 2019, pp. 1–18.

[2] OpenMP Architecture Review Board, *OpenMP API 5.2 Specification*, Nov 2021. [Online]. Available: https://www.openmp.org/specifications/

[3] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers," in *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, 1992, pp. 120–121.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for FORTRAN usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[5] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of basic linear algebra subprograms," *ACM Transactions on Mathematical Software (TOMS)*, vol. 14, no. 1, pp. 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 Transactions on Mathematical Software (TOMS)*, vol. 16, no. 1, pp. 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 *et al.*, *LAPACK users' guide*. SIAM, 1999.

[8] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, "A proposal for a set of parallel basic linear algebra sub-programs," in *International Workshop on Applied Parallel Computing*. Springer, 1995, pp. 107–114.

[9] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "A users' guide to PVM (parallel virtual machine)," Oak Ridge National Lab., TN (United States), Tech. Rep., 1991. [Online]. Available: https://www.osti.gov/biblio/5386544

[10] MPI Forum, *MPI: A message-passing interface standard*, September 2021. [Online]. Available: http://www.mpi-forum.org/

[11] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[12] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 3, pp. 1–26, 2009.

[13] A. YarKhan, J. Kurzak, P. Luszczek, and J. Dongarra, "Porting the PLASMA numerical library to the OpenMP standard," *International Journal of Parallel Programming*, vol. 45, no. 3, pp. 612–633, 2017.

[14] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief *et al.*, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 1432–1441.

[15] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," vol. 38, no. 1, pp. 37–51, 2 2012.

[16] A. Danalis, G. Bosilca, A. Bouteiller, T. Herault, and J. Dongarra, "PTG: An abstraction for unhindered parallelism," 11 2014, pp. 21–30.

[17] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[18] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[19] NVIDIA, *CUDA Toolkit Documentation*, 2022. [Online]. Available: https://docs.nvidia.com/cuda/

[20] AMD, *ROCm v5.x*, 2022. [Online]. Available: https://docs.amd.com/

[21] Khronos Group, *SYCL 2020 Specification*, May 2022. [Online]. Available: https://registry.khronos.org/SYCL/

[22] Intel, *oneAPI Spec 1.1*, Nov 2021. [Online]. Available: https://www.oneapi.io/spec/

[23] *C to Fortran ABI issues in CBLAS and LAPACKE*, 2019. [Online]. Available: https://github.com/Reference-LAPACK/lapack/issues/339

[24] Intel, "Intel oneAPI programming guide: oneMKL usage," May 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top/api-based-programming/intel-oneapi-math-kernel-library-onemkl/onemkl-usage.html

[25] A. Abdelfattah, T. Costa, J. Dongarra, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Kurzak, P. Luszczek, S. Tomov, and M. Zounon, "A set of batched basic linear algebra subprograms and LAPACK routines," *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 3, pp. 1–23, 2021.

[26] Google, "Bfloat16: The secret to high performance on cloud TPUs," Aug 2019. [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[27] Oak Ridge Leadership Computing Facility (OLCF), "Summit user guide," 2022. [Online]. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html

[28] ——, "Crusher quick-start guide," 2022. [Online]. Available: https://docs.olcf.ornl.gov/systems/crusher_quick_start_guide.html

REPRODUCIBILITY APPENDIX

A. Artifact Details

The experiments are performed on Summit and Crusher at OLCF. Each node of Summit has two 22-core IBM POWER9 CPUs and six NVIDIA V100 GPUs. Each node of Crusher has one 64-core AMD EPYC 7A53 CPU and four AMD MI250X GPUs. Each MI250X GPU contains two GCDs, so eight GPU devices are detected by the user code on a node. 1, 2, 4, 8, and 16 nodes of these systems are used in the reported results.

The experiments are performed using the SLATE library. The code is included in the artifact. There are two folders named `slate` in total since the code run on Crusher is slightly modified. This modified version uses device pointers for MPI calls to enable GPU-aware MPI communications.

Two common linear algebra routines implemented in SLATE are run on these two systems: Generalized Matrix-Matrix Multiplication (gemm) and Cholesky factorization (potrf). Double precision arithmetic is used.

B. Artifact

The artifact can be downloaded at this link: https://doi.org/10.5281/zenodo.7003870. The artifact contains a folder for each system as follows:

```
summit/
    install-slate-on-summit.sh
    run-slate-on-summit.sh
    slate/
crusher/
    install-slate-on-crusher.sh
    run-slate-on-crusher.sh
    slate/
```

Each folder has an installation script to install SLATE. The installation scripts can be run as follows:

```
cd summit/
cd slate/
source ../install-slate-on-summit.sh
```

The installation script for Crusher `install-slate-on-crusher.sh` can be run in the same way. Note that these installation scripts must be sourced since they load system modules and change environment variables. These modules and environment variables are also required by the experiments. The following commands are used to reproduce the gemm and potrf results on Summit:

```
cd summit/slate/
bsub ../run-slate-on-summit.sh
```

The following commands are used to reproduce the gemm and potrf results on Crusher:

```
cd crusher/slate/
bash ../run-slate-on-crusher.sh
```

Note that the account numbers in `run-slate-on-....sh` scripts must be set accordingly.

The largest number in the `gflop/s` column in the result output is reported as the performance of the kernel operation on a corresponding number of nodes of the system.