INITIAL INTEGRATION AND EVALUATION OF SLATE AND STRUMPACK

Pieter Ghysels, Sherry Li

Asim YarKhan, Jack Dongarra

Introduction

The aim of the joint milestone STMS10-57 was to integrate Software for Linear Algebra Targeting Exascale (SLATE) into the STRUctured Matrix PACKage (STRUMPACK). Two approaches were used to explore the integration of STRUMPACK and SLATE. First, STRUMPACK developers updated their code to use the SLATE's C++ interface for a number of compute intensive matrix operations. Second, SLATE's runtime-interception *scalapack_api* (described later) was used to transparently route a subset of ScaLAPACK calls to be executed by SLATE.

STRUMPACK

STRUMPACK is a library with dense and sparse linear algebra routines. STRUMPACK provides a sparse direct solver based on a multifrontal formulation of sparse Gaussian elimination, i.e., LU factorization. Compared to the algorithm used in SuperLU, a purely supernodal approach, the multifrontal Gaussian elimination in STRUMPACK leads to BLAS operations on larger dense blocks, and to a simpler communication pattern along the sparse elimination tree. For dense matrix algebra, STRUMPACK implements hierarchical matrix representations, using the Hierarchically Semi-Separable (HSS), Hierarchically Off-Diagonal Low-Rank (HODLR) and Block Low-Rank (BLR) formats.

SLATE

SLATE is a project that intends to provide basic dense matrix operations (e.g., matrix multiplication, rank-k update, triangular solve), linear systems solvers, least square solvers, singular value and eigenvalue solvers for current and future generation HPC architectures. SLATE aims to extract the full performance potential and maximum scalability from modern, many-node HPC machines with large numbers of cores and multiple hardware accelerators per node. As a part of SLATE's ultimate objective to replace the venerable Scalable Linear Algebra PACKage (ScaLAPACK) library, we are developing LAPACK and ScaLAPACK APIs. These APIs were initially described in an ICL Technical Report (ICL-UT-18-07).

SLATE LAPACK Compatibility API

SLATE has implemented a LAPACK/BLAS compatibility API and a wrapper library, referred to as the SLATE *lapack_api*. This API makes it possible for an application which uses the classic Fortran LAPACK/BLAS calls to easily access the advantages of using SLATE.

The SLATE LAPACK/BLAS compatibility API will accept BLAS calls from a Fortran-style interface that will then get executed using SLATE. This API is expressed by prepending a BLAS/LAPACK call with SLATE_. A Fortran call can be replaced fairly directly. For example, DGEMM would be simply translated:

```
DGEMM( transa, transb, m, n, standard parameters )

SLATE_DGEMM( transa, transb, m, n, standard parameters )
```

The SLATE API maps the Fortran-style parameters as needed and call the appropriate SLATE routine. However, some additional execution decisions need to be made that are not part of the standard API.

SLATE needs to know the execution target. The lapack_api library will set the execution target to GPU devices if they are available, otherwise use it will use the CPUs (in HostTask mode). The execution target can be overridden by the environment variable SLATE_TARGET.

The SLATE execution target is set in this order:

```
if env SLATE_LAPACK_TARGET={HostTask,HostBatch,HostNest,Devices}, use it

else if Devices are compiled in SLATE and available, use Devices

else use HostTask
```

SLATE requires a tile size (nb) for execution. For GPU targets, the lapack_api library uses a default tile size of 1024 and for CPU (HostTask) targets it sets a default tile size of 512. These values can overwritten via the environment variable SLATE_NB.

```
if env SLATE_LAPACK_NB=NNN is set, use NNN

else if Target=HostTask, nb=512

else if Target=Devices, nb=1024
```

For the lapack_api, SLATE does not intercept and override the original function names because SLATE links with the default BLAS/LAPACK libraries to do node-level computation and the name symbols would conflict. We will consider alternative interceptions methods (e.g., weak symbols with preloaded libraries) based on the future needs of our application partners.

SLATE ScaLAPACK Compatibility API

SLATE provides a compatibility API and library for ScaLAPACK/PBLAS Fortran-style interfaces and data layouts. This scalapack_api provides execution-time interception for routines that have been implemented in SLATE. These calls are executed using SLATE on distributed memory nodes and can use any GPUs available on those nodes. The transparent mapping of pre-existing ScaLAPACK/PBLAS calls to SLATE should make it substantially easier for application developers to access SLATE functionality and adopt SLATE usage.

In the scalapack_api library we create multiple names for each function, matching the standard Fortran name mangling. For example, for PDGEMM we define three named functions interfaces: PDGEMM, pdgemm, pdgemm_. These three interfaces call a SLATE routine that uses the data in the pre-existing

locations in memory. Note, there is no data copying required when using SLATE via the scalapack_api. Other parameters required for SLATE are obtained from the ScaLAPACK call. SLATE algorithms use tiled algorithms and the tiling is matched to the ScaLAPACK block size. Note: Users of the scalapack_api interface should take into account that SLATE works better with larger block sizes than ScaLAPACK.

Using these three Fortran name mangling schemes allows the API to intercept calls from programs expecting libraries that provide Fortran-style interface. To make usage transparent, we use the LD_PRELOAD capabilities of the linker to preload this library and capture the calls before they go to ScaLAPACK. Any functions that are not implemented in the SLATE scalapack_api simply resolve to their ScaLAPACK implementations. The scalapack_api obtains parameters like the execution target from environment variables.

For testing we use the testers provided within the ScaLAPACK distribution. Our testing uses a slightly modified data file, focusing on larger problems sizes and excluding some cases that are designed to trigger errors. Our current testing is as simple as

```
cd ./scalapack/trunk/PBLAS/TIMING;

env LD_PRELOAD=${SLATEDIR}/lib/libslate_scalapack_api.so mpirun -np 4 ./xspblas3tim
```

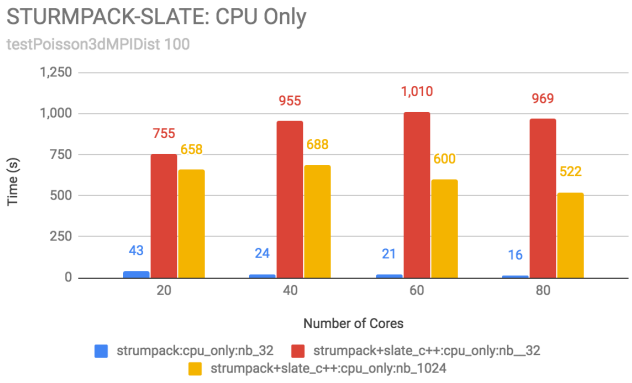
This preloading causes a subset of ScaLAPACK/PBLAS calls to execute using SLATE.

Integration and Evaluation

STRUMPACK with the SLATE C++ API

The initial integration of STURMPACK and SLATE was done by the STURMPACK team and required changes in STURMPACK to call the SLATE C++ API for matrix multiplication, LU factorization and LU solves. However, it is to be noted that the the SLATE LU solver is still being tuned and optimized. This first version of the STURMPACK-SLATE integration was timed on 4 nodes of summitdev, where each node contains 20 POWER8 cpus. The STURMPACK example executable that we used (testPoisson3dMPIDist 100) creates 3D k x k x k meshes where the largest dense matrices will be k^2 x k^2. However, as the testPoisson3dMPIDist computation continues, the matrix sizes decrease and much of the computation is done on much smaller matrices.

The initial version of the merged code (strumpack+slate_c++:cpu_only:nb_32) uses a ScaLAPACK tile size of 32, and the execution took much longer than using STURMPACK alone (strumpack:cpu_only:nb_32); the execution time was actually increasing as we increased the number of nodes. Since SLATE tends to perform better on larger tile size, we attempted to improve performance by moving to a larger tile size of 1024 (strumpack+slate_c++:cpu_only:nb_1024). The larger tile size did decrease the time taken but



it still lagged far behind using STURMPACK alone. Our observation is that SLATE is not yet tuned to perform well on the smaller matrix sizes that dominate most of this computation.

We turn to the SLATE ScaLAPACK compatibility API to capture a subset of the dense linear algebra calls (avoiding the LU factorizations) and send the larger matrix problems to SLATE to be evaluated using GPUs. The smaller matrix sizes are kept to be executed via standard ScaLAPACK.

STRUMPACK with ScaLAPACK compatibility API

The SLATE ScaLAPACK compatibility API can intercept ScaLAPACK function calls and create the necessary structures to execute the calls using SLATE. For the second integration effort, we use the unaltered STURMPACK implementation and capture the ScaLAPACK functions via the SLATE scalapack_api. Working with the STURMPACK executable, we realized that it is useful to be able to choose the functions that are intercepted and to specify the data sizes that benefit from this interception. We will incorporate this ability as standard in SLATE after determining the appropriate way for a user give their specifications.

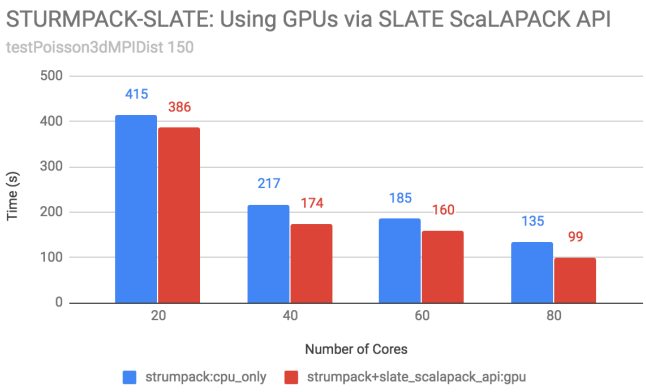
We use 4 nodes on summitdev to run some timing experiments. Each node on summitdev contains 20 Power8 cpus and 4 Nvidia P100 GPUs. The experiments are executed so that each process is bound to one GPU and 5 cpus. In earlier work we have found that this process binding configuration gives good memory locality for local data movement and provides a high throughput rate for MPI traffic. The STURMPACK code is modified to use a large block-tile size of 1024 which tends to improve GPU performance. The SLATE ScaLAPACK compatibility API was modified to constrain which calls are redirected to SLATE GPU execution. Using the compatibility API is a matter of preloading the scalapack_api library.

```
export LD_PRELOAD=$SLATE_HOME/lib/libslate_scalapack_api.so

export SLATE_SCALAPACK_TARGET=Devices

mpirun -n16 -a1 -c5 -g1 ./testPoisson3dMPIDist 150
```

The STURMPACK example that we use (testPoisson3dMPIDist 150) generates large matrices on the order of 150^2 = 22500. Using the modified SLATE ScaLAPACK compatibility interface, we capture problems that are considered large enough (M>8000 and N>8000) to benefit from execution via SLATE. The value of 8000 is currently an arbitrary constraint, and it results in approximately 49% of the PDGEMM calls being sent to SLATE. Capturing just a subset of the PDGEMM calls and sending them to be executed via SLATE on GPUs results in a very modest time improvement for larger problems.



There are a large number of LU factorizations in STURMPACK that we are currently not executing via SLATE. When SLATE has completed the optimization of LU on GPUs, these factorization can also be handled by SLATE and we will experimentally evaluate the performance.

Summary

The initial STURMPACK-SLATE integration experiments described here showed that SLATE needs to substantially improve performance for small matrix sizes. The SLATE ScaLAPACK compatibility API enabled STURMPACK to use GPUs for execution, however an improved LU implementation should provide more GPU based speedup. One unexpected outcome from these experiments is that we discovered the need for changes to the SLATE ScaLAPACK API so that it can dynamically switch from executing via SLATE or via the default ScaLAPACK implementation. These changes will soon be incorporated into SLATE's compatibility API's.