SLATE port to AMD and Intel platforms

Ahmad Abdelfattah
Mohammed Al Farhan
Cade Brown
Mark Gates
Dalal Sukkari
Asim YarKhan
Jack Dongarra

Innovative Computing Laboratory

April 5, 2021

ICL INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support 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
04-2021	first publication

```
@techreport{abdelfattah2021slate-port,
  author={Ahmad Abdelfattah and Mohammed Al Farhan and Cade Brown and Mark Gates
          and Dalal Sukkari and Asim YarKhan and Jack Dongarra},
  title={{SLATE} port to {AMD} and {Intel} platforms, {SWAN} No. 16},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2021},
  month={4},
  number={ICL-UT-21-01},
  note={revision 04-2021},
  url={https://www.icl.utk.edu/publications/swan-016},
}
```

Contents

1 Introduction

SLATE implements GPU-accelerated linear algebra, relying primarily on vendor-provided GPU BLAS for performance, in particular batched BLAS routines. Initially, SLATE was written using NVIDIA's CUDA and cuBLAS for GPU acceleration. At the time that the SLATE project was started, it was unclear what GPU technologies would exist for other platforms [1]. Since then, AMD has developed their ROCm platform, which includes the HIP portability layer, and Intel has developed their oneAPI platform based on SYCL, a C++ descendant of OpenCL. These software stacks will be used on upcoming exascale systems: Frontier, built by AMD for Oak Ridge National Laboratory, and Aurora, built by Intel for Argonne National Laboratory. Therefore, to support these new platforms, SLATE needed to be ported to both ROCm and Intel oneMKL.

2 Porting SLATE to use BLAS++

The strategy we employed in SLATE is to leverage the BLAS++ library as a portability layer. Therefore, we removed CUDA and cuBLAS functions from SLATE, and replaced them with portable abstractions in the BLAS++ library. This included:

- Replacing CUDA streams and cuBLAS handles with BLAS++ queues.
- Replacing CUDA functions (cudaMalloc, cudaFree, cudaMemcpy, cudaStreamSynchronize, etc.) with BLAS++ wrappers.
- Replacing cuBLAS functions (cublas*Gemm, cublas*GemmBatched, etc.) with BLAS++ wrappers.

2.1 Simplified code

In many cases, the new code using BLAS++ is simpler than the original code using cuBLAS, for several reasons. BLAS++ handles checking errors from CUDA and cuBLAS, and throwing errors that occur. Thus the typical error handler macros are no longer needed. Compare the original code:

```
void* dev_mem;
slate_cuda_call(
    cudaMalloc((void**)&dev_mem, size));
```

with the revised code:

```
void* dev_mem = blas::device_malloc<char>(size);
```

BLAS++ will also handle setting the device as needed based on the queue, eliminating many uses of `cudaSetDevice`. Compare Algorithm 2.1 line 2 with Algorithm 2.2. Indeed, with SYCL used by Intel oneAPI, there is no concept of a "current device", so set device calls may be even further eliminated in the future.

Algorithm 2.1 Original internal::gemm implementation snippet

```
slate_cuda_call(                                                                    1
    cudaSetDevice(device));                                                         2
                                                                                    3
// cublas_handle uses this stream                                                   4
cudaStream_t stream = C.compute_stream(device);                                     5
cublasHandle_t cublas_handle = C.cublas_handle(device);                             6
                                                                                    7
slate_cuda_call(                                                                    8
    cudaMemcpyAsync(C.array_device(device, batch_arrays_index),                     9
                    C.array_host(device, batch_arrays_index),                       10
                    sizeof(scalar_t*)*batch_count*3,                                11
                    cudaMemcpyHostToDevice,                                         12
                    stream));                                                       13
                                                                                    14
if (batch_count_00 > 0) {                                                           15
    if (layout == Layout::ColMajor) {                                               16
        slate_cublas_call(                                                          17
            cublasGemmBatched(                                                      18
                cublas_handle,  // uses stream                                      19
                cublas_op_const(opA), cublas_op_const(opB),                         20
                mb00, nb00, kb,                                                     21
                &alpha, (const scalar_t**) a_array_dev, lda00,                      22
                        (const scalar_t**) b_array_dev, ldb00,                      23
                &beta,                     c_array_dev, ldc00,                      24
                batch_count_00));                                                   25
    }                                                                               26
    else {                                                                          27
        slate_cublas_call(                                                          28
            cublasGemmBatched(                                                      29
                cublas_handle,  // uses stream                                      30
                cublas_op_const(opB), cublas_op_const(opA),                         31
                nb00, mb00, kb,                                                     32
                &alpha, (const scalar_t**) b_array_dev, ldb00,                      33
                        (const scalar_t**) a_array_dev, lda00,                      34
                &beta,                     c_array_dev, ldc00,                      35
                batch_count_00));                                                   36
    }                                                                               37
}                                                                                   38
                                                                                    39
...                                                                                 40
                                                                                    41
slate_cuda_call(                                                                    42
    cudaStreamSynchronize(stream));                                                 43
```

BLAS++ handles the `layout` argument to switch between row and column-major, eliminating one case from SLATE, as seen in Algorithm 2.1 line 16.

In BLAS++, the batch arrays for the A, B, and C matrices are passed as `std::vectors` on the host. BLAS++ handles copying these vectors to the GPU, alleviating SLATE from that task (see Algorithm 2.1 line 8).

BLAS++ provides overloading based on the data type, which is needed for the template code in SLATE. For instance, `blas::gemm( ..., A, ..., B, ..., C, ..., queue )` calls one of `cublasSgemm`, `cublasDgemm`, `cublasCgemm`, or `cublasZgemm`, depending on the data type of the A, B, and C matrices. Previously SLATE had its own overloaded lightweight wrappers for cuBLAS functions; these were removed in favor of the BLAS++ wrappers.

Algorithm 2.2 Revised internal::gemm implementation snippet

```
std::vector<Op>          opA_(1, opA);
std::vector<Op>          opB_(1, opB);
std::vector<scalar_t> alpha_(1, alpha);
std::vector<scalar_t>  beta_(1, beta);
std::vector<int64_t>        k(1, kb);
// info size 0 disables slow checks in batched BLAS++.
std::vector<int64_t> info;

blas::Queue* queue = C.compute_queue(device, queue_index);
assert(queue != nullptr);

if (c_array00.size() > 0) {
    std::vector<int64_t>    m(1,  mb00);
    std::vector<int64_t>    n(1,  nb00);
    std::vector<int64_t> ldda(1, lda00);
    std::vector<int64_t> lddb(1, ldb00);
    std::vector<int64_t> lddc(1, ldc00);
    blas::batch::gemm(
        layout, opA_, opB_,
        m, n, k,
        alpha_, a_array00, ldda,
                b_array00, lddb,
        beta_,  c_array00, lddc,
        c_array00.size(), info, *queue);
}

...

queue->sync();
```

2.2 New optimizations

BLAS++ also extends the number of batched routines available. For BLAS, cuBLAS currently provides only batched `gemm` and batched `trsm`. BLAS++ implements all Level 3 batched BLAS routines: `gemm`, `hemm`, `herk`, `her2k`, `symm`, `syrk`, `syr2k`, `trmm`, `trsm`. If an underlying batched implementation does not exist, BLAS++ uses a multi-stream approach, calling the non-batched implementation for each matrix in a different stream, currently forking up to 10 parallel streams, which are then joined. This allowed SLATE to move some operations from non-batched to batched, such as `herk` on diagonal tiles in `internal::herk`, gaining a performance improvement even with CUDA.

2.3 Lost optimizations

The port to BLAS++ did eliminate some optimizations that were in SLATE. Taking `gemm` as a typical BLAS call, it has 3 matrices: A, B, C. In SLATE, the batch arrays for all 3 were packed one after another into a single array, and a single `cudaMemcpy` was used to copy it to the GPU. In BLAS++, these are 3 `std::vector` arguments, each of which must be copied individually to the GPU, invoking 3 small memcpy calls instead of 1 larger memcpy call. This change appeared to have negligible impact on performance.

SLATE also had a split gemm implementation, where preparing the batch arrays and copying them to the GPU was handled in one OpenMP task, and the actual execution of the gemm was

in another OpenMP task [2]. This was motivated by contention between different OpenMP tasks for the DMA to copy data to the GPU. BLAS++ does not currently support this model. While this may affect performance when using many GPUs such as on an 8 GPU NVIDIA DGX node, it does not seem to affect performance on Summit, where each MPI process has either 1 or 3 GPUs, depending on job settings.

2.4 Future improvements

One significant performance issue was discovered in BLAS++, where if the `std::vector` for info was of size `batch_count`, the BLAS++ code to check for errors added significant overhead. This may be a collision of OpenMP constructs, as BLAS++ attempts to multithread the checking, but in SLATE the BLAS++ call occurs inside an OpenMP task. We will investigate further to resolve the issue. Currently we set info to be of size 0, which avoids the overhead, as seen in Algorithm 2.2 line 7.

Batched routines in BLAS++ have a different interface than in cuBLAS. Most arguments in batched BLAS++ are passed as a `std::vector`. If an argument's vector is of size 1, that argument is fixed for all items in the batch; if it is of size `batch_count`, it is variable across items in the batch. This potentially adds much more flexibility by allowing batches with different size matrices. However, as the underlying implementation (in cuBLAS, rocBLAS, etc.) presently has only fixed size matrices, it is most efficient to retain this restriction in use. Thus, SLATE uses size 1 vectors for most arguments including transposition operation, matrix dimensions, leading dimensions, alpha, and beta, as seen in Algorithm 2.2.

The need to set up `std::vector` objects for every single argument is a bit of a burden for using batched BLAS++. A possible revision of BLAS++ would use C++ techniques to template the batched functions so they could take either singletons (e.g., `int m`) or vectors (e.g., `std::vector<int>& m`). We briefly investigated this for feasibility, but need more work to implement in BLAS++. This would further simplify SLATE's code, eliminating 10 lines from Algorithm 2.2.

Currently, SLATE's routines generally build 4 batches: one for interior tiles, one for the bottom row, one for the right column, and one for the bottom-right tile. It is assumed that tiles within each of these batches are the same size, but different batches could have different sizes. This fits well with the "group API" for batch routines, first proposed by Intel and incorporated in the Batched BLAS proposal [3]. BLAS++ could be extended to support the group API, internally mapping to the fixed size API for backends that lack the group API.

3 Porting BLAS++ to rocBLAS

Most of the work in porting BLAS++ to rocBLAS was in refactoring the BLAS++ code to be more usable by applications. Previously, BLAS++ Queues were defined only if cuBLAS was available. This would necesitate having `#ifdef BLAS_HAVE_CUBLAS` in application codes like SLATE. We reorganized code so even if no GPU BLAS is available, the Queue class is defined, though it cannot be instantiated without a GPU device being available. Similarly, all GPU BLAS functions are now always defined, though they cannot be called without a GPU device and Queue, and

throw a `blas::Error`. As a special case, `blas::get_device_count()` simply returns 0 if no GPU backend exists. This reorganization eliminates the need for `#ifdef` related to BLAS++ GPU backends in SLATE code, and in other applications leveraging BLAS++.

The configuration and CMake scripts were also updated for ROCm. As part of this, preprocessor defines such as `BLAS_HAVE_CUBLAS` were moved into a header `include/blas/defines.h` created during configuration, which allows the application to know what compile-time options were used, including if a GPU backend is available.

After this refactoring of the BLAS++ code, changes related to ROCm are confined to defining what exists in the Queue class (streams, handles), and simple wrappers around BLAS and utility functions. For instance, ROCm-specific code is limited to these source code files in BLAS++:

```
include/blas/device.hh
src/device_batch_trmm.cc    // only to fix an bug with ROCm 4.0
src/device_queue.cc
src/device_utils.cc
src/rocblas_wrappers.cc
```

Higher level BLAS++ routines are mostly backend-independent and call these lower level wrappers. For instance, `blas::batch::herk` handles checking arguments, interpretting the `std::vector` arguments, copying these batch arrays to the GPU, and then invoking the low level `herk` wrapper around `cublasHerk` or `rocblasHerk`.

As BLAS++ already had a cuBLAS backend, we chose to port it to use rocBLAS rather than hipBLAS. Porting to hipBLAS, which is ROCm's own portability layer across cuBLAS and rocBLAS, would needlessly add another layer.

4 Porting BLAS++ to oneMKL

The port of BLAS++ to Intel oneMKL was significantly more difficult, because SYCL has some very different constructs than CUDA and ROCm/HIP. Indeed, HIP is specifically designed to closely match CUDA, with porting often being a matter of replacing "cuda" with "hip" and the like. Significant differences in SYCL compared to CUDA and HIP include:

- In CUDA and ROCm, devices are enumerate by integers: device 0, 1, ... In contrast, in SYCL devices must be queried, and are each represented by an opaque object. We handle this query internally, so Queues can still be constructed by an integer device ID.
- As previously mentioned, SYCL has no notion of a current device. Memory allocation must take a SYCL queue to know what device it uses. This necessitated a change in the BLAS++ API to take a BLAS++ queue in device malloc and free routines.
- It is unclear how or if SYCL supports pinned memory.
- It is unclear if SYCL has 2D memcpy, which is frequently used in dense linear algebra to copy submatrices. We emulate 2D memcpy by looping over columns of the submatrix. (In SLATE, tiles are frequently contiguous, so can be copied with a plain (1D) memcpy instead of 2D memcpy.)

- It is unclear how to synchronize between SYCL queues, in the manner we synchronize CUDA and HIP streams using events. Thus, presently BLAS++ with oneMKL does not support multi-stream for batched BLAS.
- Device memory allocation failed with segfaults in our testing. Instead, we used universal shared memory (USM) allocation.

We hope that some of these differences can be resolved as our understanding of SYCL and oneMKL improves, and as the software stack matures. However, so far none of these differences has been a complete barrier to having a portable implementation of SLATE with BLAS++.

References

[1] Ahmad Abdelfattah, Hartwig Anzt, Aurelien Bouteiller, Anthony Danalis, Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, Stephen Wood, Panruo Wu, Ichitaro Yamazaki, and Asim YarKhan. Roadmap for the Development of a Linear Algebra Library for Exascale Computing: SLATE: Software for Linear Algebra Targeting Exascale. SLATE Working Notes 1, ICL-UT-17-02, 2017. URL http://www.icl.utk.edu/publications/swan-001.

[2] Mark Gates, Ali Charara, Asim YarKhan, Dalal Sukkari, Mohammed Al Farhan, and Jack Dongarra. SLATE working note 14 Performance Tuning SLATE. Technical Report ICL-UT-XX-XX, Innovative Computing Laboratory, University of Tennessee, December 2019. revision 12-2019.

[3] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nick Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. A set of batched basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 2020. Accepted, in press.