Parallel BLAS Performance Report

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

Innovative Computing Laboratory

April 1, 2018
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.

<table>
<thead>
<tr>
<th>Revision</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>03-2018</td>
<td>first publication</td>
</tr>
</tbody>
</table>

@techreport{dongarra2018parallel,  
  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-XX},  
  note={revision 03-2018}  
}
Contents

List of Figures iv
List of Tables v

1 Introduction 1

2 Implementation 4
  2.1 Matrix Class Hierarchy ........................................ 5
  2.2 Handling of side, uplo, trans ................................ 6
    2.2.1 gemm .................................................... 6
    2.2.2 syrk, syr2k, herk, her2k .............................. 6
    2.2.3 symm, hemm ............................................ 6
    2.2.4 trmm, trsm ............................................. 7
  2.3 Handling of Multiple Precisions ............................. 7
  2.4 Parallelization ................................................ 8
    2.4.1 gemm and Other Routines ............................... 8
    2.4.2 trsm .................................................... 9

3 Experiments 10
  3.1 Environment .................................................. 10
  3.2 Performance .................................................. 11
    3.2.1 PBLAS Routines in Double Precision .................. 11
    3.2.2 Different Cases of DTRMM .............................. 14
    3.2.3 GEMM in Different Precisions .......................... 15
  3.3 Traces ....................................................... 16

4 Summary 19

Bibliography 20
List of Figures

2.1 Implementation of gemm as a sequence of outer products. 8
2.2 Steps of trsm with data dependencies (lookahead = 1). 9
3.1 Performance of dgemm. 11
3.2 Performance of dsymm. 12
3.3 Performance of dsyrk. 12
3.4 Performance of dsyr2k. 13
3.5 Performance of dtrmm. 13
3.6 Performance of dtrsm. 14
3.7 Multicore performance of dtrmm for different cases of side, uplo, transa. 14
3.8 Accelerated performance of dtrmm for different cases of side, uplo, transa. 15
3.9 Performance of gemm for different precisions (single/double, real/complex). 15
3.10 Multicore trace of dgemm with lookahead=0. 17
3.11 Multicore trace of dgemm with lookahead=1. 17
3.12 Multicore trace of dgemm with lookahead=2. 17
3.13 Accelerated trace of dgemm. 18
3.14 Accelerated trace of dgemm - closeup (single node, 3 steps). 18
3.15 Accelerated trace of dgemm - closeup (broadcast of 6 tiles). 18
List of Tables

1.1  Descriptions of level 3 (P)BLAS routines.
1.2  Operations performed by the level 3 (P)BLAS routines.
1.3  Number of floating point operations performed by the level 3 (P)BLAS routines.
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. It 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 of two United States 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 first batch of computational routines developed in the SLATE project, 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
CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>Computes a matrix-matrix product with general matrices.</td>
</tr>
<tr>
<td>symm</td>
<td>Computes a matrix-matrix product where one input matrix is symmetric.</td>
</tr>
<tr>
<td>hemm</td>
<td>Computes a matrix-matrix product where one input matrix is Hermitian.</td>
</tr>
<tr>
<td>syrk</td>
<td>Performs a symmetric rank-k update.</td>
</tr>
<tr>
<td>herk</td>
<td>Performs a Hermitian rank-k update.</td>
</tr>
<tr>
<td>syr2k</td>
<td>Performs a symmetric rank-2k update.</td>
</tr>
<tr>
<td>her2k</td>
<td>Performs a Hermitian rank-2k update.</td>
</tr>
<tr>
<td>trmm</td>
<td>Computes a matrix-matrix product where one input matrix is triangular.</td>
</tr>
<tr>
<td>trsm</td>
<td>Solves a triangular matrix equation.</td>
</tr>
</tbody>
</table>

Table 1.1: Descriptions of level 3 (P)BLAS routines.

Table 1.1 lists the level 3 (P)BLAS routines and provides their descriptions. Table 1.2 contains their mathematical definitions. Table 1.3 provides the number of floating point operation for each routine. The numbers are for real arithmetic. In complex arithmetic, the number of floating point operation is 4 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 the BLAS is the “Basic Linear Algebra Technical Forum (BLAST) 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 (S, D, C, Z). Instead, SLATE relies on C++ templates and overloading, as described in Section 3.2.3. Appendix A provides function signatures of all SLATE (P)BLAS routines.

Another notable difference is the lack of the uplo and trans parameters in the function signatures. The information whether the matrix is upper or lower is a property of the Matrix object. So is the information if 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
CHAPTER 1. INTRODUCTION

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

Table 1.2: Operations performed by the level 3 (P)BLAS routines. $\text{op}(X) = X, X^T$, or $X^H$.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of Floating Point Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>$2mnk$</td>
</tr>
<tr>
<td>symm</td>
<td>$2m^2n$ (side=Left), $2mn^2$ (side=Right)</td>
</tr>
<tr>
<td>syrk</td>
<td>$kn(n+1)$</td>
</tr>
<tr>
<td>syr2k</td>
<td>$2kn^2 + n$</td>
</tr>
<tr>
<td>trmm</td>
<td>$nn^2$ (side=Left), $mn^2$ (side=Right)</td>
</tr>
<tr>
<td>trsm</td>
<td>$nn^2$ (side=Left), $nn^2$ (side=Right)</td>
</tr>
</tbody>
</table>

Table 1.3: Number of floating point operations performed by the level 3 (P)BLAS routines.
CHAPTER 2

Implementation

The principles of the SLATE software framework were laid out in SLATE Working Note 3 \(^1\) [10]. SLATE 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 OpenMP tasking, with the top level responsible for resolving data dependencies and the bottom level responsible for deploying large numbers of independent tasks to multicore processors and accelerator devices.

- 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 \texttt{gemm}.

Also, the use of a runtime scheduling system, such as 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
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 \((A_{j,i} = \overline{A_{i,j}})\).

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 a possibly transposed copy of an original matrix \(A_0\), 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 \(A = \text{transpose}(A_0)\) and \(A = \text{conj\_transpose}(A_0)\) 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 9 cases in zgemm and 8 cases in ztrmm (times several sub-cases). ScalAPACK and PLASMA likewise have 8 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:

\[
\text{slate::gemm( alpha, transpose(A), conj_transpose(B), beta, C );}
\]

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:

\[ C \leftarrow \alpha AA^T + \beta C \quad \text{syrk,} \]
\[ C \leftarrow \alpha AA^H + \beta C \quad \text{herk,} \]
\[ C \leftarrow \alpha AB^T + \alpha BA^T + \beta C \quad \text{syrk,} \]
\[ C \leftarrow \alpha AB^H + \alpha BA^H + \beta C \quad \text{herk,} \]

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:

$$C^T \leftarrow \alpha A^T B^T + \beta C^T \quad \text{symm},$$

$$C^H \leftarrow \bar{\alpha} A^H B^H + \bar{\beta} C^H \quad \text{hemm}.$$

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, and 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 the 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,

$$B \leftarrow \alpha AB, \quad \text{trmm},$$

$$B \leftarrow \alpha A^{-1} B, \quad \text{trsm}.$$

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 when $A$ is on the right,

$$B \leftarrow \alpha BA, \quad \text{trmm},$$

$$B \leftarrow \alpha BA^{-1}, \quad \text{trsm},$$

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 the 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 (P)BLAS 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 that 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 multicore 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).

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 PUMMA [16], SUMMA [17], and DIMMA [18]. At the same time, the current emphasis in SLATE is on simplicity and robustness, and relying on efficient implementation of MPI collective communication. More advanced algorithms are
up for consideration in the future, with emphasis on algorithms 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 $5 \times 5$ tiles matrix $A$ and $5 \times 2$ 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 trsm of the next step.

![Figure 2.2: Steps of trsm with data dependencies (lookahead = 1).](image)

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 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 9.0.69, 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:

\[
gcc/7.1.0
\]
\[
cuda/9.0.69
\]
\[
sssl/5.5.0-20161110
\]
\[
spectrum-mpi/10.1.0.4-20170915
\]
\[
netlib-lapack/3.6.1
\]
\[
netlib-scalapack/2.0.2
\]

\(^1\)https://www.olcf.ornl.gov/kb_articles/summitdev-quickstart/
3.2 Performance

In order to avoid excessive number of runs, yet get the complete picture, we assess 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 against ScaLAPACK. Then, in Section 3.2.2, we present the performance of \texttt{trmm} in double precision for eight different cases of \texttt{side}, \texttt{uplo}, and \texttt{transa} (aka \texttt{op}). Finally, in Section 3.2.3 we show the performance of \texttt{gemm} (\texttt{NoTrans, NoTrans}) for different precisions (single/double, real/complex).

All runs were performed using 16 nodes of the SummitDev system, which provides \(16 \times 2 \times 10 = 320\) IBM POWER8 cores and \(16 \times 4 = 64\) 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.

Figures 3.1 to 3.6 lead to the following observations:

- In the case of multicore runs SLATE BLAS provides asymptotic performance very similar to ScaLAPACK. SLATE multicore runs delivered somewhat substandard performance for smaller matrix sizes, which is especially visible in the \texttt{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, 386, etc.). This should be easy to remedy in the future by more careful tuning for the optimal tile size.
• SLATE delivers massive performance gains through acceleration. The highest performance number was reached for the accelerated sweep of \texttt{dgemm} (Figure 3.1), which reached 170 teraFLOPS, compared to the top ScaLAPACK \texttt{dgemm} performance of 7.8 teraFLOPS (over $20 \times$ difference). This is of no surprise, as the peak performance of SummitDev’s multicores is at the level of 2.5% of SummitDev’s accelerators.

• The \texttt{dsyrk} and \texttt{dsyr2k} 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 \footnote{https://en.wikipedia.org/wiki/Roofline_model} [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
Figure 3.4: Performance of \texttt{dsyr2k} without acceleration (left) and with acceleration (right).

Figure 3.5: Performance of \texttt{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.
3.2. PERFORMANCE

3.2.2 Different Cases of DTRMM

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

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 multicore 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 the eight curves in the band.

Similar is true for the accelerated runs (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 reason for concern.

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

Figure 3.7: Multicore performance of dtrmm for different cases of side, uplo, transa. (complete sweep on the left, closeup on the right).
3.2. PERFORMANCE

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 multicore runs (left chart) and accelerated runs (right chart).

Here the performance numbers are, again, far from unexpected. In the case of multicore 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.

The situation is very different in the case of accelerated runs shown in the chart on the right.
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 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 multicore dgemm runs for $m = n = k = 5120$ and $nb = 256$, using 80 IBM POWER8 cores (4 nodes of SummitDev). The traces show execution with lookahead of zero, one and two. The point is to show that lookahead allows for complete overlapping of communication with computation. 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 $\text{lookahead} = 1$, using 16 NVIDIA P100 accelerators (4 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 floating point execution of the accelerators.
3.3. TRACES

CHAPTER 3. EXPERIMENTS

Figure 3.10: Multicore trace of \texttt{dgemm} with $m = n = k = 5120$, $nb = 256$, and $\text{lookahead} = 0$, using 4 nodes $\times$ 2 sockets $\times$ 10 cores $= 80$ cores (IBM POWER8).

Figure 3.11: Multicore trace of \texttt{dgemm} with $m = n = k = 5120$, $nb = 256$, and $\text{lookahead} = 1$, using 4 nodes $\times$ 2 sockets $\times$ 10 cores $= 80$ cores (IBM POWER8).

Figure 3.12: Multicore trace of \texttt{dgemm} with $m = n = k = 5120$, $nb = 256$, and $\text{lookahead} = 2$, using 4 nodes $\times$ 2 sockets $\times$ 10 cores $= 80$ cores (IBM POWER8).
3.3. TRACES

CHAPTER 3. EXPERIMENTS

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

Figure 3.14: Accelerated trace of \texttt{dgemm} - closeup (single node, 3 steps).

Figure 3.15: Accelerated trace of \texttt{dgemm} - closeup (broadcast of 6 tiles).

Accelerated trace of \texttt{dgemm} - closeup (single node, 3 steps).

Accelerated trace of \texttt{dgemm} - closeup (broadcast of 6 tiles).
Summary

The lessons learned in the process of developing the SLATE (P)BLAS routines and the results of the performance experiments lead 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 input parameters `side`, `uplo`, and `trans`/`op`. In the case of the, e.g., `trmm` routine, this replaces eight blocks of code with only two blocks of code.
  – One implementation provides: multicore capabilities, acceleration capabilities, and distributed memory capabilities. I.e., SLATE is a natural fit for a multicore laptop, an desktop with one or more accelerators, or a supercomputing systems such as SummitDev or Summit.

• In terms of multicore execution, SLATE provides similar capabilities to ScaLAPACK. 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. I.e., we are not aware of a viable alternative. SLATE accelerated performance is an order of magnitude higher than multicore 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.


20


Appendices
APPENDIX A

Function Signatures
APPENDIX A. FUNCTION SIGNATURES

```cpp
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

```cpp
#define omp #pragma omp
#define __omp #pragma omp
#define task depend(out: bcast[0])
#define master depend(out: bcast[0])
#define parallel depend(out: bcast[0])
#define gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(out: gemm[0]) #pragma omp task depend(out: gemm[0])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
#define parallel internal::gmem template tileBcast(target>(
#define parallel depend(in: bcast[k]) #pragma omp task depend(in: bcast[k])
#define parallel depend(out: bcast[k]) #pragma omp task depend(out: bcast[k])
#define parallel depend(in: gemm[k]) #pragma omp task depend(in: gemm[k])
#define parallel depend(out: gemm[k]) #pragma omp task depend(out: gemm[k])
```
#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<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, k, 0, nt-1));

        // B(k+1:mt-1, :) -= A(k+1:mt-1, 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])
          {
            internal::gemm<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));
          }

        // 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));
          }
        }
      }
    }
  }
}