Designing SLATE

Software for Linear Algebra Targeting Exascale

Jakub Kurzak
Panruo Wu
Mark Gates
Ichitaro Yamazaki
Piotr Luszczek
Gerald Ragghianti
Jack Dongarra

Innovative Computing Laboratory, University of Tennessee

October 4, 2017

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

The authors would like to thank Jeff Larkin from the NVIDIA Developer Technology team for sharing insights about the inner workings of NVIDIA hardware and software solutions.

Revision	**Notes**
09-2017	first publication

```
@techreport{kurzak2017designing,
  author={Kurzak, Jakub and Wu, Panruo and Gates, Mark and Yamazaki, Ichitaro
          and Luszczek, Piotr and Ragghianti, Gerald and Dongarra, Jack},
  title={Designing SLATE: Software for Linear Algebra Targeting Exascale},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2017},
  month={September},
  type={SLATE Working Note},
  number={3},
  note={revision 09-2017}
}
```

Contents

List of Figures

CHAPTER 1

Introduction

Figure 1.1 shows the SLATE software stack, designed after a careful consideration of all available implementation technologies [1][1]. The objective of SLATE is to provide dense linear algebra capabilities to the ECP applications, e.g., EXAALT, NWChemEx, QMCPACK, GAMESS, as well as other software libraries and frameworks, e.g., FBSS. In that regard, SLATE is intended as a replacement for ScaLAPACK, with superior performance and scalability in distributed memory environments with multicore processors and hardware accelerators.

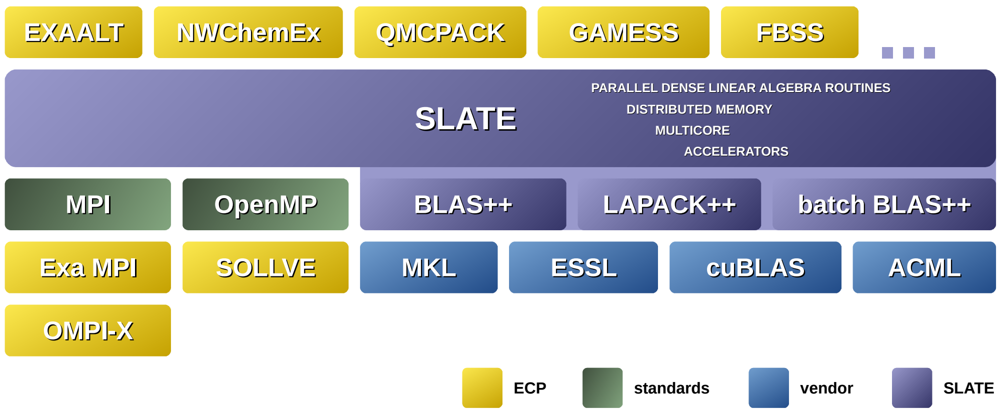

Figure 1.1: Software Stack.

[1]http://www.icl.utk.edu/publications/swan-001

The SLATE project also encompasses the design and implementation of C++ APIs for BLAS and LAPACK [10][2], and for batch BLAS. Underneath these APIs, highly optimized vendor libraries will be called for maximum performance (Intel MKL, IBM ESSL, NVIDIA cuBLAS, AMD ACML, etc.).

To maximize portability, the current design relies on the MPI standard for message passing, and the OpenMP standard for multithreading and offload to hardware accelerators. The collaborations with the ECP Exa MPI and OMPI-X projects are intended to improve message passing capabilities, while the collaboration with the ECP SOLLVE project is intended to improve multithreading capabilities.

There will also be opportunities for replacing the layer of MPI and OpenMP with a specialized runtime system, such as DTE (a.k.a. PaRSEC) [7], contingent on the runtime providing good interoperability with MPI and OpenMP, implementing support for nested parallelism, and demonstrating acceptable scheduling overheads.

Overall, the objective of SLATE is to leverage years of experience maintaining legacy linear algebra software (LAPACK [4], ScaLAPACK [5]), developing new linear algebra software (PLASMA [15], DPLASMA [6], MAGMA [8]), and implementing runtime scheduling systems (QUARK [14], PaRSEC [7], PULSAR [13]), to deliver a software package that:

Targets Modern Hardware such as the upcoming CORAL systems, where the number of nodes is large, and each node contains a heavyweight multicore processor and a number of heavyweight hardware accelerators.

Guarantees Portability by relying on standard computational components (vendor implementations of BLAS and LAPACK), and standard parallel programming technologies (MPI, OpenMP) or portable runtime systems (e.g., PaRSEC).

Provides Scalability by employing proven techniques of dense linear algebra, such as the 2D block cyclic data distribution, as well as modern parallel programming approaches, such as dynamic scheduling and communication overlapping.

Facilitates Productivity by relying on the intuitive *Single Program Multiple Data* (SPMD) programming model and a set of simple abstractions to represent dense matrices and dense matrix operations.

Assures Maintainability by employing useful facilities of the C++ language, such as templates and overloading of functions and operators, and focusing on minimizing code bloat by relying on compact representations.

[2] http://www.icl.utk.edu/publications/swan-002

CHAPTER 2

Design

2.1 Matrix Layout

The new matrix storage introduced in SLATE is probably its most impactful feature. In this respect, SLATE represents a radical departure from the traditional wisdom of dense linear algebra software. Unlike in other packages, including LAPACK, ScaLAPACK, PLASMA, MAGMA, Elemental, where the matrix occupies a contiguous memory region, in SLATE the matrix consists of a collection of individually allocated tiles, with no correlation between their positions in the matrix and their memory locations. The new structure, introduced in SLATE, offers numerous advantages, e.g.:

- The same structure can be used for holding many different matrix types[1], e.g., general, symmetric, triangular, band, symmetric band, etc. (Figure 2.1). No memory is wasted for storing parts of the matrix that hold no useful data, e.g., the upper triangle of a lower triangular matrix. There is no need for using complex matrix layouts, such as the *Recursive Packed Format* (RPF) [2, 3, 11] in order to save space.
- The matrix can be easily converted, in parallel, from one layout to another with $O(P)$ memory overhead, where P is the number of processors (cores/threads) used. Possible conversions include: changing the layout of tiles from column major to row major, "packing" of tiles for efficient execution of the `GEMM` operation[2], low-rank compression of tiles, re-tiling of the matrix (changing the tile size), etc. Notably, transposition of the matrix can be accomplished by transposition of each tile and remapping of the indexes. There is no need for complex in-place layout translation and transposition algorithms [12].

[1] http://www.netlib.org/lapack/lug/node24.html
[2] https://software.intel.com/en-us/articles/introducing-the-new-packed-apis-for-gemm

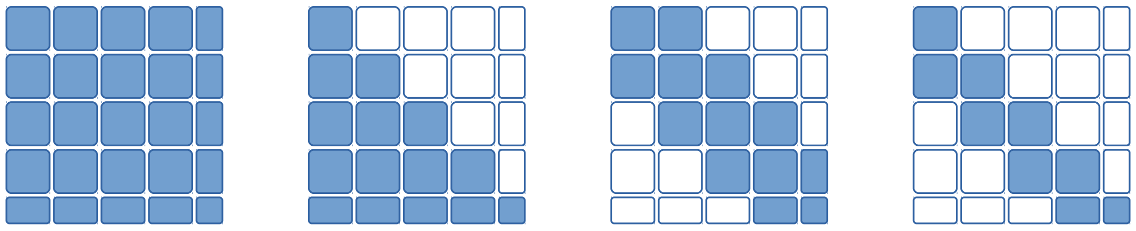

Figure 2.1: Different types of matrices accommodated by the matrix class, from left to right: general, (lower) triangular or symmetric, general band, (lower) triangular or symmetric band. Hollow boxes indicate tiles that are not created and do not consume memory.

- Tiles can easily be moved or copied among different memory spaces. Both inter-node communication and intra-node communication is vastly simplified. Tiles can easily and efficiently be transferred between nodes using MPI. Tiles can be easily moved in and out of faster memory, such as the MCDRAM in the Xeon Phi processors. Tiles can also be copied to one or more device memories in the case of GPU acceleration.

Note: Although the PLASMA library stores the matrix by tiles, it still relies on contiguous matrix allocation, with direct mapping of tiles' coordinates to their memory addresses, which deprives it of the benefits of independent management of individual tiles.

In the current prototype, the matrix is implemented as an object of type `std::map` holding pointers to tiles, and indexed by objects of type `std::tuple<int64_t, int64_t, int>`:

```
std::map<std::tuple<int64_t, int64_t, int>, Tile<FloatType>*> *tiles_;
```

The first two elements of the tuple indicate the location of the tile in the matrix (row, column). The third element indicates the memory space, i.e., OpenMP device number. Notably, multiple replicas of the same tile can exist simultaneously in different memory spaces.

Note: Initially, `std::map` was chosen, for simplicity. However, the standard implementation of `std::map` relies on a red-black tree, with $O(log\ n)$ complexity, which is a performance concern. Although this has not prevented good performance numbers so far, it is a reason for concern in the long run. The simple solution is the replacement of `std::map` with `std::unordered_map`, which is of $O(1)$ complexity, the caveat being that we need to provide the hashing function for the keys (`std::tuple<int64_t, int64_t, int>`), which may require some careful consideration.

Another important feature of the SLATE matrix is that, in the distributed memory environment, it provides a local view of the distributed matrix, i.e.:

- It relies on global indexing of tiles, meaning that each tile is identified by the same unique tuple across all processes.

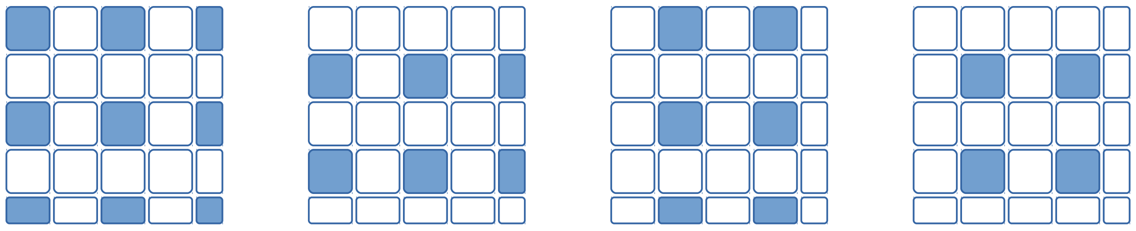

Figure 2.2: 2D block cyclic mapping of a matrix to 4 processes arranged in a 2×2 grid. Hollow boxes indicate tiles that are not created and do not consume memory.

- Each process only allocates its portion of the global matrix, following some predefined distribution, e.g., 2D block cyclic. (Figure 2.2).
- Remote tiles are accessed by creating tiles in the local matrix, at their appropriate coordinates, and bringing over the data by the means of message passing. Figure 2.3 shows a symmetric (positive definite) matrix in the process of Cholesky factorization by 4 processes in a 2×2 arrangement. Blue tiles are owned by process 0. Yellow tiles are brought in from process 2 for applying the transformations of the second panel factorization.

The global indexing of tiles has the advantage of presenting the developer with a simple SPMD programming model, not too different from the serial (superscalar) programming model of the PLASMA library.

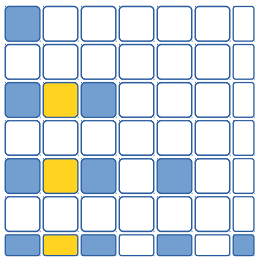

Figure 2.3: Permanent tiles (blue) and transient tiles (yellow).

The distribution of tiles to processes is specified by a globally defined function. By default, SLATE provides the standard 2D block cyclic distribution, but the user can supply an arbitrary mapping function. Similarly, in the case of offload to accelerators, distribution to multiple accelerators is specified by a mapping function. By default, SLATE applies 1D block cyclic distribution to the local tiles, but the user can replace it with an arbitrary function.

Finally, SLATE does not require uniform tile sizes. The only requirement is that diagonal tiles are square (Figure 2.4). Although the current prototype is not generalized to that extent, there are no particular difficulties in supporting rectangular tiles. This will facilitate in the future the use of the SLATE infrastructure for developing, e.g., *Adaptive Cross Approximation* (ACA) linear solvers. Similarly to the distribution of tiles, the row heights and column widths have to be specified by a globally defined function.

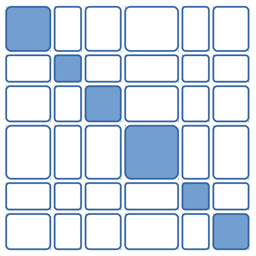

Figure 2.4: Rectangular tiling.

2.2 Class Structure

The design of SLATE revolves around two classes: `Matrix` and `Tile`. The `Tile` is intended as a simple class for maintaining the state of individual tiles and implementing rudimentary (serial) tile operations, while the `Matrix` class is intended as a much more complex class, maintaining the state of distributed matrices throughout the execution of parallel matrix algorithms in distributed memory environments.

Note: The code snippets presented in this report come from the prototype of the Cholesky factorization, developed in the course of the design phase, and are intended for illustration of the design ideas and not the final product. Specifically, many low level implementation details are omitted for brevity, such as locks. Also, at this stage the code is void of any inheritance structure, which is intended in the final product.

The following is a simplified code snippet of the `Tile` class:

```
#ifndef SLATE_TILE_HH
#define SLATE_TILE_HH

#include <blas.hh>
#include <lapack.hh>
...

namespace slate {

template<typename FloatType>
class Tile {
public:
    Tile(int64_t mb, int64_t nb);
    Tile(const Tile<FloatType> *src_tile, int dst_device_num);
    ~Tile();

    void gemm(blas::Op transa, blas::Op transb, FloatType alpha,
              Tile<FloatType> *a, Tile<FloatType> *b, FloatType beta);

    void syrk(blas::Uplo uplo, blas::Op trans,
              FloatType alpha, Tile<FloatType> *a, FloatType beta);

    void trsm(blas::Side side, blas::Uplo uplo, blas::Op transa,
              blas::Diag diag, FloatType alpha, Tile<FloatType> *a)
    ...

    int64_t mb_; ///< tile height
    int64_t nb_; ///< tile width

    FloatType *data_; ///< tile data
    ...

private:
    static int host_num_;   ///< OpenMP initial device number
           int device_num_; ///< OpenMP device number
    ...
};

template<typename FloatType>
int Tile<FloatType>::host_num_ = omp_get_initial_device();

} // namespace slate

#endif // SLATE_TILE_HH
```

The code illustrates the following design decisions:

Lines 4, 5: The implementation of the `Tile` class is based on the C++ APIs for BLAS and LAPACK, being developed simultaneously in the course of the SLATE project[3]. The primary objective of those APIs is the use of templating for handling of multiple precisions (single/double, real/complex) [10].

Line 10: The `Tile` class is templated for supporting multiple precisions, initially the basic four precisions of LAPACK and ScaLAPACK, but possibly also extended precisions, such as double-double or triple-float, or lower precisions, such as the half precision (commonly referred to as FP16)[4].

Line 13, 14: The class provides a basic set of constructors. A simple constructor creates a tile with specific dimensions in the host memory (the memory of the "initial device"). A copy constructor creates a copy of the tile in the memory space of another device.

Line 17, 24: The class implements a set of methods for performing basic dense linear algebra operations, matrix (tile) multiplication, symmetric rank-k update, triangular solve, etc. These methods basically forward the call to the appropriate routines of BLAS and LAPACK through their respective C++ APIs. They also implement auxiliary tasks, such as enforcement of atomic access through locks, critical sections, etc. They are also intended to include comprehensive error checks, such as checks of parameter dimensions, checks for null pointers, etc.

Line 27-30: The class stores basic information about the tile, such as its dimensions and the pointer to its data. This information may become more complex, as polymorphism is introduced to handle different types of tiles, e.g., low-rank compressed tiles.

Line 34, 35: The class stores the tile's location, i.e., the OpenMP device number where the tile data is located. A class (static) field stores the device number identifying the host, referred to as the "initial device" in the OpenMP nomenclature, so that the `omp_get_initial_device()` function does not have to be invoked every time the host needs to be identified. The field is initialized in line 40.

The following is a simplified code snippet of the `Matrix` class:

```
#ifndef SLATE_MATRIX_HH
#define SLATE_MATRIX_HH

#include "slate_Tile.hh"

#include <algorithm>
#include <functional>
#include <map>
#include <set>
#include <vector>
...

#include <mpi.h>
#include <omp.h>

```

[3] http://www.icl.utk.edu/publications/swan-002
[4] https://en.wikipedia.org/wiki/Half-precision_floating-point_format

```
16 namespace slate {
17
18 template<typename FloatType>
19 class Matrix {
20 public:
21     Matrix(int64_t m, int64_t n, FloatType *a, int64_t lda,
22            int64_t mb, int64_t nb);
23     Matrix(int64_t m, int64_t n, FloatType *a, int64_t lda,
24            int64_t mb, int64_t nb, MPI_Comm mpi_comm, int64_t p, int64_t q);
25     Matrix(const Matrix &a, int64_t it, int64_t jt, int64_t mt, int64_t nt);
26     ~Matrix();
27
28     void potrf(blas::Uplo uplo);
29     void trsm(blas::Side side, blas::Uplo uplo,
30               blas::Op trans, blas::Diag diag,
31               FloatType alpha, const Matrix &a);
32     ...
33
34 private:
35     Tile<FloatType>* &operator()(int64_t i, int64_t j) {
36         return (*tiles_)[{it_+i, jt_+j, host_num_}];
37     }
38     Tile<FloatType>* &operator()(int64_t i, int64_t j) const {
39         return (*tiles_)[{it_+i, jt_+j, host_num_}];
40     }
41     Tile<FloatType>* &operator()(int64_t i, int64_t j, int device) {
42         return (*tiles_)[{it_+i, jt_+j, device}];
43     }
44     Tile<FloatType>* &operator()(int64_t i, int64_t j, int device) const {
45         return = (*tiles_)[{it_+i, jt_+j, device}];
46     }
47
48     int64_t it_; ///< first row of tiles
49     int64_t jt_; ///< first column of tiles
50     int64_t mt_; ///< number of tile rows
51     int64_t nt_; ///< number of tile columns
52
53     std::map<std::tuple<int64_t, int64_t, int>, Tile<FloatType>*> *tiles_;
54
55     MPI_Comm mpi_comm_;
56
57     int mpi_size_;
58     int mpi_rank_;
59
60     static int host_num_;    ///< OpenMP initial device number
61            int num_devices_; ///< OpenMP number of devices
62
63     std::function <int64_t (int64_t i, int64_t j)> tileRankFunc;
64     std::function <int64_t (int64_t i, int64_t j)> tileDeviceFunc;
65     std::function <int64_t (int64_t i)> tileMbFunc;
66     std::function <int64_t (int64_t j)> tileNbFunc;
67 };
```

The code illustrates the following design decisions:

Lines 4-14: The `Matrix` class is implemented based on the `Tile` class, employs a set of *Standard Template Library* (STL) containers, and relies on MPI and OpenMP for handling message passing and node-level parallelism.

Lines 21-25: The `Matrix` class provides a standard set of constructors. In a single node scenario, the SLATE matrix can be built from a LAPACK matrix, given the pointer and the leading dimension of the LAPACK matrix, and the intended tiling factors for the SLATE matrix. In a distributed memory scenario, SLATE also requires the MPI communicator and the dimensions of the process grid (P and Q), if the 2D block cyclic distribution is the intention. Arbitrary, user-defined, mapping functions will also be supported.

Lines 28-31: The class provides a set of methods implementing the main functionality of the SLATE package: parallel BLAS, parallel norms, parallel matrix factorizations and solves, etc.

Lines 35-46: The class overloads the parentheses operator to provide access to tiles based on their coordinates. I.e., for a matrix object `"a"`, the expression `a(i, j)` returns the pointer to the tile located in row `i` and column `j`. The optional third parameter allows for accessing the tile in a specified device memory. If omitted, the host memory is assumed.

Lines 48-51: Each matrix can be a submatrix within a larger matrix (Figure 2.5) The `it_` and `jt_` fields contain the vertical and horizontal offsets from the beginning of the main matrix, respectively. The `mt_` and `nt_` fields contain the size of the (sub)matrix. The values indicate the number of tiles, not elements. The fields have identical names and meaning to the ones in the matrix descriptor in the PLASMA library.

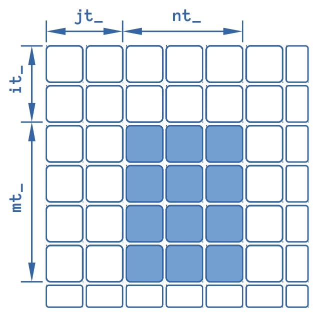

Figure 2.5: Matrix dimensions.

Lines 53: As already described in Section 2.1, the matrix is a collection of tiles, stored as an object of type `std::map` (alternatively `std::unordered_map`), and indexed by tile coordinates in the matrix and the device number. Unlike in other linear algebra packages, tile coordinates have no relation to the tile's location in the memory.

Lines 55-61: The class contains the basic MPI information, such as the MPI communicator, the MPI size (number of participating processes), and the MPI rank (the index of the local process), as well as the basic OpenMP information, such as the number identifying the host and the number of devices.

Lines 63-66: Finally, the class contains the functions describing the partitioning and tiling of the matrix. `tileRankFunc` defines the mapping of tiles to the MPI ranks, based on the tiles' coordinates, while `tileDeviceFunc` defines the mapping of tiles to devices. `tileMbFunc` defines the height for a row of tiles, based on its vertical location, while `tileNbFunc` defines the width of a column of tiles, based on its horizontal location.

Notably, there seems to be no need for a separate structure to maintain state. Traditionally, software libraries introduce a designated object to preserve state (`Handle`, `Context`, etc.). In SLATE, it looks like the `Matrix` is perfectly suitable for that purpose, and there is no need for a separate structure. E.g., right now, the `Matrix` stores the MPI communicator and the process grid dimensions, which would traditionally be stored by an object such as `Handle` or `Context`.

2.3 Model of Parallelism

The cornerstones of SLATE are: 1) the SPMD programming model for productivity and maintainability, 2) dynamic task scheduling using OpenMP for maximum node-level parallelism and portability, 3) the technique of *lookahead* for prioritization of the *critical path*, 4) non-blocking messaging using MPI for communication overlapping, 5) primarily reliance on the 2D block cyclic distribution for scalability, 6) reliance on the GEMM operation, specifically its batch rendition, for maximum hardware utilization.

The Cholesky factorization prototype, developed in the course of this work, established the basic framework for the development of other routines. Figure 2.6 illustrates the main principles. Dataflow tasking (`omp task depend`) is used for scheduling operations on large blocks of the matrix (large boxes connected with arrows), and nested tasking (`omp task`) is used for scheduling individual tile operations to individual cores. At the same time, batch BLAS calls are used for fast processing of large blocks of the matrix using powerful devices, such as GPU accelerators or large numbers of cores of the Xeon Phi processors.

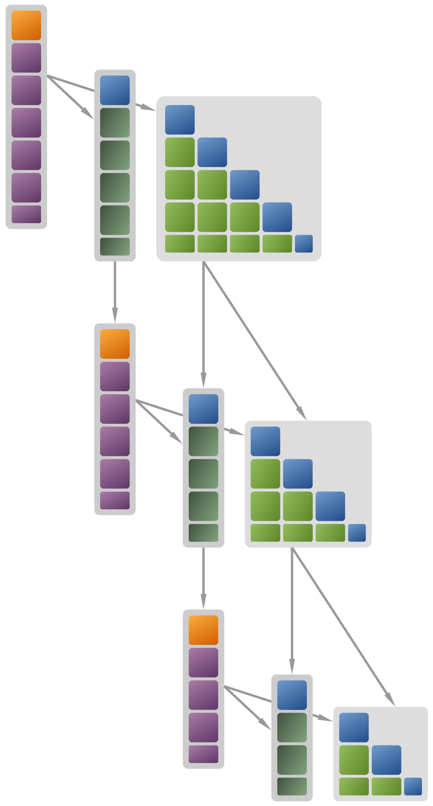

Figure 2.6: Dynamic tasking.

This approach is superior to dataflow scheduling on a tile by tile basis. For a matrix of $N \times N$ tiles, tile by tile scheduling creates $O(N^3)$ dependencies. Combined with large numbers of cores, this often leads to catastrophic scheduling overheads. This is one of the main performance handicaps of the OpenMP version of the PLASMA library [15], specifically in the case of processors with large numbers of cores, such as the Xeon Phi family [9]. In contrast, the SLATE approach creates $O(N)$ dependencies, instead of $O(N^3)$, which completely eliminates the issue of scheduling overheads. At the same time, this solution is a necessity for scheduling large bundles of tile operations to accelerators.

One or more columns of the trailing submatrix are singled out for prioritized processing to facilitate faster advance along the critical path, i.e., to implement the lookahead. Prioritization of tasks can be accomplished using the OpenMP `priority` clause. At the same time, the depth of the lookahead needs to be limited, as it is proportional to the size of the extra memory required for communication buffers. Deep lookahead translates to depth-first processing, synonymous with left-looking algorithms, which can provide scheduling benefits in shared memory environments, but can also lead to catastrophic memory overheads in distributed memory environments, which was a painful lesson of the PULSAR project [13].

Distributed memory computing is implemented by filtering the tile operations through the matrix mapping function (Section 2.1) and issuing appropriate communication calls (Section 2.4) in the course of the computation. Management of separate memories of multiple accelerators is handled by a node-level memory consistency protocol (Section 2.5).

2.4 Message Passing Communication

Communication in SLATE relies on explicit dataflow information. When a tile is modified by a process, and the data needs to be propagated across other processes, a call is issued to a function that initiates the communication. A synchronizing call is required on the receiving side, before the tile data can be accessed.

Communication is done on a tile basis. The initiating call specifies coordinates of the tile that needs to be propagated, and the boundaries of the matrix region where the tile will be applied. The completion call specifies the coordinates of the tile that is being received, and is basically synonymous with the *wait* operation.

Consider the step of the Cholesky factorization shown in Figure 2.7. After the factorization of the first panel, the panel tiles will be applied across the trailing submatrix. Tile $(2, 0)$ will be applied to the right, and its mirror image (transposition) will be applied to the second column of the matrix (the lower part). The initiating call has the form: `tileSend(2, 0, {2, 2, 1, 2}, {2, 6, 2, 2})`, and the receiving calls have the form: `tileSync(2, 1)`, `tileSync(2, 2)`, `tileSync(3, 2)`, etc. In the C++ implementation, the ranges are passed as braced initializer lists of type `std::array<int64_t, 4>`.

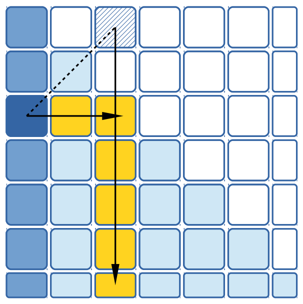
Figure 2.7: Tile based communication.

The actual set of processes participating in the exchange is computed in the `tileSend` function, based on the function mapping tiles to ranks. The *send* function builds the list of participating processes and initiates the communication. Also, on the receiving side, it populates the local matrix with the tiles meant for reception of the remote data. These *transient tiles* are created with a lifespan, which is atomically decremented on each access, so that the tile can be discarded when it is used up. This approach is a necessity in the case of dynamic scheduling, as there is no clear completion point, when the transient tiles can be safely discarded. At the same time, preserving them for too long, past their usefulness, leads to excessive memory usage.

Most of the communication in SLATE will be of multicast nature, and the non-blocking collectives of MPI 3 seem to be the natural choice. Currently, however, they suffer from a serious shortcoming, as discussed in the following section. Because of that, a temporary alternative implementation, using point-to-point communication, is also discussed.

2.4.1 Ibcast/Wait Implementation

In the general case of arbitrary mapping of tiles to the processes' ranks, basically a multicast operation needs to be performed. This can be done by creating an MPI group of participating processes, then creating a subcommunicator for that group, and then using the broadcast operation across that communicator. This can be accomplished with the following set of MPI functions:

```
int MPI_Group_incl(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup);

int MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm *newcomm);

int MPI_Comm_rank(MPI_Comm comm, int *rank);

int MPI_Group_translate_ranks(MPI_Group group1, int n, const int ranks1[],
                              MPI_Group group2, int ranks2[]);

int MPI_Ibcast(void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm, MPI_Request *request);

int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

First, the `MPI_Group_incl()` function is used to create the group of participating processes. Then the `MPI_Comm_create_group()` is used to create a subcommunicator for that group. Then each process can use `MPI_Comm_rank()` to find out its own rank in the new communicator, and `MPI_Group_translate_ranks()` to find out the rank of the broadcast root. At this point, `MPI_Ibcast()` can be used to initiate the broadcast, which can be completed with the usual `MPI_Wait()` function.

The problem with this, fairly straightforward, scenario is that, while `MPI_Ibcast()` is non blocking, `MPI_Comm_create_group()` is blocking, and it is actually blocking across the main communicator, which completely nullifies the advantages of `MPI_Ibcast()`. At this point, it is not clear if the blocking behavior of `MPI_Comm_create_group()` is a fundamental restriction or an implementation artifact. As long as `MPI_Comm_create_group()` blocks, `MPI_Ibcast()` cannot really be taken advantage of, unless a communicator created once can be used multiple times. Unfortunately, this is not the case for SLATE, if arbitrary mapping of tiles to ranks is allowed.

2.4.2 Isend/Irecv/Wait Implementation

Due to the aforementioned shortcomings of MPI's non-blocking collective communication, we implemented a temporary solution, based on MPI's non-blocking point-to-point communication, utilizing the most basic set of MPI functions:

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request);

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request);

int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

This means that the originating process sends the message in sequence to all the recipients. While this is far form optimal, it is also not overly detrimental, as with the 2D block cyclic distribution each tile is sent to roughly $\sqrt{P}$ destinations, where P is total number processes.

At this stage, optimal broadcast patterns are much less critical than non-blocking properties. Eventually, this issue will have to be address when targeting really large scale problems. The ideal solution is implementation of truly non-blocking multicast functionality in MPI libraries.

2.5 Node Level Memory Consistency Model

A couple of different solutions are available for dealing with the complexity of node-level memory architecture, i.e., dealing with separate physical memories of multiple hardware accelerators. The most viable options include the use of CUDA managed memory, OpenMP directives, and OpenMP offload API. The following sections discuss the main aspects of each solution.

2.5.1 CUDA Managed Memory

NVIDIA CUDA offers a set of functions for managing memory using the Unified Memory system[5], which automatically transfers memory between the host memory and the memories of multiple devices, as needed, on a page basis:

```
__host__  cudaError_t
cudaMallocManaged(void** devPtr, size_t size, unsigned int flags=cudaMemAttachGlobal);

__host__ __device__
cudaError_t cudaFree(void* devPtr);

__host__  cudaError_t
cudaMemAdvise(const void* devPtr, size_t count, cudaMemoryAdvise advice, int device);

__host__  cudaError_t
cudaMemPrefetchAsync(const void* devPtr, size_t count, int dstDevice, cudaStream_t stream=0);
```

Managed memory is allocated using the `cudaMallocManaged()` function and freed using the `cudaFree()` function. The `cudaMemAdvise()` function can be used to hint the usage pattern, and the `cudaMemPrefetchAsync()` function can be used to prefetch.

`cudaMemAdvise()` can be used to advise the Unified Memory subsystem about the usage pattern for the memory range starting at the specified address and extending to `size` bytes. The start address of the memory range will be rounded down, and the end address rounded up, to be aligned to the CPU page size, before the advice is applied. The `cudaMemAdvise()` funciton allows for specifying if the memory will be mostly read or written and what is the preferred location for that memory.

`cudaMemPrefetchAsync()` prefetches memory to the specified destination device. `devPtr` is the base device pointer of the memory to be prefetched and `dstDevice` is the destination device. `count` specifies the number of bytes to copy, and `stream` is the stream in which the operation is enqueued. If there is insufficient memory to prefetch the desired region, the Unified Memory driver may evict pages from other managed allocations to the host memory in order to make room.

[5] http://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

New generations of NVIDIA GPUs support device memory oversubscription, i.e., replication of the same memory region in different devices. This capability is indicated by a non-zero value for the device attribute `cudaDevAttrConcurrentManagedAccess`. Concurrent managed access can be accomplished by passing `cudaMemAdviseSetReadMostly` to the `cudaMemAdvise()` function. If `cudaMemPrefetchAsync()` is subsequently called on this region, it will create a read-only copy of the data on the destination processor. If any processor writes to this region, all copies of the corresponding page will be invalidated except for the one where the write occurred.

Note: Care needs to be taken when passing managed memory to libraries, particularly MPI. The MPI needs to be CUDA-aware and have explicit support for managed memory. Otherwise, the MPI call might fail or, in the worst case, produce silently wrong results. This may happen when, e.g., the MPI tries to register managed memory for *Remote Direct Memory Access* (RDMA).

2.5.2 OpenMP Directives

Offload to accelerators can be accomplished using OpenMP `#pragma` directives, which allow for assigning work to multiple devices, and moving data to and from their corresponding memories[6]. Most tasks can be accomplished using the following subset:

```
#pragma omp target
#pragma omp target data
    device
    map
        to
        from
        tofrom
        alloc
#pragma omp target enter data
#pragma omp target exit data
#pragma omp target update
```

The `omp target` directive instructs the compiler to execute the enclosed block of code on a device. In the case of multiple devices, the device number can be specified using the `device` clause. The `map` clause allows for explicitly mapping data in host memory to device memory. If the map type is `to` or `tofrom`, then the host data is copied to the device. If the value is `from` or `alloc`, then the device data is not initialized. If the data was created when the target region was encountered, then it is deallocated on exit from the region. If the map type was `from` or `tofrom`, then it is copied to the host memory before it is deallocated.

The `omp target data` directive maps data in host memory to device memory and also defines the lexical scope of the data being mapped, allowing for a reduction of data copies, when multiple target regions are using the same data[7]. The `omp target enter data`, and `omp target enter data`, and `omp target update` clauses provide further flexibility in handling data in a way that minimizes data motion.

[6] https://www.ibm.com/support/knowledgecenter/en/SSXVZZ_13.1.5/com.ibm.xlcpp1315.lelinux.doc/compiler_ref/tuoptppp.html

[7] https://www.ibm.com/support/knowledgecenter/en/SSXVZZ_13.1.5/com.ibm.xlcpp1315.lelinux.doc/compiler_ref/prag_omp_target_data.html

2.5.3 OpenMP API

The OpenMP standard also offers a set of API functions for managing multiple accelerator devices and the data traffic among their corresponding memories[8]. Most task can be accomplished using the following small subset:

```
int omp_get_initial_device(void);

int omp_get_num_devices(void);

void* omp_target_alloc(size_t size, int device_num);

void omp_target_free(void *device_ptr, int device_num);

int omp_target_memcpy(void *dst, void *src, size_t length,
                      size_t dst_offset, size_t src_offset,
                      int dst_device_num, int src_device_num);
```

The `omp_get_initial_device()` function returns a unique number which identifies the host. The number is outside the range from 0 to `omp_get_num_devices()`-1. The value is implementation dependent. GCC returns $-$`omp_get_num_devices()`. The `omp_get_num_devices()` function returns the number of *target* devices, i.e., the number of accelerators, and is basically equivalent to `cudaGetDeviceCount()`.

The `omp_target_alloc()` and `omp_target_free()` functions are used to allocate and free memory respectively. If the host number is passed as `device_num`, then host memory is allocated, if a number between 0 and `omp_get_num_devices()`-1 is passed, then device memory is allocated. This is different from CUDA, where device memory is allocated with `cudaMalloc()`, while host memory can be allocated with one of the standard functions - `malloc()`, `calloc()`, etc. - or `cudaMallocHost()`, if pinned memory allocation is desired.

The `omp_target_memcpy()` function copies memory between pointers, which can be either host or target device pointers. The equivalent CUDA function is `cudaMemcpy()`. In the past `cudaMemcpy()` required the direction of the copy to be specified (either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`. Currently, however, `cudaMemcpy()` also accepts the value `cudaMemcpyDefault`, in which case the direction is deduced from the pointers.

2.5.4 Discussion

The OpenMP API, described in this subsection, is currently the solution of choice. CUDA managed memory, while offering the highest level of convenience and automation, suffers from the obvious shortcoming of being a proprietary solution. There are also concerns about its interoperability with MPI, as mentioned in Section 2.5.1, as well as issues with support across different devices. In principle, OpenMP directives provide a similar level of convenience. At this point, however, it is not clear if they provide an appropriate level of control. Another reason of concern is lagging support of OpenMP offload directives in compilers. Missing API functions can be easily substituted with custom implementations, which is not the case for compiler directives.

[8] https://www.ibm.com/support/knowledgecenter/en/SSXVZZ_13.1.5/com.ibm.xlcpp1315.lelinux.doc/compiler_ref/bifs_omp.html

2.6 Cholesky Example

The following code shows the Cholesky factorization – implemented using the prototype SLATE infrastructure – with somewhat compressed whitespace, but otherwise complete:

```
template<typename FloatType>
void Matrix<FloatType>::potrf(blas::Uplo uplo, int64_t lookahead)
{
  using namespace blas;

  uint8_t *column;
  Matrix<FloatType> a = *this;

  #pragma omp parallel
  #pragma omp master
  for (int64_t k = 0; k < nt_; ++k) {
    #pragma omp task depend(inout: column[k]) // ---- panel factorization and propagation
    {
      if (tileIsLocal(k, k))
        a(k, k)->potrf(uplo);

      if (k < nt_-1)
        tileSend(k, k, {k+1, nt_-1, k, k});

      for (int64_t m = k+1; m < nt_; ++m) {
        #pragma omp task
        if (tileIsLocal(m, k)) {
          tileWait(k, k);
          a.tileMoveToHost(m, k);
          a(m, k)->trsm(Side::Right, Uplo::Lower,
                        Op::Trans, Diag::NonUnit, 1.0, a(k, k));
        }}
      #pragma omp taskwait

      for (int64_t m = k+1; m < nt_; ++m)
        tileSend(m, k, {m, m, k+1, m}, {m, nt_-1, m, m});
    }
    for (int64_t n = k+1; n < k+1+lookahead && n < nt_; ++n) { // ----- lookahead columns
      #pragma omp task depend(in: column[k]) \
                       depend(inout: column[n])
      {
        #pragma omp task
        if (tileIsLocal(n, n)) {
          tileWait(n, k);
          a(n, n)->syrk(Uplo::Lower, Op::NoTrans, -1.0, a(n, k), 1.0);
        }
        for (int64_t m = n+1; m < nt_; ++m) {
          #pragma omp task
          if (tileIsLocal(m, n)) {
            tileWait(m, k);
            tileWait(n, k);
            a.tileMoveToHost(m, n);
            a(m, n)->gemm(Op::NoTrans, Op::Trans, -1.0, a(m, k), a(n, k), 1.0);
          }}
        #pragma omp taskwait
      }}
    if (k+1+lookahead < nt_) { // ------------------------------------ trailing submatrix
      #pragma omp task depend(in: column[k]) \
                       depend(inout: column[k+1+lookahead]) \
                       depend(inout: column[nt_-1])
      Matrix(a, k+1+lookahead, k+1+lookahead, nt_-1-k-lookahead, nt_-1-k-lookahead).syrk(
        Uplo::Lower, Op::NoTrans, -1.0,
        Matrix(a, k+1+lookahead, k, nt_-1-k-lookahead, 1), 1.0);
    }}}
```

The code is intended to illustrate the basic mechanics of the SLATE approach. It is not intended to represent the final solution, as many conventions are expected to change in the course of the project. It is also quite likely that new abstraction layers will emerge as the project progresses.

The first block of code (lines 12-32) factors and propagates the panel. The diagonal block is factored and sent down the column. Then a triangular solve is applied to the tiles in the column, and then each tile is sent in the horizontal direction, while its transposed image is sent in the vertical direction.

The second block of code (lines 33-51) applies the transformations to the *lookahead* number of columns of the trailing submatrix. Diagonal tiles are updated using the `syrk()` method of the `Tile` class, while all the other tiles are updated using the `gemm()` method of the `Tile` class.

The last block of code (lines 52-59) applies the transformations to all the remaining columns of the trailing submatrix using the `syrk()` method of the `Matrix` class. The method relies primarily on batch BLAS routines to dispatch the operations to available devices, either multicore processors or hardware accelerators, and also invokes the necessary local communication for tiles that are not available in the memories of appropriate devices. In the case of matrices that do not fit in the combined memories of all the available accelerators, this method will also be responsible for implementing streaming access to the trailing submatrix.

Memory traffic is explicit – represented by `tileSend()` and `tileWait()` methods of the `Matrix` class for distributed memory communication, and the `tileMoveToHost()` method of the `Matrix` class for bringing tiles back to host memory from device memories. Similar methods are available for moving tiles in the opposite direction (from the host to one of the devices), as well as for making duplicate, read only, copies of tiles in memories of multiple devices.

Although the code has some complexity to it, it is a fairly straightforward and compact representation, especially considering that it supports distributed memory systems and offload to accelerators. In comparison, the PLASMA library involves about the same amount of code to express the Cholesky factorization for shared memory systems only (only multithreading). At the same time, the MAGMA library, contains a separate function for offload to a single GPU with the matrix located in the host memory (`magma_?potrf`), a separate function for offload to a single GPU, with the matrix located in the GPU memory (`magma_?potrf_gpu`), a separate function for offload to multiple GPUs, with the matrix located in the CPU memory (`magma_?potrf_m`), and a separate function for offload to multiple GPUs, with the matrix distributed across the memories of the GPUs (`magma_?potrf_mgpu`). Each of them is at least as long as the SLATE Cholesky routine. MAGMA does not support distributed memory.

CHAPTER 3

Preliminary Performance Results

3.1 Experimental Setup

Preliminary performance results were collected using the *Summitdev* system[1] at the *Oak Ridge National Laboratory* (ORNL), which is intended to mimic the OLCF's next supercomputer Summit. Summitdev is based on IBM POWER8 processors and NVIDIA P100 (Pascal) accelerators, and is one generation behind Summit, which will be based on POWER9 processors and V100 (Volta) accelerators.

The Summitdev system contains three racks, each with 18 IBM POWER8 S822LC nodes, for a total of 54 nodes. Each node contains 2 POWER8 CPUs, 10 cores each, and 4 P100 GPUs. Each node has 256 GB of DDR4 memory. Each GPU has 16 GB of HBM2 memory. The GPUs are connected by NVLink 1.0 at 80 GB/s. The nodes are connected with a fat-tree EDR InfiniBand. The software environment used for the experiments included GCC 7.1.0, CUDA 8.0.54, ESSL 5.5.0, and Spectrum MPI 10.1.0.3.

3.2 Multicore Scaling

This section presents the results of multicore scaling experiments (both *strong scaling* and *asymptotic scaling*). Performance of SLATE is compared to the performance of ScaLAPACK and the theoretical maximum performance, defined as the single node performance of DGEMM multiplied by the number of nodes.

[1]https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/

Basic performance tuning was done for both SLATE and ScaLAPACK. The best setup for SLATE turned out to be one process per socket. I.e., the number of processes is double the number of nodes. The best setup for ScaLAPACK turned out to be one process per core. I.e., the nuber of processes is 20 times the number of nodes. Tuning also involved finding good tiling factors for SLATE, and good blocking factors for ScaLAPACK. The tiling factor of 256 delivered good performance for SLATE across all runs. Optimal blocking factors for ScaLAPACK included 64, 96, and 128. Performance depended heavily on the shape of the process grid for both SLATE and ScaLAPACK. The best numbers are shown in each case.

Figure 3.1 shows the results of the strong scaling experiment, where a matrix of size $40,000 \times 40,000$ is factored using an increasing number of nodes. Figure 3.2 shows the results of the asymptotic scaling experiment, where a matrix of size $20,000 \times 20,000$ is factored using a single node, a matrix of size $40,000 \times 40,000$ is factored using a 4 nodes, and a matrix of size $80,000 \times 80,000$ is factored using 16 nodes. Both SLATE and ScaLAPACK scale fairly well, for the range of problem sizes, with SLATE enjoying a modest performance advantage over ScaLAPACK.

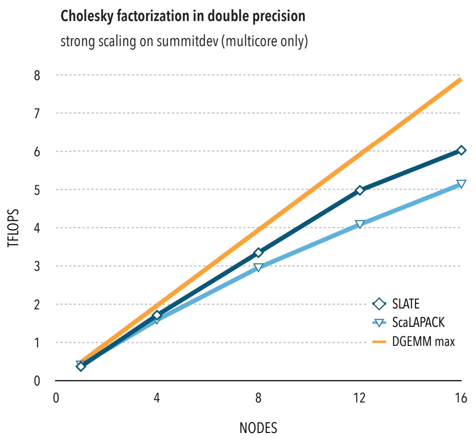

Figure 3.1: Strong scaling using POWER8.

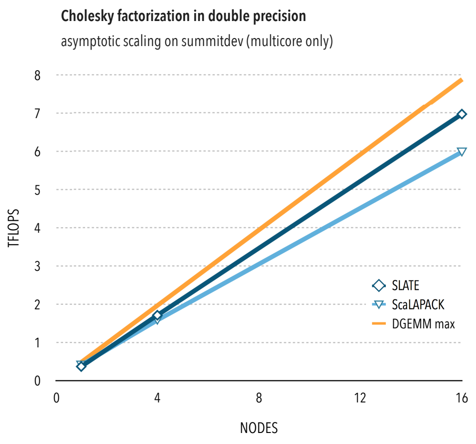

Figure 3.2: Asymptotic scaling using POWER8.

3.3 GPU Scaling

This section presents the results of GPU scaling experiments (single node performance and asymptotic scaling). Basic performance tuning was done for SLATE. The tile size of 512 turned out to deliver good performance across all runs. Also the lookahead of one proved to be the best choice in all cases.

Unfortunately, here, we failed to produce a comparison with ScaLAPACK. To start with, currently, the OLCF facility has no license for the PESSL library. At the same time, we found no evidence of GPU support for the Cholesky factorization in the ESSL library. Finally, we failed to make the Netlib ScaLAPACK call the GPU enabled DGEMM from ESSL. Therefore, no meaningful comparison with ScaLAPACK was possible at this time. We hope to be able to produce such a comparison in near future.

Figure 3.3 shows the performance of single node runs, using a single GPU and all four GPUs. The experiment was run until the GPU memory was exhausted. The largest matrix that fit in the memory of a single GPU was $56,000 \times 56,000$. The largest matrix that fit in the combined memories of all four GPUs was $112,000 \times 112,000$. For a single GPU, top performance slightly exceeded 4 GFLOPS. For four GPUs, top performance slightly exceeded 9 TFLOPS. At this point, serial bottlenecks still prevent the code from utilizing the GPUs to the fullest. Notably, the MAGMA library reaches 12 GFLOPS in the same setup.

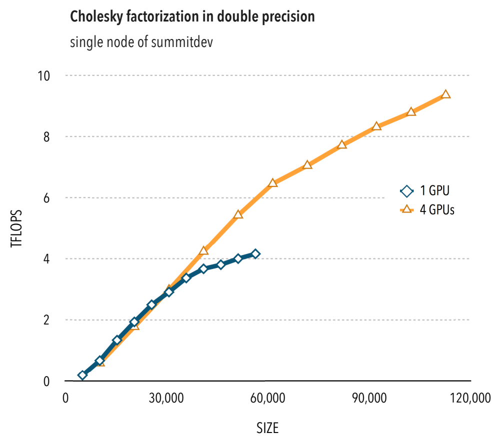

Figure 3.3: Node performance using P100.

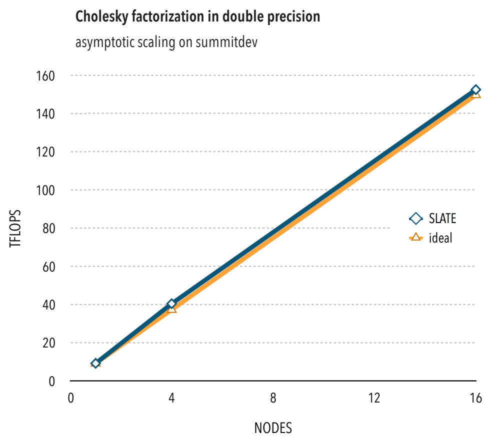

Figure 3.4: Asymptotic scaling using P100.

Figure 3.4 shows asymptotic scaling using one node, four nodes and 16 nodes, i.e., 4 GPUs, 16 GPU, and 64 GPUs, with matrix sizes of $112,000 \times 112,000$, $225,000 \times 225,000$, and $450,000 \times 450,000$. Here, SLATE actually shows a slight superlinear scaling. This is due to the fact that the single node run is not reaching its maximum performance for the matrix size of $112,000 \times 112,000$. Doubling the matrix size for the four nodes run doubles the amount of work *per node*, which minimizes the impact of serial bottlenecks. The same trend continues for the 16 nodes run. SLATE reached 40 TFLOPS on 4 nodes and exceeded 150 TFLOPS on 16 nodes.

3.4 Discussion

Overall, the experimental results, presented in this chapter, seem to be validating the proposed methodology. The prototype code exceeded the performance of ScaLAPACK in distributed memory runs with multicore processors, achieved $\sim$90% of MAGMA's performance for a single GPU run, and $\sim$75% of MAGMA performance for a four GPUs run. At the same time, it showed superlinear scaling from one node to four nodes and 16 nodes. Notably, the 150 TFLOPS performance achieved on 16 nodes, translates to $\sim$80% of the DGEMM peak (single node DGEMM performance of 12 TFLOPS times 16).

It needs to be pointed out that Summitdev is a very unforgiving system, and so will be the upcoming Summit machine. Within each node of Summitdev, the two POWER8 processors have only about $\frac{1}{40}th$ of the performance of the four P100 GPUs. Any computational part not offloaded to the GPUs immediately becomes a bottleneck. At the same time, the NVLink connection does not completely solve the problem of host to device communication. And finally, there is a massive disproportion between the computing power of a node and its network connection. This forced us to blow up the matrix size to almost half a million, in order to get good performance from 64 GPUs.

3.5 Traces

This section presents a handful of traces collected in the course of the preliminary runs. The traces were collected using an ad hoc tracing code, which collects tasks' start and end times with `omp_get_wtime()`, and then prints them to a *Scalable Vector Graphics* (SVG) file when the execution completes. The same color coding is used across all traces. `potrf` tasks are brown, `trsm` tasks are purple, `syrk` tasks are blue, and `gemm` tasks are green. The `gemm` tasks offloaded to GPUs are bright green. Host to device communication is gray, and MPI communication is very light gray. Also, since the complete traces are very long, all figures contain three pieces: a slice at the beginning of the execution, a slice in the middle, and a slice at the end.

Figure 3.5 shows 4 nodes (80 cores) factoring a $25,000 \times 25,000$ matrix wiht the tile size of 256×256. The process grid is 2×2 and the lookahead is one. The execution is fairly smooth – with some small gaps for communication and synchronization – until the end of the factorization, when the trace becomes sparse due to the cores running out of work. Otherwise, communication is mostly hidden and the cores are kept busy most of the time.

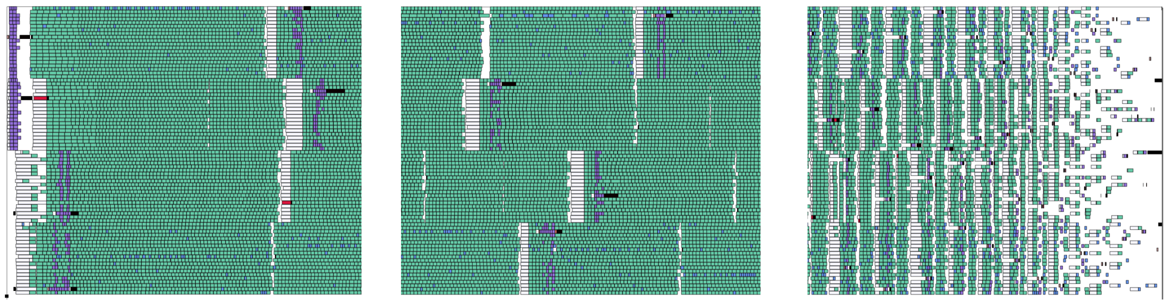

Figure 3.5: Multicore trace.

Figure 3.6 shows 20 cores and one GPU factoring a $56,000 \times 56,000$ matrix with the tile size of 512×512. The lookahead is one. The top stripe shows the beginning of the execution. First, the first diagonal block is factored (brown). Then triangular solve is applied to all the tiles in the first column below the diagonal block (purple). At this point, the lookahead takes effect and the CPUs start executing tasks simultaneously with the GPU. The CPUs update the first trailing column (green), then update all diagonal tiles of the remaining trailing submatrix, then factor the second diagonal block, and then apply triangular solve to all the tiles below the diagonal in the trailing column. At the same time, the GPU updates all the tiles below the diagonal in the entire trailing submatrix (bright green). The first time the GPU accesses the trailing submatrix, the submatrix is transferred from the host memory to the device memory (the long gray bar). Then the trailing submatrix stays resident in the device memory, and only one column is transferred between the host memory and the device memory at a time. In general, the GPU is busy most of the time, until the end of the factorization, when work become scarce. Also, unlike the MPI communication, the host to device communication is currently not hidden. Although this is not a big problem, it will be a target of future optimizations.

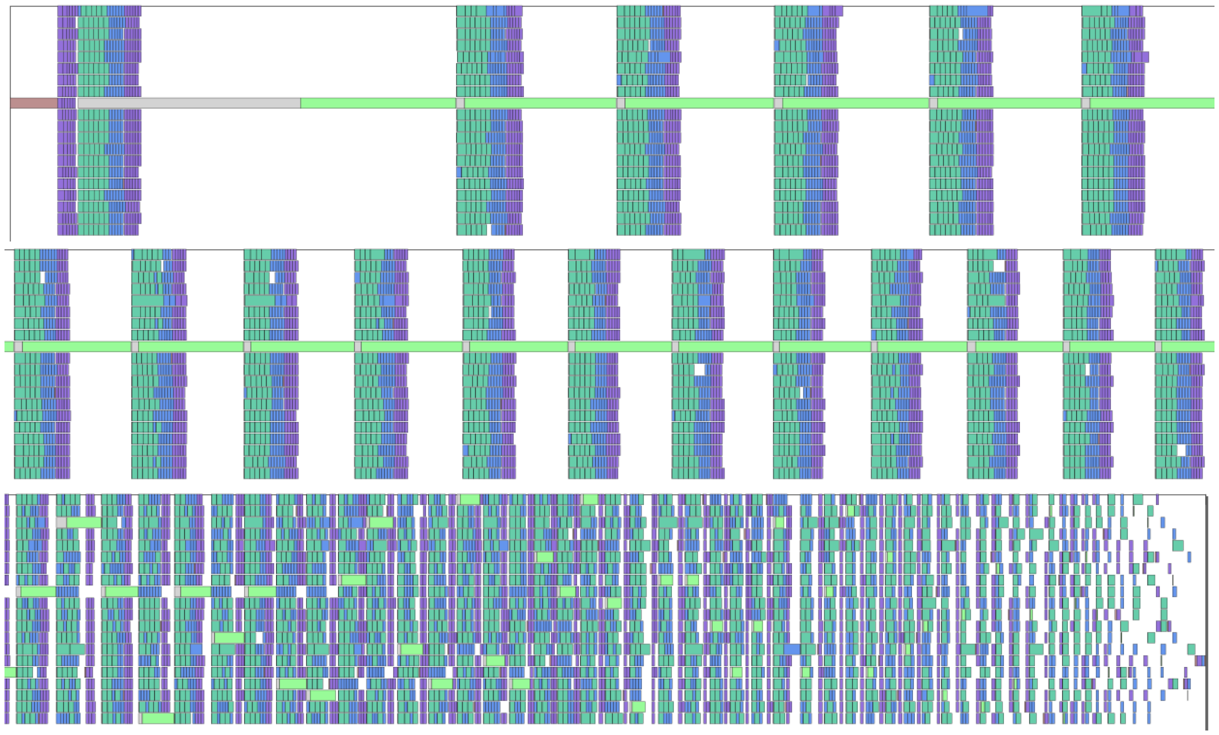

Figure 3.6: Single GPU trace.

Figure 3.7 shows 20 cores and four GPUs factoring a $112,000 \times 112,000$ matrix with the tile size of 512×512. The lookahead is one. The matrix is distributed to the GPUs in a 1D block cyclic fashion. The execution progresses similarly to the execution of the one GPU run. It is clearly noticeable that the GPUs have an asymmetric access to the memory, as the initial transfer of the matrix to the device memory takes about twice as long for one pair of GPU as it takes for the other pair.

The most dramatic observation of this run is the horrendous disproportion of CPU power to GPU power. It takes the CPU a comparable amount of time to deal with one column of the trailing submatrix as it takes the GPUs to deal with more than 200 remaining columns. Also, the host to device communication starts playing a bigger role in the overall performance profile. In the middle of the factorization, the CPUs start overtaking the GPUs in the execution time, and eventually dominate the execution time towards the end.

Distributed memory multi GPU runs required very large matrices to get good performance. At this point, collection and plotting of traces became prohibitively expensive for our makeshift tracing tool. We plan to collect larger traces in the future, either by fixing the scalability bottlenecks or by switching to a production quality tracing tool.

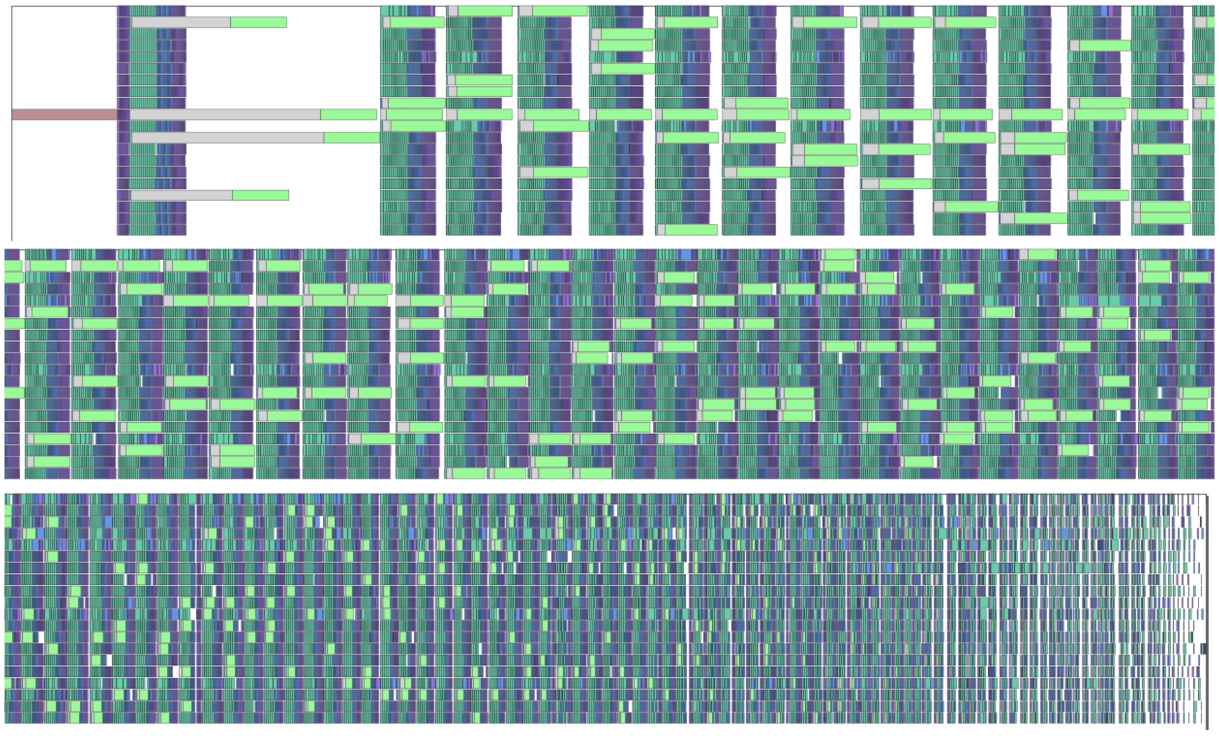

Figure 3.7: Multi GPU trace.

CHAPTER 4

Other Considerations

4.1 Software Engineering

Basic software engineering practices will be followed in the development of the SLATE software. The software is hosted on Bitbucket in the repository https://bitbucket.org/icl/slate, with the C++ APIs for BLAS and LAPACK hosted as independent repositories https://bitbucket.org/icl/blaspp and https://bitbucket.org/icl/lapackpp, and included in SLATE as subrepositories. The Doxygen system[1] will be used for creating a reference manual, and accompanied by entry level documentation developed in Sphinx[2], the documentation generator of Python. We will follow our coding guidelines[3], which are based on established software practices, mostly on the Google C++ Coding Style Guide[4], with some of the important provisions including:

- Following basic formatting rules such as 80 character lines, indentation with 4 spaces, and basic naming conventions, mostly following the Google guide.
- Using precise-width integer types in all cases when the built-in `int` type is insufficient and staying away from unsigned integer types.
- Extensively using 64-bit integers, specifically the `int64_t` type for all mathematical objects, such as sizes of matrices and tiles, offests, coordinates, etc.
- Passing input parameters as values or const references, and passing output and in/out parameters as pointers to non-const.

[1] http://www.doxygen.org
[2] http://www.sphinx-doc.org
[3] https://bitbucket.org/icl/style/wiki/ICL_C_CPP_Coding_Style_Guide
[4] https://google.github.io/styleguide/cppguide.html

- Using OpenMP-style stubs for compiling without certain components, e.g., compiling without MPI for shared memory systems.
- Using C++ exceptions for error handling.

4.2 Development Road Bumps

We hit a number of roadblocks in the process of prototyping SLATE. The more painful ones included:

Lagging compiler support for tasking and offload extensions: Virtually all compilers, other than GCC, lag in support for either the OpenMP tasking directives, or the `omp_target_` functins, or both. All of the compilers we looked at, including XLC, PGI, ICC, and clang, haves some deficiencies. We settled on the use of GCC, which has outstanding support for the tasking extensions, just to discover thread safety issues with its `omp_target_` functions, as described in the following bullet.

Questionable thread safety of offload directives in GCC: We encountered race conditions when working with GCC's `omp_target_` functions. Although, at this point, we cannot be certain that the problem did not come from elsewhere, replacing them with semantically equivalent CUDA calls fixed the problem.

Questionable thread safety of OpenMPI: We experienced problems with the support for the `MPI_THREAD_MULTIPLE` mode in OpenMPI. Surprisingly, though, we had a hard time fixing the issue by placing MPI calls in critical sections. The problem was not fixed until we switched to the Intel MPI on Intel systems and the Spectrum MPI on IBM systems, while keeping the MPI calls in critical sections.

Catastrophic overheads of CUDA memory management: We encountered overheads of CUDA memory management, which can only be described as catastrophic. Attempts of allocating and freeing memory in the course of a factorization completely annihilate the performance. This is an old issue, which has been encountered in the course of the PaRSEC and PULSAR projects, and led to the adoption of custom memory managers. Here, we resorted to the same solution of building a rudimentary memory manager. This fixed the problem of memory management overheads.

Hopefully, all the encountered problems can be resolved in collaboration with the compiler vendors and the GCC developer community, and with the providers of MPI (Intel, IBM), as well as developers of the OpenMPI and MPICH libraries (the ECP OMPI-X and Exa MPI projects). We are also looking forward to the collaboration with the ECP SOLLVE project, which focuses on advancing the OpenMP extensions using the LLVM infrastructure.

Bibliography

[1] Ahmad Abdelfattah, Hartwig Anzt, Aurelien Bouteiller, Anthony Danalis, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Stephen Wood, Panruo Wu, Ichitaro Yamazaki, and Asim YarKhan. Roadmap for the development of a linear algebra library for exascale computing: SLATE: Software for linear algebra targeting exascale. SLATE Working Note 1, Innovative Computing Laboratory, University of Tennessee, June 2017. revision 06-2017.

[2] Bjarne Stig Andersen, Jerzy Waśniewski, and Fred G Gustavson. A recursive formulation of Cholesky factorization of a matrix in packed storage. *ACM Transactions on Mathematical Software (TOMS)*, 27(2):214–244, 2001.

[3] Bjarne Stig Andersen, John A Gunnels, Fred Gustavson, and Jerzy Wasniewski. A recursive formulation of the inversion of symmetric positive definite matrices in packed storage data format. *PARA*, 2:287–296, 2002.

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' Guide*. SIAM, 1999.

[5] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK Users' Guide*. SIAM, 1997.

[6] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441. IEEE, 2011.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault,

and Jack J Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[8] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical Computations with GPUs*, pages 3–28. Springer, 2014.

[9] Joseph Dorris, Jakub Kurzak, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Task-based Cholesky decomposition on Knights Corner using OpenMP. In *International Conference on High Performance Computing*, pages 544–562. Springer, 2016.

[10] Mark Gates, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. C++ API for BLAS and LAPACK. SLATE Working Note 2, Innovative Computing Laboratory, University of Tennessee, June 2017. revision 06-2017.

[11] Fred Gustavson, André Henriksson, Isak Jonsson, Bo Kågström, and Per Ling. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Applied Parallel Computing Large Scale Scientific and Industrial Problems*, pages 195–206, 1998.

[12] Fred Gustavson, Lars Karlsson, and Bo Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software (TOMS)*, 38 (3):17, 2012.

[13] Jakub Kurzak, Piotr Luszczek, Ichitaro Yamazaki, Yves Robert, and Jack Dongarra. Design and implementation of the PULSAR programming system for large scale computing. *Supercomputing Frontiers and Innovations*, 4(1):4–26, 2017.

[14] Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures.* PhD thesis, University of Tennessee, 2012.

[15] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the PLASMA numerical library to the OpenMP standard. *International Journal of Parallel Programming*, 45(3):612–633, 2017.