Initial Integration and Evaluation of SLATE Parallel BLAS in LATTE

Marc Cawkwell, Danny Perez, Arthur Voter

Asim YarKhan, Gerald Ragghianti, Jack Dongarra,

Introduction

The aim of the joint milestone STMS10-52 was to implement the high performance linear algebra library SLATE into the quantum molecular dynamics (QMD) package LATTE. The performance of LATTE, which is written in Fortran90, depends heavily on calls to high performance math libraries for matrix diagonalization and generalized matrix-matrix multiplication. These math libraries are usually vendor-optimized versions of the original LAPACK and BLAS packages and include Intel's MKL, AMD's ACML, and Nvidia's cuBLAS. We have assessed the performance of the generalized matrix-matrix multiplication subroutine, DGEMM, in SLATE on both multi-core CPUs and general purpose Nvidia GPUs.

LATTE

Quantum molecular dynamics simulations are used to directly study the motion of atoms in molecules and materials using interatomic forces computed from the underlying electronic structure. These simulations are vastly more computationally expensive than those based on simple, parameterized pairwise forms for the interatomic potentials but they provide a rigorous, predictive treatment of the physics and chemistry of materials, which are crucial for studying the making and breaking of interatomic bonds. LATTE uses an approximate, semi-empirical description of interatomic bonding to compute the interatomic forces that drive MD trajectories. Its use of semi-empirical theory and novel solvers provide significant improvements in performance over regular *ab initio* methods. Nevertheless, the scaling of the computational time is still cubic, $O(N^3)$, with respect to the number of atoms, N, being simulated.

The self-consistent calculation of the density matrix, P, from the Hamiltonian matrix, H, is the bottleneck step in calculating the potential energy and interatomic forces in QMD simulations. Matrix diagonalization has been the workhorse algorithm for computing the density matrix in electronic structure theory for many years. It is unquestionably powerful because it allows a finite electronic temperature to be added to a simulation relatively easily and provides a wealth of information of the electronic eigenspectrum, but it exhibits $O(N^3)$ complexity and it is difficult to parallelize. The LATTE code has instead pursued algorithms based on generalized matrix-matrix multiplication because of i) the close-to-peak performance of generalized matrix-matrix multiplication subroutines on a variety of architectures, ii) the potential for achieving O(N) performance via sparse data structures, and iii) the ease at which the workload can be distributed over distributed memory nodes.

Algorithms for obtaining an idempotent density matrix by purification in a recursive series of matrix polynomials date back many years.[1, 2] The SP2 algorithm of Niklasson[3, 4] overcomes the shortcomings

exhibited by several alternate approaches by requiring no prior knowledge of the chemical potential and minimizing storage for intermediate matrices.[5] The SP2 algorithm starts with the Hamiltonian, H, rescaled such that its eigenvalues occupy the interval [0,1] in reverse order, that is,

$$X_0 = \frac{\varepsilon_{max} I - H}{\varepsilon_{max} - \varepsilon_{min}}$$

where ε_{max} and ε_{min} are estimates for the largest and smallest eigenvalues of the Hamiltonian, respectively. The density matrix, P, is obtained from a recursive expansion,

$$P = f_i(f_{i-1}(\cdots f_1(X_0)\cdots)))$$

where the matrix polynomials are

$$X_{i+1} = 2X_i - X_i^2$$

or

$$X_{i+1} = X_i^2$$

depending on whether the trace of X_i is larger or smaller than the target occupancy, respectively. Hence at each step in recursive expansion only one generalized matrix-matrix multiplication is necessary, along with storage for one intermediate array. The drawback to the SP2 algorithm is that a gap must be present at the chemical potential, i.e., its performance is poor for metals, and that it does not allow for the fractional occupation of states in the vicinity of the chemical potential.

SLATE

The objective of the Software for Linear Algebra Targeting Exascale (SLATE) project is to provide fundamental dense linear algebra capabilities to the US Department of Energy and to the high-performance computing (HPC) community at large. To this end, SLATE will 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.

The ultimate objective of SLATE is to replace the venerable Scalable Linear Algebra PACKage (ScaLAPACK) library, which has become the industry standard for dense linear algebra operations in distributed memory environments. However, after two decades of operation, ScaLAPACK is past the end of its lifecycle and overdue for a replacement, as it can hardly be retrofitted to support hardware accelerators, which are an integral part of today's HPC hardware infrastructure.

Primarily, 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. For typical dense linear algebra workloads, this means getting close to the theoretical peak performance and scaling to the full size of the machine (i.e., thousands to tens of thousands of nodes). This is to be accomplished in a portable manner by relying on standards like MPI and OpenMP.

During interactions between the SLATE and EXAALT teams it became clear that joint work would be much easier if there were a simple way to call SLATE from Fortran code. The SLATE team has developed two APIs to make this easier.

- A BLAS compatibility API layer (called lapack_api) to make it easier to call SLATE from a sequential Fortran style code.
- A PBLAS compatibility layer (called scalapack_api) to allow transparent linking to distributed-memory SLATE routines via a Linux preload library interception.

The advantage of using SLATE lies in its use of multiple GPU resources to speed up computation. Additionally, SLATE will use an applications data from its location in memory, using the pre-existing LAPACK or ScaLAPACK data layout.

BLAS Compatibility API

This 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, for DGEMM would be simply translated:

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

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

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

SLATE needs to know the execution target. The lapack_api will set the execution target to GPU devices if they are available, otherwise 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_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 sets a default tile size of 1024 and for CPU (HostTask) targets it sets a default tile size of 512. These values can overridden via the environment variable SLATE_NB.

```
if env SLATE_NB is set, use it

else if Target=HostTask, nb=512

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

For the lapack_api, we did not try to intercept and override the original function names because SLATE links with 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 needs of our application partners.

To test our lapack_api we have used the testers provided with the LAPACK distribution. For example, the BLAS/TESTING/cblat3.f tester, which contains complex-level-3-blas testers was altered so that the BLAS3 routine calls were prepended with "SLATE_". The test input was modified to remove some tests designed to trigger errors. Running the tester redirects the calls to SLATE algorithms and checks the results.

PBLAS Compatibility API

SLATE provides a compatibility API for PBLAS/ScaLAPACK style interfaces and data layouts. This scalapack_api library provides direct interception for PBLAS calls that have been implemented in SLATE. These PBLAS calls then run using SLATE algorithms on distributed memory nodes and use the GPUs available on those nodes. We hope that transparently mapping pre-existing ScaLAPACK call to SLATE will 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 provided locations in memory, note, there is no data movement required here. Other parameters required for SLATE are obtained from the ScaLAPACK call. SLATE algorithms use tiled algorithms and the tiling we use is simply 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 the 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 library is still under development, but we use the testers provided within the ScaLAPACK distribution to check functionality. Our test uses a slightly modified data file to drive the testing, focusing on larger problems sizes and excluding some cases that are designed to trigger errors. However, 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 simple preloading enables the standard ScaLAPACK to execute using the multiple GPUs available on the nodes. We are using single precision BLAS3 tester in this example because our hardware nodes contain Nvidia GTX 1060 GPUs which don't have good double precision performance.

Currently the scalapack_api contains implementations for the gemm, symm, syr2k, syrk, trmm, trsm and potrf functions in all precisions (single, double, complex, double complex). We will continue to add functionality as it is implemented in SLATE.

Performance Experiments

One of the most appealing aspects of the SLATE library is that the same API is used to call the threaded DGEMM for CPU and a blocked DGEMM for multiple GPUs. The user then decides on which architecture the calculations are to be performed by switching an environment variable before execution.

The performance of the SLATE DGEMM was evaluated using a standalone version of the SP2 algorithm where the rescaled Hamiltonians, X_0, were read from file. Only the recursive expansion required to obtain an idempotent density matrix was timed. These calculations were performed on the SummitDev supercomputer at ORNL, which is intended to mimic the OLCF's next supercomputer, Summit.

SummitDev is based on IBM POWER8 processors and NVIDIA P100 (Pascal) accelerators, and is one generation behind Summit, which is based on IBM POWER9 processors and 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 sizes of problems studied so far did not justifiy the use of SLATE's distributed computing capabilities. Therefore, here we are presenting single node runs. Fig. 1 presents the wall time to compute the density matrix as a function of the number of OpenMP threads for an X_0 of dimension 6912x6912 using the IBM ESSL and SLATE DGEMM. Fig. 2 presents the wall time to compute the density matrix when using SLATE's GPU acceleration capabilities.

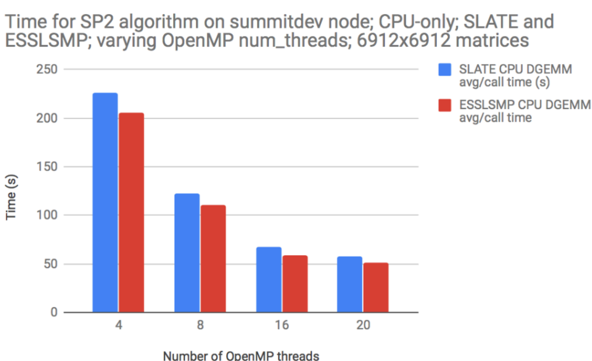

Figure 1: Multicore Performance.

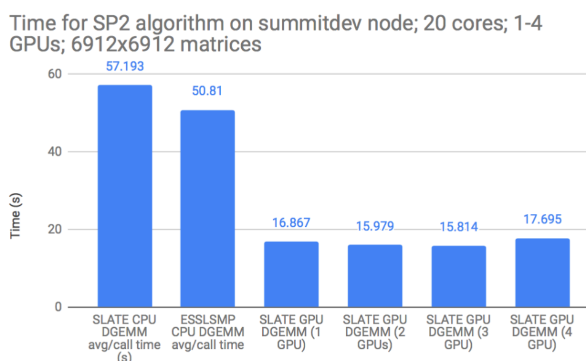

Figure 2: Accelerated Performance.

For multicore runs, both SLATE and ESSL deliver higher performance with increasing number of threads, with ESSL showing a marginal performance advantage. For accelerated runs, SLATE delivers 3.5x speedup compared to multicore runs, but fails to deliver increased performance with increased number of GPUs, which is not surprising given the relatively small problem size.

Finally, we have successfully replaced all of the DGEMMs in the LATTE code with calls to the SLATE library. This process was extremely easy because of the API provided in the SLATE package. DGEMMs are called in LATTE about 60 times per MD time step. Most of these are within the recursive SP2 loop for the density matrix, but they are also used in orthogonalization steps and during the calculation of the Pulay force. Unfortunately, this exercise failed to deliver further performance improvements, most likely due to the small granularity of the underlaying DGEMM calls.

Summary

The collaboration between the EXAALT team and the SLATE team lead to the development of LAPACK and ScaLAPACK compatibility interfaces in SLATE, which make it trivial to call SLATE from Fortran codes, that currently rely on the aforementioned legacy packages. The LAPACK API only requires prefixing of the function call with "SLATE_" without any changes to the functions' signatures. At the same time, the ScaLAPACK API requires no changes to the source code, i.e., can be applied at link time. The EXAALT team found the SLATE compatibility APIs to be easy to use, and the switch to accelerated execution to be seamless (simply setting an environment variable).

At the same time, SLATE delivered limited performance benefits, for the studied problem sizes. While multicore runs delivered performance on par with ESSL, accelerated runs delivered only small performance improvements. Also, at this stage, SLATE was not able to benefit from the use of multiple GPUs. These findings are not surprising given the small size of the studied problem sizes. Going forward, the SLATE team will investigate the opportunities for improving SLATE's performance for small matrices, while the EXAALT team will look into increasing the problem sizes.

References

1. R. McWeeny, "THE DENSITY MATRIX IN SELF-CONSISTENT FIELD THEORY .1. ITERATIVE CONSTRUCTION OF THE DENSITY MATRIX", *Proceedings of the Royal Society of London Series a-Mathematical and Physical Sciences*, **235**, 496-509 (1956).

2. A. H. R. Palser and D. E. Manolopoulos, "Canonical purification of the density matrix in electronic-structure theory", *Physical Review B*, **58**, 12704-12711 (1998).

3. A. M. N. Niklasson, "Expansion algorithm for the density matrix", *Physical Review B*, **66**, (2002).

4. E. H. Rubensson and A. M. N. Niklasson, "interior Eigenvalues from Density Matrix Expansions in Quantum Mechanical Molecular Dynamics", *SIAM J. Sci. Comput.*, **36**, 148 (2014).

5. E. Rudberg and E. H. Rubensson, "Assessment of density matrix methods for linear scaling electronic structure calculations", *Journal of Physics-Condensed Matter*, **23**, (2011).

6. M. J. Cawkwell, E. J. Sanville, S. M. Mniszewski and A. M. N. Niklasson, "Computing the Density Matrix in Electronic Structure Theory on Graphics Processing Units", *Journal of Chemical Theory and Computation*, **8**, 4094-4101 (2012).

7. M. J. Cawkwell, M. A. Wood, A. M. N. Niklasson and S. M. Mniszewski, "Computation of the Density Matrix in Electronic Structure Theory in Parallel on Multiple Graphics Processing Units", *Journal of Chemical Theory and Computation*, **10**, 5391-5396 (2014).