SLATE WORKING NOTES
5

Parallel BLAS Performance Report

Jakub Kurzak
Mark Gates
Asim YarKhan
Ichitaro Yamazaki
Panruo Wu
Piotr Luszczek
Jamie Finney
Jack Dongarra

Innovative Computing Laboratory

April 10, 2018

ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE

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.

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.

Revision	Notes
03-2018	first publication
04-2018	copy editing

```
@techreport{kurzak2018parallel,
  author={Kurzak, Jakub and Gates, Mark and YarKhan, Asim and
          Yamazaki, Ichitaro and Wu, Panruo and Luszczek, Piotr
          and Finney, Jamie and Dongarra, Jack},
  title={{SLATE} Working Note 5: Parallel {BLAS} Performance Report},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2018},
  month={March},
  number={ICL-UT-18-01},
  note={revision 04-2018}
}
```

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Parallel Basic Linear Algebra Subprograms (PBLAS) [1] [1, 2] is an implementation of the Basic Linear Algebra Subprograms (BLAS) [2] [3, 4] intended for distributed-memory machines. PBLAS provides a computational backbone for the Scalable Linear Algebra PACKage (ScaLAPACK) [3] [5, 6], a distributed-memory implementation of the Linear Algebra PACKage (LAPACK) [4] [7, 8]. PBLAS depends on sequential BLAS operations for local computation and the Basic Linear Algebra Communication Subprograms (BLACS) [5] [9] for communication between nodes.

Software for Linear Algebra Targeting Exascale (SLATE) [6] [10] is being developed as part of the Exascale Computing Project (ECP) [7], which is a collaborative effort between two US Department of Energy (DOE) organizations, the Office of Science and the National Nuclear Security Administration (NNSA). The purpose of SLATE is to serve as a replacement for ScaLAPACK for the upcoming pre-exascale and exascale DOE machines. SLATE will accomplish this objective by leveraging recent progress in parallel programming models and by strongly focusing on supporting hardware accelerators.

This report focuses on the SLATE project's first batch of computational routines, which implement Level 3 PBLAS. Specifically, initial SLATE PBLAS performance numbers are reported, alongside ScaLAPACK performance numbers on the SummitDev machine at the Oak Ridge Leadership Computing Facility (OLCF). More details about the design of the SLATE software infrastructure can be found in the report by Kurzak et al. [10].

[1] http://www.netlib.org/scalapack/pblas_qref.html
[2] http://www.netlib.org/blas/
[3] http://www.netlib.org/scalapack/
[4] http://www.netlib.org/lapack/
[5] http://www.netlib.org/blacs/
[6] http://icl.utk.edu/slate/
[7] https://www.exascaleproject.org

Name	Description
gemm	Computes a matrix-matrix product with general matrices.
symm	Computes a matrix-matrix product where one input matrix is symmetric.
hemm	Computes a matrix-matrix product where one input matrix is Hermitian.
syrk	Performs a symmetric rank-k update.
herk	Performs a Hermitian rank-k update.
syr2k	Performs a symmetric rank-2k update.
her2k	Performs a Hermitian rank-2k update.
trmm	Computes a matrix-matrix product where one input matrix is triangular.
trsm	Solves a triangular matrix equation.

Table 1.1: Descriptions of Level 3 PBLAS routines.

Table 1.1 lists the Level 3 PBLAS routines and provides their descriptions. Table 1.2 contains their mathematical definitions. Table 1.3 provides the number of floating-point operations for each routine. The numbers are for real arithmetic. In complex arithmetic, the number of floating-point operations is four times higher. All execution rates reported in Section 3.2 were calculated using the formulas from Table 1.3.

The ultimate source of information about BLAS is the Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. [8] Another recommended read is the article by Blackford et al. [11]. A reference implementation of BLAS in Fortran is available from Netlib. [9] Also available from Netlib is a quick reference guide. [10]

Unlike dense linear algebra packages developed in Fortran and C, SLATE does not provide a separate routine for each of the four precisions (single [S], double [D], single complex [C], double complex [Z]). Instead, SLATE relies on C++ templates and overloading, as described in Section 3.2.3. Appendix A provides function signatures of all SLATE PBLAS routines.

Another notable difference is the lack of the `uplo` and `trans` parameters in the function signatures. The information about whether the matrix is upper or lower is a property of the `Matrix` object. So is the information about whether transposition needs to be applied when operating on the matrix, as further explained in Section 2.2.

[8] http://www.netlib.org/blas/blast-forum/blas-report.pdf
[9] http://www.netlib.org/blas/blas-3.8.0.tgz
[10] http://www.netlib.org/blas/blasqr.pdf

Name	Operation	Matrices
gemm	$C \leftarrow \alpha op(A)op(B) + \beta C$	$C : m \times n$, $op(A) : m \times k$
symm	$C \leftarrow \alpha AB + \beta C$ or	$C : m \times n$, $A = A^T$
	$C \leftarrow \alpha BA + \beta C$	
hemm	$C \leftarrow \alpha AB + \beta C$ or	$C : m \times n$, $A = A^H$
	$C \leftarrow \alpha BA + \beta C$	
syrk	$C \leftarrow \alpha op(A)op(A)^T + \beta C,$	$C = C^T$, $op(A) : n \times k$
herk	$C \leftarrow \alpha op(A)op(A)^H + \beta C$	$C = C^H$, $op(A) : n \times k,$
syr2k	$C \leftarrow \alpha op(A)op(B)^T + \alpha op(B)op(A)^T + \beta C$	$C = C^T$, $op(A) : n \times k$
her2k	$C \leftarrow \alpha op(A)op(B)^H + \bar{\alpha} op(B)op(A)^H + \beta C$	$C = C^H$, $op(A) : n \times k$
trmm	$B \leftarrow \alpha op(A)B$ or	A triangular, $B : m \times n$
	$B \leftarrow \alpha Bop(A)$	
trsm	$B \leftarrow \alpha op(A)^{-1}B$ or	A triangular, $B : m \times n$
	$B \leftarrow \alpha Bop(A)^{-1}$	

Table 1.2: Operations performed by the Level 3 PBLAS routines. $op(X) = X$, X^T, or X^H.

Name	Number of Floating Point Operations
gemm	$2mnk$
symm	$2m^2n$ (side=Left), $2mn^2$ (side=Right)
syrk	$kn(n + 1)$
syr2k	$2kn^2 + n$
trmm	nm^2 (side=Left), mn^2 (side=Right)
trsm	nm^2 (side=Left), mn^2 (side=Right)

Table 1.3: Number of floating-point operations performed by the Level 3 PBLAS routines.

CHAPTER 2

Implementation

The principles of the SLATE software framework were laid out in SLATE Working Note 3 [1] [10]. SLATE's design relies on the following principles:

- The matrix is represented as a set of individual tiles with no constraints on their locations in memory with respect to one another. Any tile can reside anywhere in memory and have any stride. Notably, a SLATE matrix can be created from a LAPACK matrix or a ScaLAPACK matrix without making a copy of the data.
- Node-level scheduling relies on nested Open Multi Processing (OpenMP) tasking, with the top level responsible for resolving data dependencies and the bottom level responsible for deploying large numbers of independent tasks to multi-core processors and accelerator devices.
- The Message Passing Interface (MPI) is used for message passing with emphasis on collective communication, with the majority of communication being cast as broadcasts.
- Batch BLAS is used extensively for maximum node-level performance. Most routines spend the majority of their execution in the call to batch `gemm`.

Also, the use of a runtime scheduling system, such as the Parallel Runtime Scheduling and Execution Controller (PaRSEC) [2] [12] or Legion [3,4] [13], is currently under investigation.

[1]http://www.icl.utk.edu/publications/swan-003
[2]http://icl.utk.edu/parsec/
[3]http://legion.stanford.edu
[4] http://www.lanl.gov/projects/programming-models/legion.php

2.1 Matrix Class Hierarchy

SLATE has the matrix classes below. The SLATE BLAS routines require the correct matrix types for their arguments, but inexpensive shallow copy conversions exist between the various matrix types. For instance, a general `Matrix` can be converted to a `TriangularMatrix` for doing a triangular solve (`trsm`).

BaseMatrix Abstract base class for all matrices.

- **Matrix** General, $m \times n$ matrix.
- **BaseTrapezoidMatrix** Abstract base class for all upper or lower trapezoid storage, $m \times n$ matrices. For upper, tiles $A(i, j)$ for $i \leq j$ are stored; for lower, tiles $A(i, j)$ for $i \geq j$ are stored.
  - **TrapezoidMatrix** Upper or lower trapezoid, $m \times n$ matrix; the opposite triangle is implicitly zero.
    - **TriangularMatrix** Upper or lower triangular, $n \times n$ matrix.
  - **SymmetricMatrix** Symmetric, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is implicitly known by symmetry ($A_{j,i} = A_{i,j}$).
  - **HermitianMatrix** Hermitian, $n \times n$ matrix, stored by its upper or lower triangle; the opposite triangle is implicitly known by symmetry ($A_{j,i} = \bar{A}_{i,j}$).

The `BaseMatrix` class stores the matrix dimensions; whether the matrix is upper, lower, or general; whether it is not transposed, transposed, or conjugate-transposed; how the matrix is distributed; and the set of tiles.

Copying a matrix object is an inexpensive shallow copy, with a reference-counted C++ shared pointer to the actual data. Sub-matrices are also implemented by creating an inexpensive shallow copy, with the matrix object storing the offset from the top-left of the original matrix and the transposition operation with respect to the original matrix.

Transpose and conjugate-transpose are supported by creating an inexpensive shallow copy and changing the transposition operation flag stored in the new matrix object. For a matrix `A` that is possibly a transposed copy of an original matrix `A0`, the function `A.op()` returns `Op::NoTrans`, `Op::Trans`, or `Op::ConjTrans`, indicating whether `A` is not transposed, transposed, or conjugate-transposed, respectively. The functions `A = transpose(A0)` and `A = conj_transpose(A0)` return new matrices with the operation flag modified appropriately. Querying a matrix object takes the transposition and sub-matrix offsets into account. For instance, `A.mt()` is the number of block rows of $op(A_0)$, where $op(A_0) = A_0$, A_0^T, or A_0^H. The function `A(i, j)` returns the i, j-th tile of $op(A_0)$, with the tile's operation flag set to match the `A` matrix.

SLATE supports upper and lower storage with `A.uplo()` returning `Uplo::Upper` or `Uplo::Lower`. Tiles likewise have a flag indicating upper or lower storage, accessed by `A(i, j).uplo()`. For tiles on the matrix diagonal, the `uplo` flag is set to match the matrix, while for off-diagonal tiles it is set to `Uplo::General`.

2.2 Handling of side, uplo, trans

The classical BLAS routines take parameters such as `side`, `uplo`, `trans` (named "`op`" in SLATE), and `diag` to specify operation variants. Traditionally, this has meant that implementations have numerous cases. The reference BLAS has nine cases in `zgemm` and eight cases in `ztrmm` (times several sub-cases). ScaLAPACK and the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) likewise have eight cases in `ztrmm`.

In SLATE, both `uplo` and `op` are stored within the matrix object itself, and it supports inexpensive shallow copy transposition. This means we can implement just one or two cases and map all the other cases to that implementation by appropriate transpositions.

2.2.1 gemm

The high-level SLATE `gemm` implements only one case:

$$C \leftarrow \alpha AB + \beta C$$

To obtain other cases, the matrices A or B can be (conjugate) transposed before the call. For instance:

```
    slate::gemm( alpha, transpose(A), conj_transpose(B), beta, C );
```

At the high level, `gemm` can ignore the operations on A and B. The matrix object, if transposed, handles swapping indices to obtain the correct tiles during the algorithm. At the low level, the transposition operation is set on the tiles, and is passed on to the underlying BLAS `gemm` routine.

2.2.2 syrk, syr2k, herk, her2k

For rank k and rank $2k$ updates, SLATE implements only one case each:

$$\begin{aligned} C &\leftarrow \alpha AA^T + \beta C && \texttt{syrk}, \\ C &\leftarrow \alpha AA^H + \beta C && \texttt{herk}, \\ C &\leftarrow \alpha AB^T + \alpha BA^T + \beta C && \texttt{syrk}, \\ C &\leftarrow \alpha AB^H + \bar{\alpha} BA^H + \beta C && \texttt{herk}, \end{aligned}$$

with the symmetric or Hermitian matrix C stored lower. As with `gemm`, the A or B matrices can be (conjugate) transposed beforehand to obtain other cases. To handle upper storage matrices, SLATE internally (conjugate) transposes C.

2.2.3 symm, hemm

The high-level SLATE `symm` and `hemm` implement the two cases where the symmetric or Hermitian matrix A is stored as lower or as upper, and is on the left,

$$C \leftarrow \alpha AB + \beta C.$$

To handle when A is on the right,

$$C \leftarrow \alpha BA + \beta C,$$

all three matrices are (conjugate) transposed to convert it to one of the left cases:

$$\begin{aligned} C^T &\leftarrow \alpha A^T B^T + \beta C^T && \texttt{symm}, \\ C^H &\leftarrow \bar{\alpha} A^H B^H + \bar{\beta} C^H && \texttt{hemm}. \end{aligned}$$

If A was upper, then A^T and A^H are logically lower (though still physically stored as upper) and are handled by the left-lower case; if A was lower, then A^T and A^H are logically upper and are handled by the left-upper case. Since C is now transposed, the tile BLAS `gemm` must handle a transposed C, which in relevant cases can be mapped back to regular `gemm` by re-transposing the entire `gemm` equation.

2.2.4 trmm, trsm

Similarly to `symm`, the high-level SLATE `trmm` and `trsm` each implement the two cases where the triangular matrix A is stored as lower or as upper, and is on the left,

$$\begin{aligned} B &\leftarrow \alpha AB, && \texttt{trmm}, \\ B &\leftarrow \alpha A^{-1}B, && \texttt{trsm}. \end{aligned}$$

One case uses a forward sweep, from $k = 0, \ldots, m-1$, while the other case uses a backward sweep, from $k = m-1, \ldots, 0$. As with the `symm` and `hemm`, the cases where A is on the right,

$$\begin{aligned} B &\leftarrow \alpha BA, && \texttt{trmm}, \\ B &\leftarrow \alpha BA^{-1}, && \texttt{trsm}, \end{aligned}$$

are handled by (conjugate) transposing the entire equation to reduce it to one of the left cases. As with `gemm`, A can be transposed beforehand.

By using matrix object abstraction to hide transpositions, SLATE is thus able to significantly reduce the number of cases to implement, resulting in a smaller code base to implement and maintain.

2.3 Handling of Multiple Precisions

SLATE handles multiple precisions by C++ templating, so there is only one precision-independent version of the code, which is then instantiated for the desired precisions. SLATE's BLAS++ component [14] provides overloaded, precision-independent wrappers for all the underlying BLAS routines, which SLATE's parallel BLAS are built on top of. For instance, `blas::gemm` in BLAS++ maps to the classical `sgemm`, `dgemm`, `cgemm`, or `zgemm` BLAS routines, depending on the precision of its arguments. For real, symmetric matrices, symmetric and Hermitian matrices are considered interchangeable, so `hemm` maps to `symm`, `herk` to `syrk`, and `her2k` to `syr2k`.

This mapping aides in templating higher-level routines, such as Cholesky, which does a `herk` (mapped to `syrk` in real) to update the trailing matrix.

Where a data type is always real, `blas::real_type<scalar_t>` is a C++ type trait to provide the real type associated with the type `scalar_t`, so `blas::real_type< std::complex<double> >` is `double`. This is used, for instance, in `herk`, where alpha and beta are always real.

Currently, the SLATE library has explicit instantiations of the four main data types: `float`, `double`, `std::complex<float>`, and `std::complex<double>`. The SLATE BLAS code should be able to accommodate other data types, such as quad precision, given appropriate underlying BLAS routines.

2.4 Parallelization

2.4.1 gemm and Other Routines

All PBLAS routines, except for `trsm`, are embarrassingly parallel in the sense that each tile of the output matrix can be computed independently. In SLATE, the operation is broken down into a sequence of outer products. Figure 2.1 illustrates this concept for `gemm`. All other routines follow the same principle. This is necessary in order to put an upper bound on the size of data buffers required for communication of matrices A and B. It also allows for the use of the batch `gemm` operation, which does not allow for write dependencies on the output tiles.

The SLATE implementation consists of a pipelined loop—which interleaves the steps of communication and computation—where the `lookahead` parameter defines how much the communication can get ahead of the computation. Appendix B.1 shows the pipelined loops of the `gemm` implementation. Section 3.3 contains traces illustrating the effects of lookahead.

The communication steps perform broadcasts of tiles of A and B according to the distribution of the matrix C. The computation steps perform outer product updates to the local part of C in each node. In the case of accelerated execution, the updates are executed as calls to batch `gemm` (`Target::Devices`). In the case of multi-core execution, the updates can be executed as:

- a set of OpenMP tasks (`Target::HostTask`),
- a nested `parallel for` loop (`Target::HostNest`),
- a call to batch `gemm` (`Target::HostBatch`).

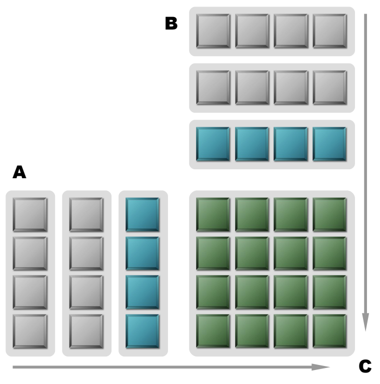


Figure 2.1: Implementation of `gemm` as a sequence of outer products.

It needs to be pointed out that many algorithms have been developed for the efficient implementations of `gemm` for distributed-memory machines, Cannon's algorithm [15] being the canonical example. Other notable examples include Parallel Universal Matrix Multiplication Algorithms (PUMMA) [16], the Scalable Universal Matrix Multiplication Algorithm (SUMMA) [17], and A Design and Implementation Methodology for Metaheuristic Algorithms (DIMMA) [18].

However, the current emphasis in SLATE is on simplicity and robustness, as well as relying on efficient implementation of MPI collective communication. More advanced algorithms are up for consideration in the future, particularly those that minimize communication [19, 20].

2.4.2 trsm

Unlike the other routines, `trsm` has data dependencies. Figure 2.2 shows the steps of `trsm` (`Side::Left`, `Uplo::Lower`, `Op::NoTrans`) with a 5×5 tiles matrix A and 5×2 tiles matrix B, and lookahead of one. The execution proceeds as follows: At each step, `trsm` is applied to a row of tiles of matrix B, followed by `gemm` for all the tiles below. The application of `gemm` is split into two parts—the first one involving lookahead rows and the second one involving all the remaining rows. Completing the first lookahead row allows for starting the next step's `trsm`.

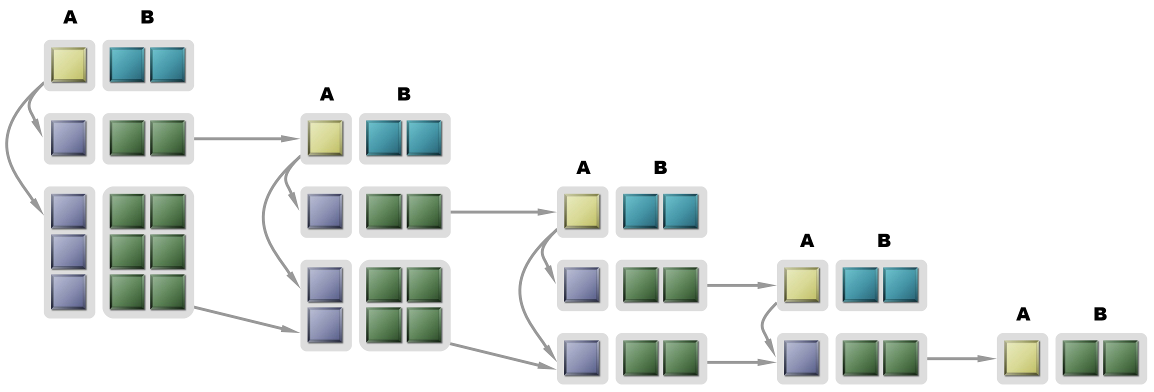


Figure 2.2: Steps of `trsm` with data dependencies (`lookahead = 1`).

Conceptually, the implementation of `trsm` is not too different from the implementation of the Cholesky factorization (`potrf`) presented in the SLATE Working Note 3 [10]. Appendix B.2 shows the `Left`, `Lower`, `NoTrans` part of the implementation. Communication is associated with the `trsm` step, and overlapping of communication relies on lookahead. Also, while the large `gemm` operation is mapped to the user-specified target, the `trsm` and the lookahead `gemm` operations are always mapped to `Target::HostTask` in the current implementation.

CHAPTER 3

Experiments

3.1 Environment

Performance numbers were collected using the SummitDev system [1] at the OLCF, which is intended to mimic the OLCF's next supercomputer, Summit. SummitDev is based on the IBM POWER8 processors and the NVIDIA P100 (Pascal) accelerators, and is one generation behind Summit, which will be based on the IBM POWER9 processors and the NVIDIA V100 (Volta) accelerators.

The SummitDev system contains three racks, each with eighteen IBM POWER8 S822LC nodes, for a total of fifty-four nodes. Each node contains two POWER8 CPUs, ten cores each, and four 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 enhanced data rate (EDR) InfiniBand.

The software environment used for the experiments included GNU Compiler Collection (GCC) 7.1.0, CUDA 9.0.69, Engineering Scientific Subroutine Library (ESSL) 5.5.0, Spectrum MPI 10.1.0.4, Netlib LAPACK 3.6.1, and Netlib ScaLAPACK 2.0.2—i.e., the output of `module list` included:

```
gcc/7.1.0
cuda/9.0.69
essl/5.5.0-20161110
spectrum-mpi/10.1.0.4-20170915
netlib-lapack/3.6.1
netlib-scalapack/2.0.2
```

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

3.2 Performance

In order to avoid excessive numbers of runs while still getting the complete picture, we asses the performance of SLATE BLAS in the following way. First, in Section 3.2.1, we look at the performance of all the routines in double precision and compare them to ScaLAPACK. Then, in Section 3.2.2, we present the performance of `trmm` in double precision for eight different cases of `side`, `uplo`, and `transa` (aka `op`). Finally, in Section 3.2.3 we show the performance of `gemm` (`NoTrans`, `NoTrans`) for different precisions (single/double, real/complex).

All runs were performed using sixteen nodes of the SummitDev system, which provides $16\ nodes \times 2\ sockets \times 10\ cores = 320$ IBM POWER8 cores and $16\ nodes \times 4\ devices = 64$ NVIDIA P100 accelerators. SLATE was run with one process per node, while ScaLAPACK was run with one process per core, which is still the prevailing method of getting the best performance from ScaLAPACK. Only rudimentary performance tuning was done in both cases.

3.2.1 PBLAS Routines in Double Precision

Figures 3.1 to 3.6 show the performance of SLATE BLAS with and without acceleration compared to the performance of ScaLAPACK without acceleration. We are not aware of a viable solution for ScaLAPACK acceleration. All runs use 16 nodes of the SummiDev system. For the accelerated runs, this translates to 64 accelerators.

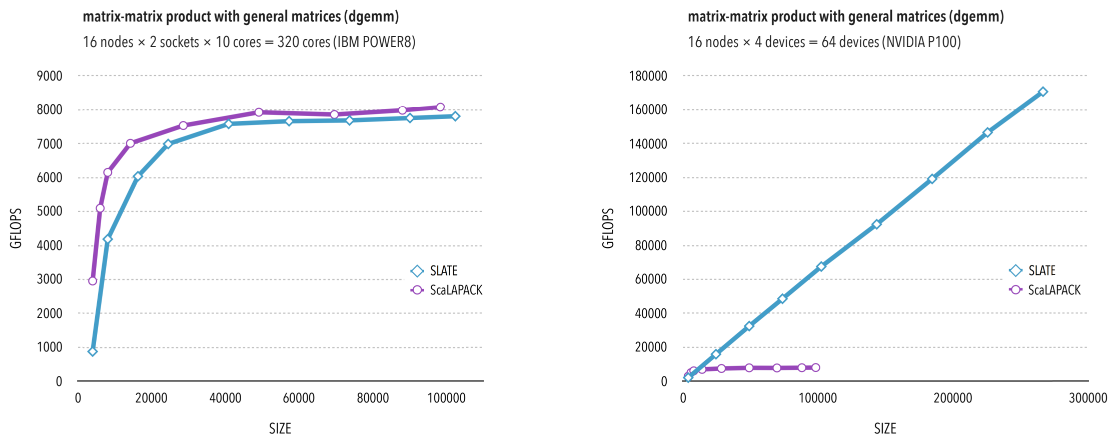
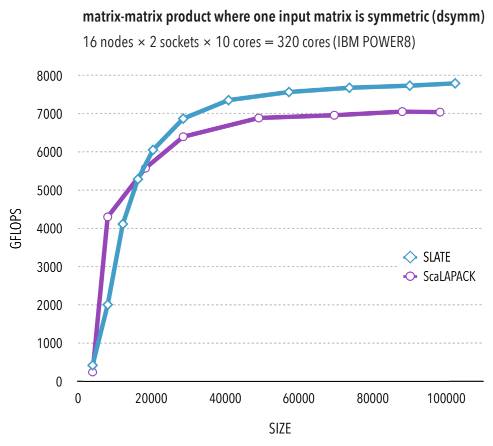


Figure 3.1: Performance of `dgemm` without acceleration (left) and with acceleration (right).

Figures 3.1 to 3.6 lead to the following observations:

- In the case of multi core runs, SLATE BLAS provides asymptotic performance very similar to ScaLAPACK. SLATE multi core runs delivered somewhat substandard performance for smaller matrix sizes, which is especially visible in the `dtrsm` sweep. This can be attributed to inadequate tuning for the optimal tile size: while fairly small blocking factors worked well for ScaLAPACK runs (80, 96, etc.), fairly large tile sizes were used for SLATE (256, 336, etc.). This should be easy to remedy in the future by more carefully tuning for the optimal tile size.



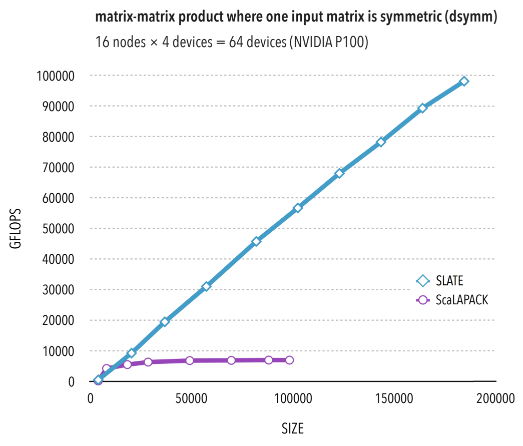
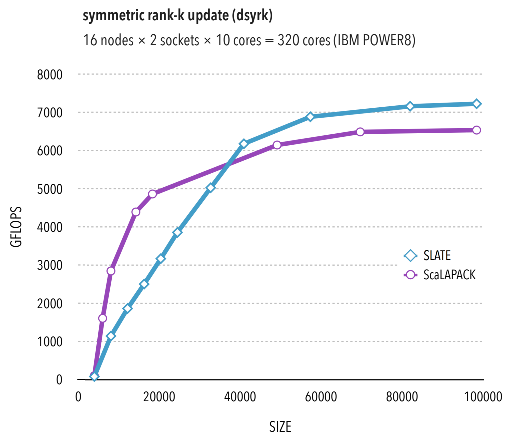


Figure 3.2: Performance of `dsymm` without acceleration (left) and with acceleration (right).



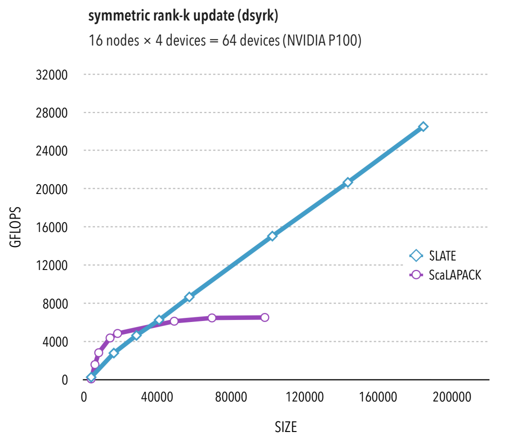


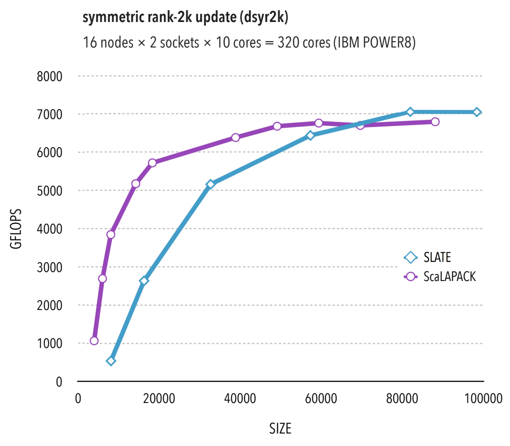
Figure 3.3: Performance of `dsyrk` without acceleration (left) and with acceleration (right).

- SLATE delivers massive performance gains through acceleration. The highest performance number was reached for the accelerated sweep of `dgemm` (Figure 3.1), which reached 170 teraFLOP/s, compared to the top ScaLAPACK `dgemm` performance of 7.8 teraFLOP/s (over 20× difference). This is of no surprise, as the peak performance of SummitDev's multi cores is at the level of 2.5% of SummitDev's accelerators.

- The `syrk` and `syr2k` routines require further attention, as their top accelerated performance is significantly lower than the accelerated performance of other routines.

- All accelerated performance curves are basically linear. The reason for this behavior is very clear, given that the PBLAS performance profile follows the Roofline model [2] [21]. It is clear that the performance is nowhere near saturation. For the tested matrix sizes, the performance is completely bound by communication. This is also of no surprise given the ratio of SummitDev's node performance to its communication bandwidth. This is

[2]https://en.wikipedia.org/wiki/Roofline_model



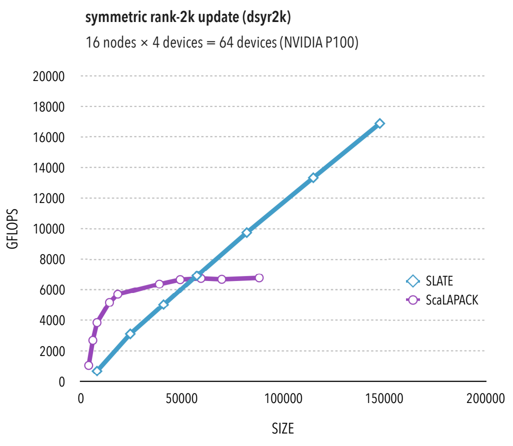
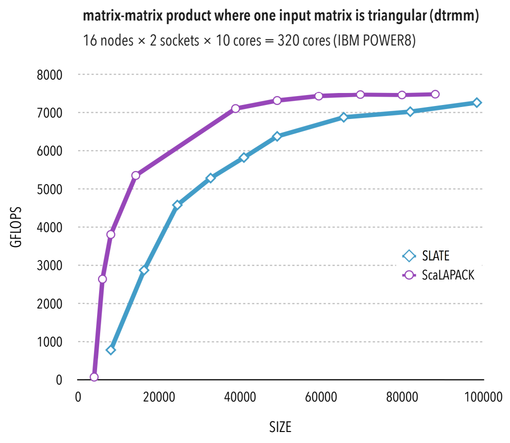


Figure 3.4: Performance of `dsyr2k` without acceleration (left) and with acceleration (right).



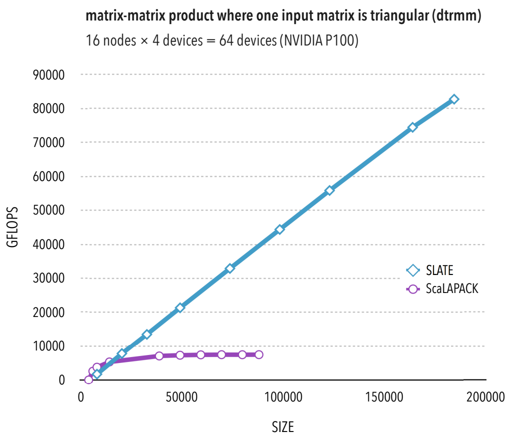
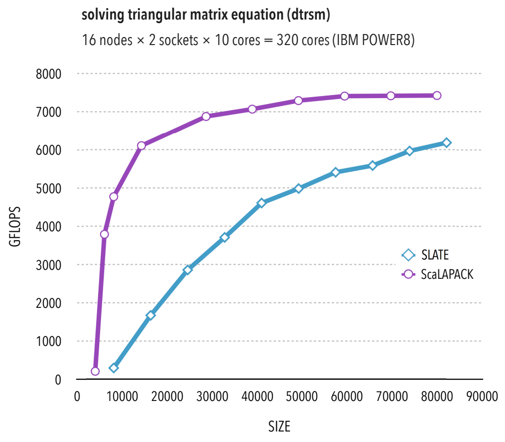


Figure 3.5: Performance of `dtrmm` without acceleration (left) and with acceleration (right).

further confirmed by the runs in complex arithmetic, presented in Section 3.2.3, as well as the traces of accelerated execution, presented in Appendix 3.3.



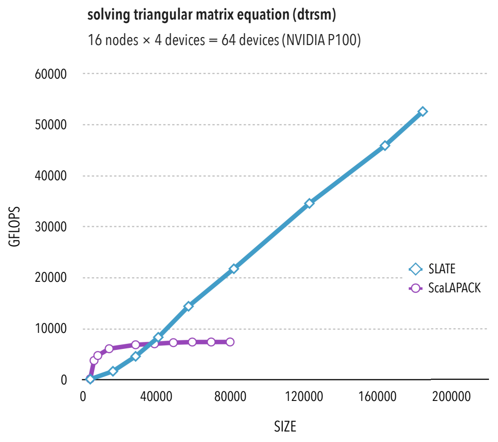
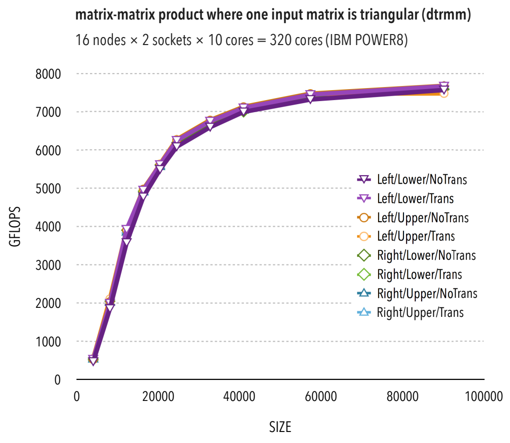


Figure 3.6: Performance of `dtrsm` without acceleration (left) and with acceleration (right).

3.2.2 Different Cases of DTRMM

Figures 3.7 and 3.8 show the performance of the SLATE `dtrmm` routine for different cases of `side`, `uplo`, and `transa`. Figure 3.7 shows multi-core performance and Figure 3.8 shows accelerated performance. Just as before, all runs use 16 nodes of the SummiDev system.



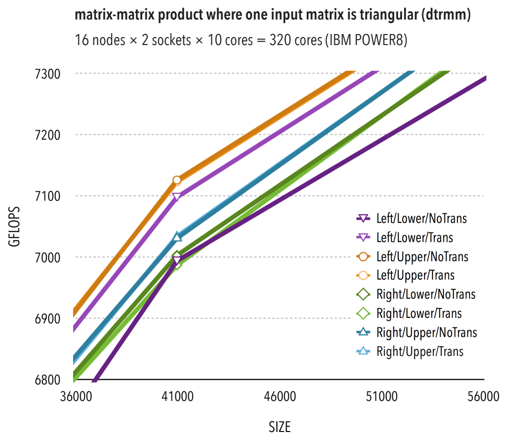
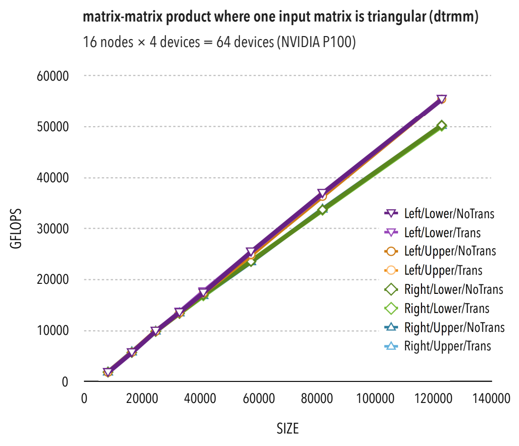


Figure 3.7: Multi-core performance of `dtrmm` for different cases of `side`, `uplo`, `transa` (complete sweep on the left, closeup on the right).

The expectation is that the choice of `side`, `uplo`, and `transa` parameters has very little impact on performance. As a matter of fact, this turns out to be the case. In the case of multi core runs (Figure 3.7), there is virtually no difference in performance. In the chart on the left, all curves basically overlap. The large magnification on the right shows all eight curves in the band.

Likewise, parameter choice for the accelerated runs has little impact on performance (Figure 3.8). Here all lines are contained within two bands, and the magnification on the right shows that the difference comes from the choice of the `side` parameter. The difference in performance between `Side::Left` and `Side::Right` is about 10%, which is no cause for concern.



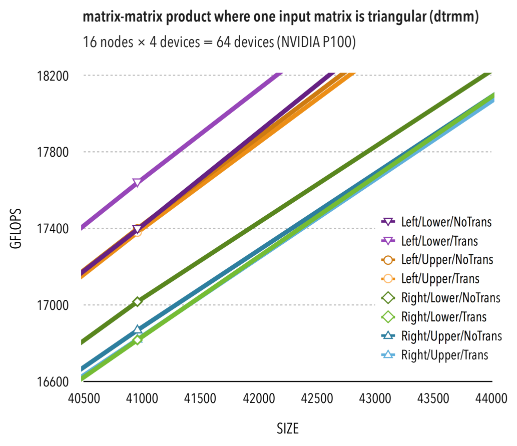
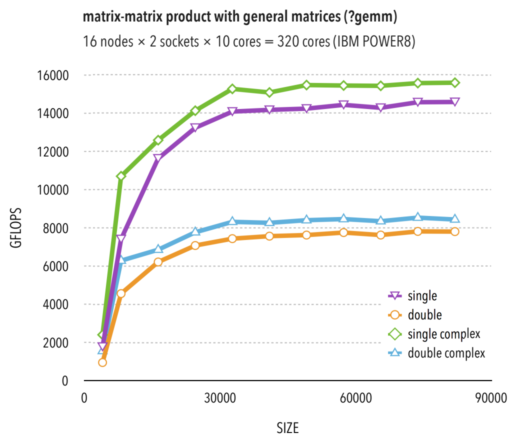


Figure 3.8: Accelerated performance of `dtrmm` for different cases of `side`, `uplo`, `transa` (complete sweep on the left, closeup on the right).

3.2.3 GEMM in Different Precisions

The last part of the performance assessment is the impact of different precisions (single/double, real/complex) on performance. Figure 3.9 shows the performance of the `gemm` routine in the four standard precisions (S, C, D, Z) for multi core runs (left chart) and accelerated runs (right chart).



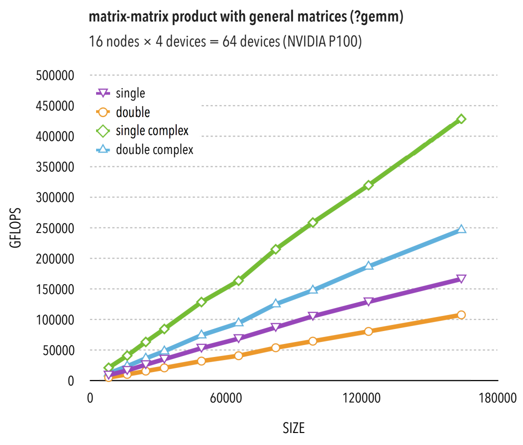


Figure 3.9: Performance of `gemm` for different precisions (single/double, real/complex).

Here the performance numbers are, again, far from unexpected. In the case of multi core runs, single precision is about twice as fast as double precision, which comes directly from the $2\times$ factor between the single-precision peak and the double-precision peak of the POWER8 cores. At the same time, complex arithmetic is slightly faster than real arithmetic. This is because the complex arithmetic is twice as compute-intensive as real arithmetics, which bring the performance number a notch closer to the hardware peak.

For accelerated runs, shown in the chart on the right, the situation is very different. Here, the performance of single precision is twice the performance of double precision; but also, the performance of complex arithmetic is twice the performance of real arithmetic. This is because here the performance has much less to do with the floating-point peaks and much more to do with the communication bandwidth. As already mentioned in Section 3.2.1, accelerated performance is nowhere near the floating-point peak. Instead, it is completely bound by the communication bandwidth. As a result, the accelerated performance is directly correlated with the ratio of computation to communication, which is $2\times$ higher in single precision than in double precision, and $2\times$ higher in complex arithmetic than in real arithmetic.

3.3 Traces

The traces presented in this section were produced by a small tracing component embedded in SLATE. Tasks' start and end times are collected using `omp_get_wtime()`, and printed at the end of the run to a Scalable Vector Graphic (SVG) file. This is a simplistic yet effective approach inherited from the PLASMA and MAGMA projects [22], and the PULSAR project [23].

Figures 3.10, 3.11, and 3.12 show traces of multi core `dgemm` runs for $m = n = k = 5120$ and $nb = 256$, using eighty IBM POWER8 cores (four nodes of SummitDev). The traces show execution with lookahead of zero, one and two. The point is to show that lookahead allows communication and computation to overlap completely. While changing the value from zero to one provides the biggest improvement, increasing the value to two provides a further small improvement.

Figure 3.13 shows a trace of an accelerated `dgemm` run for $m = n = k = 102400$, $nb = 1024$, and $lookahead = 1$, using sixteen NVIDIA P100 accelerators (four nodes of SummitDev). Figure 3.14 is a closeup showing execution of three steps on a single node, and Figure 3.15 is a closeup showing communication involved in the broadcast of six tiles. The trace shows that, while scheduling of tasks to accelerators and overlapping of communication and computation works as expected, performance is severely handicapped by the inability of the communication subsystem to keep up with the rate of the accelerators' floating-point execution.

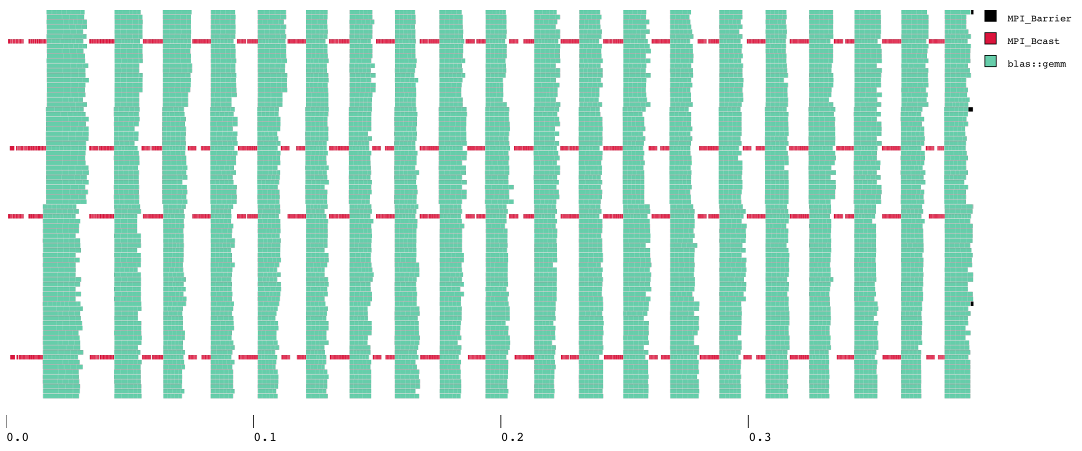


Figure 3.10: Multi core trace of dgemm with $m = n = k = 5120$, $nb = 256$, and $lookahead = 0$, using 4 nodes × 2 sockets × 10 cores = 80 cores (IBM POWER8).

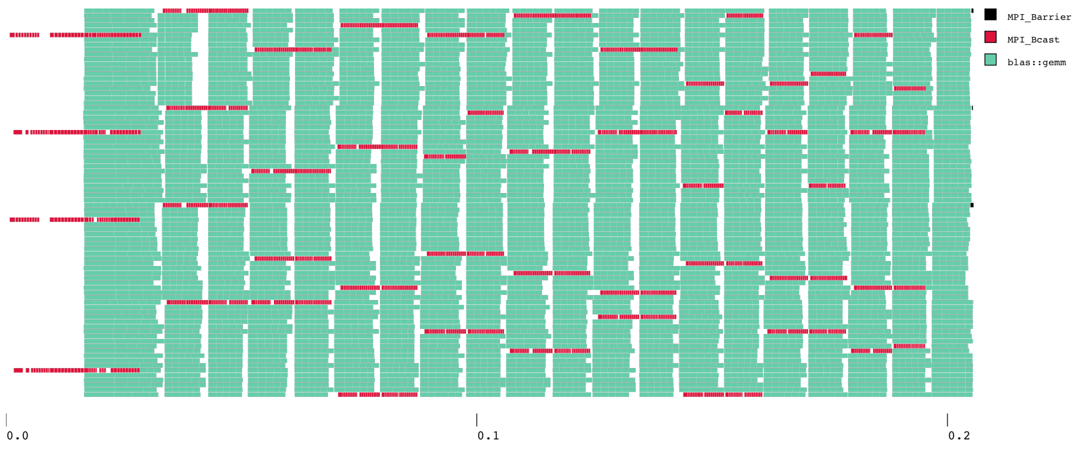


Figure 3.11: Multi core trace of dgemm with $m = n = k = 5120$, $nb = 256$, and $lookahead = 1$, using 4 nodes × 2 sockets × 10 cores = 80 cores (IBM POWER8).

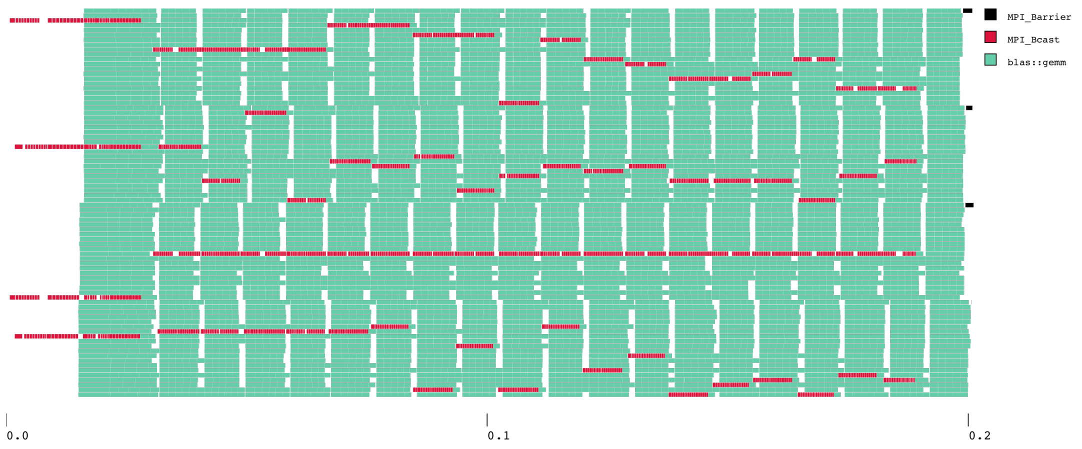


Figure 3.12: Multicore trace of dgemm with $m = n = k = 5120$, $nb = 256$, and $lookahead = 2$, using 4 nodes × 2 sockets × 10 cores = 80 cores (IBM POWER8).

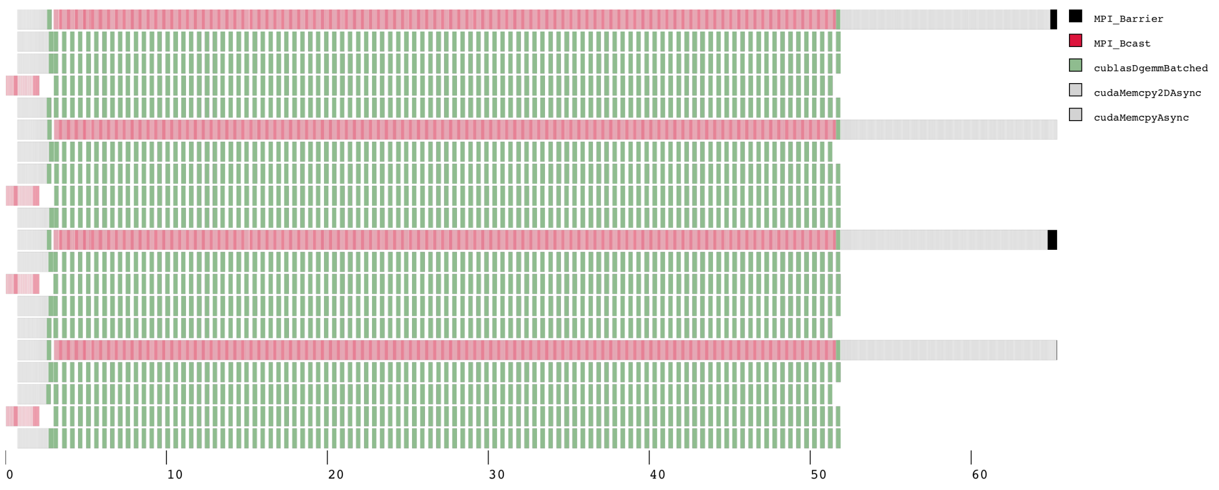


Figure 3.13: Accelerated trace of `dgemm` with $m = n = k = 102400$, $nb = 1024$, and $lookahead = 1$, using 4 nodes $\times$ 4 devices = 16 devices (NVIDIA P100).

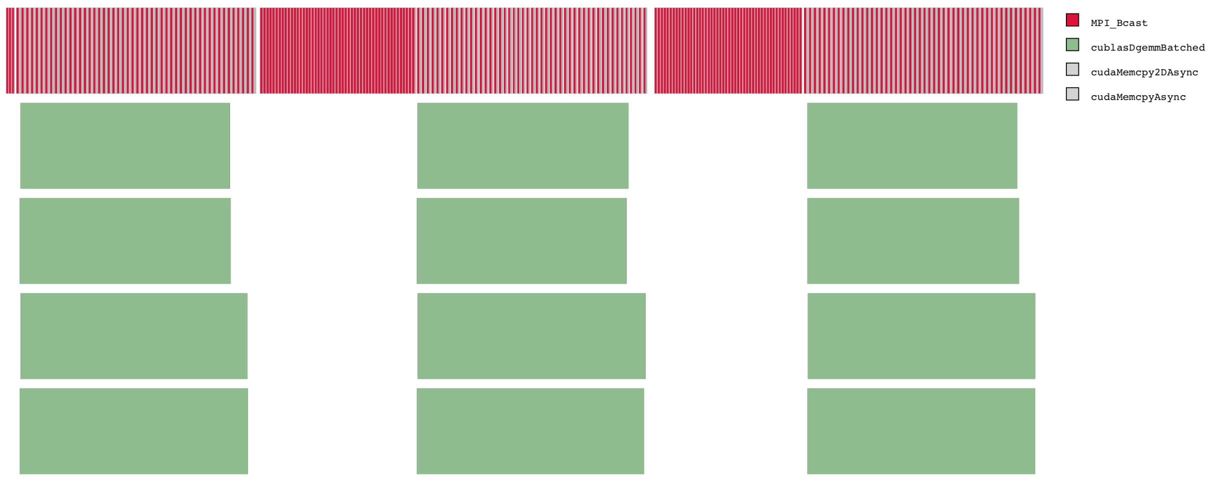


Figure 3.14: Accelerated trace of dgemm — closeup (single node, 3 steps).
Accelerated trace of `dgemm` — closeup (single node, 3 steps).

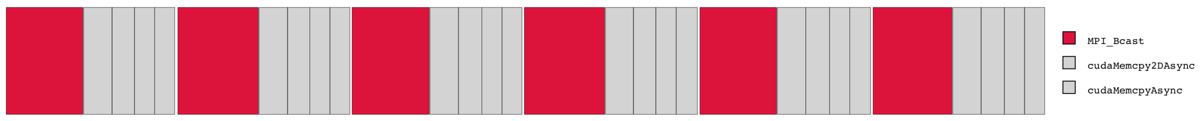


Figure 3.15: Accelerated trace of dgemm — closeup (broadcast of 6 tiles).
Accelerated trace of `dgemm` — closeup (broadcast of 6 tiles).

CHAPTER 4

Summary

The lessons learned in the process of developing the SLATE PBLAS routines and the results of the performance experiments led to the following conclusions:

- Moving from C and Fortran to C++ and designing SLATE from the ground up provides significant software engineering improvements over ScaLAPACK. Consider that:
  - Using C++ templates for handling multiple precisions allows for a single SLATE routine to replace four ScaLAPACK routines, e.g., `slate::gemm` replaces ScaLAPACK's `sgemm`, `dgemm`, `cgemm`, and `zgemm`.
  - Treating transposition as a matrix property allows for one implementation to address multiple cases of the input parameters `side`, `uplo`, and `trans`/`op`. In the case of, for example, the `trmm` routine, we can replace eight blocks of code with only two blocks of code.
  - One implementation provides: multi-core capabilities, acceleration capabilities, and distributed-memory capabilities. Therefore, SLATE is a natural fit for a multi-core laptop, a desktop with one or more accelerators, or supercomputing systems such as SummitDev or Summit.
- In terms of multi-core execution, SLATE provides capabilities similar to ScaLAPACK's. In most cases, equal or better asymptotic performance is reached. The performance disadvantage for smaller matrix sizes can most likely be resolved by more careful tuning.
- In terms of accelerated performance, SLATE provides unique capabilities. In short, we are not aware of a viable alternative. SLATE's accelerated performance is an order of magnitude higher than multi-core performance for most of the routines. At the same

time, some of the routines (`syrk`, `syr2k`) require further attention, as their performance is not yet on a par with the others.

Bibliography

[1] Jaeyoung Choi, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, David Walker, and R Clinton Whaley. A proposal for a set of parallel basic linear algebra subprograms. In *International Workshop on Applied Parallel Computing*, pages 107–114. Springer, 1995.

[2] J Choiy, J Dongarraz, S Ostrouchovx, A Petitetx, D Walker, and RC Whaleyx. Lapack working note 100 a proposal for a set of parallel basic linear algebra subprograms. *University of Tennessee, Knoxville*, 1995.

[3] Jack J Dongarra, Jermey Du Cruz, Sven Hammarling, and Iain S Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):18–28, 1990.

[4] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. *ACM Transactions on Mathematical Software (TOMS)*, 14(1):18–32, 1988.

[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] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 120–127. IEEE, 1992.

[7] 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.

[8] Edward Anderson, Zhaojun Bai, Jack Dongarra, Anne Greenbaum, Alan McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, C Bischof, and Danny Sorensen. Lapack: A

portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11. IEEE Computer Society Press, 1990.

[9] Ed Anderson, A Benzoni, J Dongarra, S Moulton, S Ostrouchov, Bernard Tourancheau, and Robert van de Geijn. Basic linear algebra commnunication subprograms. In *Distributed Memory Computing Conference, 1991. Proceedings., The Sixth*, pages 287–290. IEEE, 1991.

[10] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. SLATE working note 3: Designing SLATE: Software for linear algebra targeting exascale. Technical Report ICL-UT-17-06, Innovative Computing Laboratory, University of Tennessee, September 2017. revision 09-2017.

[11] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28 (2):135–151, 2002.

[12] 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.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[14] Mark Gates, Piotr Luszczek, Ahmad Abdelfattah, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. SLATE working note 2: C++ API for BLAS and LAPACK. Technical Report ICL-UT-17-03, Innovative Computing Laboratory, University of Tennessee, June 2017. revision 03-2018.

[15] Lynn E Cannon. A cellular computer to implement the Kalman filter algorithm. Technical report, MONTANA STATE UNIV BOZEMAN ENGINEERING RESEARCH LABS, 1969.

[16] Jaeyoung Choi, David W Walker, and Jack J Dongarra. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency and Computation: Practice and Experience*, 6(7):543–570, 1994.

[17] Robert A Van De Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency-Practice and Experience*, 9(4):255–274, 1997.

[18] Jaeyoung Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. In *High Performance Computing on the Information Superhighway, 1997. HPC Asia'97*, pages 224–229. IEEE, 1997.

[19] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[20] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[21] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[22] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[23] 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.

Appendices

APPENDIX A

Function Signatures

```
template <Target target=Target::HostTask, typename scalar_t>
void gemm(scalar_t alpha, Matrix<scalar_t>& A,
                          Matrix<scalar_t>& B,
          scalar_t beta,  Matrix<scalar_t>& C,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void hemm(blas::Side side,
          scalar_t alpha, HermitianMatrix<scalar_t>& A,
                          Matrix<scalar_t>& B,
          scalar_t beta,  Matrix<scalar_t>& C,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void herk(blas::real_type<scalar_t> alpha, Matrix<scalar_t>& A,
          blas::real_type<scalar_t> beta,  HermitianMatrix<scalar_t>& C,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void her2k(scalar_t alpha,                  Matrix<scalar_t>& A,
                                            Matrix<scalar_t>& B,
           blas::real_type<scalar_t> beta, HermitianMatrix<scalar_t>& C,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void symm(blas::Side side,
          scalar_t alpha, SymmetricMatrix<scalar_t>& A,
                          Matrix<scalar_t>& B,
          scalar_t beta,  Matrix<scalar_t>& C,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void syrk(scalar_t alpha, Matrix<scalar_t>& A,
          scalar_t beta,  SymmetricMatrix<scalar_t>& C,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void syr2k(scalar_t alpha, Matrix<scalar_t>& A,
                           Matrix<scalar_t>& B,
           scalar_t beta,  SymmetricMatrix<scalar_t>& C,
           const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void trmm(blas::Side side, blas::Diag diag,
          scalar_t alpha, TriangularMatrix<scalar_t>& A,
                          Matrix<scalar_t>& B,
          const std::map<Option, Value>& opts = std::map<Option, Value>());

template <Target target=Target::HostTask, typename scalar_t>
void trsm(blas::Side side, blas::Diag diag,
          scalar_t alpha, TriangularMatrix<scalar_t>& A,
                          Matrix<scalar_t>& B,
          const std::map<Option, Value>& opts = std::map<Option, Value>());
```

APPENDIX B

Implementation Snippets

B.1 gemm

```
#pragma omp parallel
#pragma omp master
{
    #pragma omp task depend(out:bcast[0])
    {
        for (int64_t i = 0; i < A.mt(); ++i)
            A.template tileBcast<target>(
                i, 0, C.sub(i, i, 0, C.nt()-1));

        for (int64_t j = 0; j < B.nt(); ++j)
            B.template tileBcast<target>(
                0, j, C.sub(0, C.mt()-1, j, j));
    }

    for (int64_t k = 1; k < lookahead+1 && k < A.nt(); ++k)
        #pragma omp task depend(in:bcast[k-1]) \
                         depend(out:bcast[k])
        {
            for (int64_t i = 0; i < A.mt(); ++i)
                A.template tileBcast<target>(
                    i, k, C.sub(i, i, 0, C.nt()-1));

            for (int64_t j = 0; j < B.nt(); ++j)
                B.template tileBcast<target>(
                    k, j, C.sub(0, C.mt()-1, j, j));
        }

    #pragma omp task depend(in:bcast[0]) \
                     depend(out:gemm[0])
    internal::gemm<target>(
        alpha, A.sub(0, A.mt()-1, 0, 0),
               B.sub(0, 0, 0, B.nt()-1),
        beta,  C.sub(0, C.mt()-1, 0, C.nt()-1));

    for (int64_t k = 1; k < A.nt(); ++k) {
        if (k+lookahead < A.nt())
            #pragma omp task depend(in:gemm[k-1]) \
                             depend(in:bcast[k+lookahead-1]) \
                             depend(out:bcast[k+lookahead])
            {
                for (int64_t i = 0; i < A.mt(); ++i)
                    A.template tileBcast<target>(
                        i, k+lookahead, C.sub(i, i, 0, C.nt()-1));

                for (int64_t j = 0; j < B.nt(); ++j)
                    B.template tileBcast<target>(
                        k+lookahead, j, C.sub(0, C.mt()-1, j, j));
            }

        #pragma omp task depend(in:bcast[k]) \
                         depend(in:gemm[k-1]) \
                         depend(out:gemm[k])
        internal::gemm<target>(
            alpha,         A.sub(0, A.mt()-1, k, k),
                           B.sub(k, k, 0, B.nt()-1),
            scalar_t(1.0), C.sub(0, C.mt()-1, 0, C.nt()-1));
    }
}
```

B.2 trsm

```
    #pragma omp parallel
    #pragma omp master
    {
        if ((A.uplo() == Uplo::Lower && A.op() == Op::NoTrans) ||
            (A.uplo() == Uplo::Upper && A.op() != Op::NoTrans)) {
            // ----------------------------------------
            // Lower/NoTrans or Upper/Trans, Left case
            // Forward sweep
            for (int64_t k = 0; k < mt; ++k) {
                scalar_t alph = k == 0 ? alpha : scalar_t(1.0);

                // panel (Akk tile)
                #pragma omp task depend(inout:row[k]) priority(1)
                {
                    // send A(k, k) to ranks owning block row B(k, :)
                    A.template tileBcast<target>(
                        k, k, B.sub(k, k, 0, nt-1));

                    // solve A(k, k) B(k, :) = alpha B(k, :)
                    internal::trsm<Target::HostTask>(
                        Side::Left, diag,
                        alph, A.sub(k, k),
                              B.sub(k, k, 0, nt-1), 1);

                    // send A(i=k+1:mt-1, k) to ranks owning block row B(i, :)
                    for (int64_t i = k+1; i < mt; ++i)
                        A.template tileBcast(
                            i, k, B.sub(i, i, 0, nt-1));

                    // send B(k, j=0:nt-1) to ranks owning block col B(k+1:mt-1, j)
                    for (int64_t j = 0; j < nt; ++j)
                        B.template tileBcast(
                            k, j, B.sub(k+1, mt-1, j, j));
                }

                // lookahead update, B(k+1:k+la, :) -= A(k+1:k+la, k) B(k, :)
                for (int64_t i = k+1; i < k+1+lookahead && i < mt; ++i) {
                    #pragma omp task depend(in:row[k]) \
                                     depend(inout:row[i]) priority(1)
                    {
                        internal::gemm<Target::HostTask>(
                            scalar_t(-1.0), A.sub(i, i, k, k),
                                            B.sub(k, k, 0, nt-1),
                            alph,           B.sub(i, i, 0, nt-1), 1);
                    }
                }

                // trailing update, B(k+1+la:mt-1, :) -= A(k+1+la:mt-1, k) B(k, :)
                // Updates rows k+1+la to mt-1, but two depends are sufficient:
                // depend on k+1+la is all that is needed in next iteration;
                // depend on mt-1 daisy chains all the trailing updates.
                if (k+1+lookahead < mt) {
                    #pragma omp task depend(in:row[k]) \
                                     depend(inout:row[k+1+lookahead]) \
                                     depend(inout:row[mt-1])
                    {
                        internal::gemm<target>(
                            scalar_t(-1.0), A.sub(k+1+lookahead, mt-1, k, k),
                                            B.sub(k, k, 0, nt-1),
                            alph,           B.sub(k+1+lookahead, mt-1, 0, nt-1));
                    }
                }
            }
        }
```