SLATE WORKING NOTES

7

Implementation of the C++ API for Batch BLAS

Ahmad Abdelfattah
Mark Gates
Jakub Kurzak
Piotr Luszczek
Jack Dongarra

Innovative Computing Laboratory

June 29, 2018

ICL INNOVATIVE COMPUTING LABORATORY

THE UNIVERSITY OF TENNESSEE KNOXVILLE

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nation's exascale computing imperative.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Revision	**Notes**
06-2018	first publication

```
@techreport{abdelfattah2018implementation,
  author={Abdelfattah, Ahmad and Gates, Mark and Kurzak, Jakub and
          Luszczek, Piotr and Dongarra, Jack},
  title={{SLATE} Working Note 7: Implementation of the C++ {API} for Batch {BLAS}},
  institution={Innovative Computing Laboratory, University of Tennessee},
  year={2018},
  month={June},
  number={ICL-UT-XX-XX},
  note={revision 06-2018}
}
```

Contents

Contents ii

List of Figures iii

List of Tables iv

1 Highlights of the C++ API 1
- 1.1 A Unified API for All Purposes . . . 1
- 1.2 API Example . . . 2
- 1.3 Ensuring Correct Semantics . . . 3
- 1.4 BLAS Error Checking . . . 3
- 1.5 Summary . . . 4

2 Implementation Details 5
- 2.1 GPU Abstraction Layer . . . 5
- 2.2 The blas::Queue Class . . . 8
- 2.3 Supporting Different Use Cases . . . 10
- 2.4 A Complete Example for Batch GEMM . . . 10
- 2.5 Summary . . . 12

3 Performance Numbers 13
- 3.1 Analysis of Overheads . . . 13
- 3.2 Small-to-medium Sizes . . . 14
- 3.3 Relatively Large Sizes . . . 15
- 3.4 Summary . . . 17

Bibliography 18

List of Figures

3.1 Performance analysis of batch DGEMM on square matrices of size 32 14
3.2 Performance analysis of batch DGEMM on square matrices of size 128 15
3.3 Performance analysis of batch DGEMM on square matrices of size 256 16
3.4 Performance analysis of batch DGEMM on square matrices of size 512 16

List of Tables

1.1 An example batch GEMM use case 3

CHAPTER 1

Highlights of the C++ API

This chapter revisits the design of the C++ API for batch BLAS, which has been previously proposed in SLATE Working Note #4 [1]. The chapter focuses on the design aspects that would affect the implementation of the API itself.

1.1 A Unified API for All Purposes

The main motivation behind a C++ API is to take advantage of certain features that are absent in other programming languages, such as C and Fortran. In fact, the diverse expectations of what batch BLAS routines can do have led to many divergent C APIs from different vendors and library developers, as previously discussed [1]. Two main reasons have led to such divergent APIs. The first is inability to overload function names in C, and the second is the absence of standardized containers that can encapsulates useful information for batch BLAS routines.

The proposed C++ API is a unified API that can cover all the possible use cases of a batch BLAS routine. It uses the `std::vector` container to encapsulate vector arguments. The size of the vector determines whether an input argument is fixed/varied across the batch. For example, a vector of size one means that such an argument is unified across the batch. This eliminates the need for unnecessary duplications, and removes the border between batches of “strictly fixed” sizes and batches of “strictly variable” sizes. The API is flexible enough to host any combination of unified/varying arguments.

The proposed API also takes advantage of the function-overloading. In order to support both host CPUs and accelerators (e.g. GPUs), the API distinguishes between the two targets by using an extra argument for accelerators. Such an argument is the C++ `Queue` class, which abstracts the accelerator data types, runtime calls, optimized BLAS calls, and other accelerator-specifics that

are dependent on the hardware itself. The proposition also uses C++ namespaces to distinguish between batch BLAS calls, and regular (non-batch) calls.

1.2 API Example

The C++ API for batch matrix multiplication `GEMM` looks like:

```
namespace blas{

namespace batch{

inline
void gemm(
    std::vector<blas::Op> const &transA,
    std::vector<blas::Op> const &transB,
    std::vector<int64_t>  const &m,
    std::vector<int64_t>  const &n,
    std::vector<int64_t>  const &k,
    std::vector<float >   const &alpha,
    std::vector<float*>   const &Aarray, std::vector<int64_t> const &ldda,
    std::vector<float*>   const &Barray, std::vector<int64_t> const &lddb,
    std::vector<float >   const &beta,
    std::vector<float*>   const &Carray, std::vector<int64_t> const &lddc,
    const size_t batch,                  std::vector<int64_t>       &info );

}

}
```

All vector arguments are passed by reference in order to avoid unnecessary copies. The same routine name is overloaded to support accelerators. In this case, the API accepts a `blas::Queue` object as an extra argument.

```
namespace blas{

namespace batch{

inline
void gemm(
    std::vector<blas::Op> const &transA,
    std::vector<blas::Op> const &transB,
    std::vector<int64_t>  const &m,
    std::vector<int64_t>  const &n,
    std::vector<int64_t>  const &k,
    std::vector<float >   const &alpha,
    std::vector<float*>   const &Aarray, std::vector<int64_t> const &ldda,
    std::vector<float*>   const &Barray, std::vector<int64_t> const &lddb,
    std::vector<float >   const &beta,
    std::vector<float*>   const &Carray, std::vector<int64_t> const &lddc,
    const size_t batch,                  std::vector<int64_t>       &info,
    blas::Queue &queue );

}

}
```

1.3 Ensuring Correct Semantics

As mentioned before, the size of each vector plays an essential role in the semantics and the behavior of the batch routine. Each batch routine is equipped with a thin layer, called size error checking, that checks for the correctness of the semantics imposed by the vector sizes. In fact, there are some semantically-wrong use cases for each batch routine. Such use cases are detected in the size error checking layer, and an exception is thrown to the user. As an example, each input vector can have one of two sizes (1 or `batch`). However, it is semantically wrong to have the matrix `A` vector of size = 1, for example, while the corresponding `lda` vector is of size `batch`. This is because a single matrix cannot have multiple leading dimensions. The size error checking layer is responsible for detecting similar wrong use cases at the very beginning of the routine call.

batch index	0	1	2	3
`transA`	`NoTrans`			
`transB`	`NoTrans`		`Trans`	
`m`	16			
`n`	32	~~8~~ -1	24	50
`k`	16			
`alpha`	α_0	α_1	α_2	α_3
`A`	$A_{(16\times16)}$			
`lda`	20			
`B`	$B_{0(16\times32)}$	$B_{1(16\times8)}$	$B_{2(24\times16)}$	$B_{3(50\times16)}$
`ldb`	16	18	~~24~~ 20	50
`beta`	β_0	β_1	β_2	β_3
`C`	$C_{0(16\times32)}$	$C_{1(16\times8)}$	$C_{2(16\times24)}$	$C_{3(16\times50)}$
`ldc`	16			

Table 1.1: An example batch `GEMM` use case

1.4 BLAS Error Checking

The regular BLAS error checking is available in the C++ batch routines. However, it provides more flexibility to the user, who can choose one of three available error checking modes. It has been shown before that the use of the legacy `xerbla()` function is not practical, and so it was replaced by the `info` parameter. The latter is another `std::vector` whose size determines the behavior of the error checking layer. The user can totally skip the BLAS error checking layer by passing `info` as an empty vector. If `info.size()` returns 1, that turns on an argument-based error checking, where each argument is checked individually across all problems in the batch. If an error is detected, the `info` parameter is set to a unique value that reflects the index of the wrong argument, and an exception is thrown to the user. It is important to note that this mode does not specify which entry of the vector caused an error. Checking subsequent arguments is also canceled once an error is found in the current argument. On the other hand, if `info.size()` is equal to the batch size, a more informative error checking is performed, where each problem

has its own error code. The error code reflects the first error detected in the argument subset of a certain problem. An exception is thrown if any error is detected. The user can inspect the `info` vector by inserting the routine call in a `try-catch` statement.

As an example, consider batch `GEMM` call for 4 multiplications, as summarized in Table 1.1. All operations share the same input matrix `A`, but have four different matrices for the `B` argument. Note that this is a use case that is not supported by any software library, except for Intel MKL. However, using MKL for the problem shown in Table 1.1 requires duplication of some arguments. Since the MKL group interface requires identical arguments per group, the shown problem must be divided into four different groups.

Imagine replacing some arguments with wrong values, which are shown in red in Table 1.1. Depending on the error mode selected, the user gets different error codes. If `info.size()` returns 1, then the user receives an exception due to the error in the `n` argument, with the value of `info` set to -4. If `info.size()` returns 4, then an error code is generated for each problem, and the `info` vector, upon exit, will have the values $\{0, -4, -10, 0\}$.

1.5 Summary

This chapter shed the light on some important features of the C++ API of batch BLAS. API unification, semantic correctness, and error checking are features with the most influence on the implementation, which is thoroughly discussed in Chapter 2.

CHAPTER 2

Implementation Details

This chapter introduces details about the implementation of the C++ API for batch BLAS. It focuses on the support for GPU accelerators, since they are the key to performance on today's heterogeneous supercomputers. In addition, the SLATE library is designed to use batch BLAS (i.e. batch GEMM) on GPUs for the compute-intensive rank updates [2]. The extension for group-based API or stride-based APIs [1] is not discussed.

2.1 GPU Abstraction Layer

The Batch BLAS++ is intended to support both host CPUs and hardware accelerators. Two backends are considered for GPU support, namely NVIDIA's cuBLAS library[1], and AMD's rocBLAS library[2]. Both libraries have a common structure, in terms of handles, execution queues (i.e. streams), events, and others. The Batch BLAS++ will eventually abstract both libraries, including data types and the BLAS routines provided by both. Currently, only the cuBLAS backend is implemented. As an example, the following code shows the abstraction of some cuBLAS (and eventually rocBLAS) data types.

```
1 #ifdef HAVE_CUBLAS
2
3 // types
4 typedef    int                 device_blas_int;
5 typedef    cudaError_t         device_error_t;
6 typedef    cublasStatus_t      device_blas_status_t;
7 typedef    cublasHandle_t      device_blas_handle_t;
8 typedef    cublasOperation_t   device_trans_t;
9 typedef    cublasDiagType_t    device_diag_t;
```

[1] https://developer.nvidia.com/cublas
[2] https://github.com/ROCmSoftwarePlatform/rocBLAS

```
10 typedef    cublasFillMode_t     device_uplo_t;
11 typedef    cublasSideMode_t     device_side_t;
12
13 // constants
14 #define    DevSuccess         cudaSuccess
15 #define    DevBlasSuccess     CUBLAS_STATUS_SUCCESS
16 #define    DevNoTrans         CUBLAS_OP_N
17 #define    DevTrans           CUBLAS_OP_T
18 #define    DevConjTrans       CUBLAS_OP_C
19 #define    DevDiagUnit        CUBLAS_DIAG_UNIT
20 #define    DevDiagNonUnit     CUBLAS_DIAG_NON_UNIT
21 #define    DevUploUpper       CUBLAS_FILL_MODE_UPPER
22 #define    DevUploLower       CUBLAS_FILL_MODE_LOWER
23 #define    DevSideLeft        CUBLAS_SIDE_LEFT
24 #define    DevSideRight       CUBLAS_SIDE_RIGHT
25
26 #elif defined(HAVE_ROCBLAS)
27
28 /* define rocBLAS types and constants */
29
30 #endif
```

Another important abstraction is GPU errors, which can be categorized into runtime errors and BLAS errors. The GPU abstraction layer in Batch BLAS++ abstracts both types of errors. The macros `device_error_check` and `device_blas_check` throw exceptions to the user if an error is encountered on the runtime/BLAS side, respectively.

```
1  namespace blas{
2
3  inline
4  bool is_device_error(device_error_t error){ return (error != DevSuccess); }
5
6  inline
7  bool is_device_error(device_blas_status_t status){ return (status != DevBlasSuccess); }
8
9  // blaspp aborts on device errors
10 #define device_error_check( error ) \
11     do{ device_error_t e = error; \
12         blas::internal::abort_if( blas::is_device_error(e), \
13         __func__, "%s", blas::device_error_string( e ) ); } while(0)
14
15 // blaspp aborts on device blas errors
16 #define device_blas_check( status )  \
17     do{ device_blas_status_t s = status; \
18         blas::internal::abort_if( blas::is_device_error(s), \
19         __func__, "%s", blas::device_error_string( s ) ) ; } while(0)
20
21 }    // namespace blas
```

Following the abstraction of types, constants, and errors, some utility functions can also be abstracted, such as conversion of constants and memory management.

```
1  namespace blas {
2
3  // conversion from BLAS++ constants to device constants
4  inline
5  device_trans_t    blas::device_trans_const(blas::Op trans){
6      blas_error_if( trans != Op::NoTrans &&
7                     trans != Op::Trans &&
8                     trans != Op::ConjTrans );
9
10     device_trans_t trans_ = DevNoTrans;
11     switch(trans)
12     {
```

```
        case Op::NoTrans:   trans_ = DevNoTrans;   break;
        case Op::Trans:     trans_ = DevTrans;     break;
        case Op::ConjTrans: trans_ = DevConjTrans; break;
        default:;
    }
    return trans_;
}

// memory allocation
template<typename T>
inline
T* blas::device_malloc(int64_t nelements){
    T* ptr = NULL;
    #ifdef HAVE_CUBLAS
    device_error_check( cudaMalloc((void**)&ptr, nelements * sizeof(T)) );
    #elif defined(HAVE_ROCBLAS)
    /* allocation on AMD GPUs */
    #endif
    return ptr;
}

// free a device pointer
inline
void blas::device_free(void* ptr){
    #ifdef HAVE_CUBLAS
    device_error_check( cudaFree( ptr ) );
    #elif defined(HAVE_ROCBLAS)
    /* free memory on AMD GPUs */
    #endif
}

} // namespace blas
```

While data transfers are possible through memory copy functions (e.g. cudaMemcpy), there are sophisticated functions that are more useful for dense linear algebra. Such functions are part of the vendor BLAS library, rather than the GPU runtime.

```
namespace blas {

// copy matrix to GPU
template<typename T>
inline
void blas::device_setmatrix(
                int64_t m, int64_t n,
                T* hostPtr, int64_t ldh,
                T* devPtr,  int64_t ldd, Queue &queue){
    #ifdef HAVE_CUBLAS
    device_blas_check( cublasSetMatrixAsync(
                       (device_blas_int)m,    (device_blas_int)n, (device_blas_int)sizeof(T),
                       (const void *)hostPtr, (device_blas_int)ldh,
                       (      void *)devPtr,  (device_blas_int)ldd, queue.stream() ) );
    #elif defined(HAVE_ROCBLAS)
    /* call rocBLAS set matrix routine */
    #endif
}

// copy matrix from GPU
template<typename T>
inline
void blas::device_getmatrix(
                int64_t m, int64_t n,
                T* devPtr,  int64_t ldd,
                T* hostPtr, int64_t ldh, Queue &queue){
    #ifdef HAVE_CUBLAS
    device_blas_check( cublasGetMatrixAsync(
```

```
                        (device_blas_int)m,    (device_blas_int)n, (device_blas_int)sizeof(T),
                        (const void *)devPtr,  (device_blas_int)ldd,
                        (      void *)hostPtr, (device_blas_int)ldh, queue.stream() ) );
    #elif defined(HAVE_ROCBLAS)
    /* call rocBLAS get matrix routine */
    #endif
}

} // namespace blas
```

Finally, the GPU abstraction layer provides wrappers for all the BLAS routines that are supported by the vendor BLAS library. The APIs of such wrappers are compliant with the BLAS++ interface [3], except that they accept a `blas::Queue` object to indicate that this is an accelerator interface. As an example, the `DGEMM` interface for GPUs looks like:

```
namespace blas{

inline
void gemm(
    blas::Layout layout,
    blas::Op transA, blas::Op transB,
    int64_t m, int64_t n, int64_t k,
    double alpha,
    double const *dA, int64_t ldda,
    double const *dB, int64_t lddb,
    double beta,
    double       *dC, int64_t lddc,
    blas::Queue &queue );

}
```

All level-3 BLAS routines have C++ interfaces. These APIs are used to provide an reference implementation for the use cases that are not directly supported by the vendor's batch routines.

2.2 The blas::Queue Class

The `Queue` class provides an execution context for launching kernels on GPU accelerators. It hides the complexity of setting up the proper environment for executing computational workloads, GPU synchronization, calling the vendor BLAS routines, and others. In particular, the current implementation of the `Queue` class provides the following functionalities:

(1) A seamless selection of GPUs on multi-GPU systems. Each `Queue` object is simply bound to a certain GPU.

(2) Setting up the vendor BLAS library, so that the user can directly call the C++ wrappers for a given routine. For example, with a cuBLAS backend, the `Queue` class is responsible for managing the cuBLAS handle, which is needed for most cuBLAS routines.

(3) Every `Queue` object has a distinct execution queue (e.g. CUDA stream for NVIDIA GPUs). The object is responsible for creating, destroying, and launching kernels into the execution queeu.

(4) The `Queue` class also provides the functionality of GPU synchronization, so that the host CPU waits for the completion for all the workloads submitted into a given `Queue` object.

Eventually, the Queue will also support more fine-grain synchronization against certain tasks that are submitted to a given Queue object.

(5) A workspace on the GPU memory for the setup of batch routines is also provided. The C++ API for batch BLAS uses std::vector containers for scalar and array arguments. Such vectors are stored in the CPU memory. In order to call the vendor-supplied batch BLAS routines, array argument must be stored in the GPU memory. The blas::Queue class allows the user, through a constructor function, to set up a GPU workspace of certain size. Such workspace is used underneath to copy the array arguments, if required.

The following code provides a basic implementation for the Queue class using CUDA (and cuBLAS) as a backend.

```
namespace blas {

class Queue
{
public:
    void** devPtrArray;
    void** hostPtrArray;

    Queue(){
        blas::get_device( &device_ );
        device_error_check( cudaStreamCreate(&stream_) );
        device_blas_check( cublasCreate(&handle_) );
        device_blas_check( cublasSetStream( handle_, stream_ ) );
    }

    Queue(blas::Device device, int64_t batch_size){
        device_ = device;
        blas::set_device( device_ );
        hostPtrArray = blas::device_malloc_pinned<void*>( 3 * batch_size );
        devPtrArray  = blas::device_malloc<void*>( 3 * batch_size );

        device_error_check( cudaStreamCreate(&stream_) );
        device_blas_check( cublasCreate(&handle_) );
        device_blas_check( cublasSetStream( handle_, stream_ ) );
    }

    blas::Device          device() { return device_;   }
    device_blas_handle_t  handle() { return handle_;   }

    void sync(){
        device_error_check( cudaStreamSynchronize(this->stream()) );
    }

    ~Queue(){
        blas::device_free( devPtrArray );
        blas::device_free_pinned( hostPtrArray );

        device_blas_check( cublasDestroy(handle_) );
        device_error_check( cudaStreamDestroy(stream_) );
    }

private:
    blas::Device          device_;      // associated device ID
    device_blas_handle_t  handle_;      // associated device blas handle
    cudaStream_t          stream_;      // associated CUDA stream; may be NULL
};

}        //  namespace blas
```

2.3 Supporting Different Use Cases

The proposed C++ API supports significantly more use cases than what the vendor-supplied routines provides. Most vendor libraries support only batches of fixed sizes. The C++ API inspects the argument to see if there is a match between the input arguments and the signature of the vendor's batch routine. If a match is found, the C++ routine calls the wrapper of the vendor's batch routine. As an example, the C++ wrapper of the batch `DGEMM` routine looks like:

```
void DEVICE_BATCH_dgemm(
    device_blas_handle_t handle,
    device_trans_t transA, device_trans_t transB,
    device_blas_int m, device_blas_int n, device_blas_int k,
    double alpha,
    double const * const * dAarray, device_blas_int ldda,
    double const * const * dBarray, device_blas_int lddb,
    double beta,
    double** dCarray, device_blas_int lddc,
    device_blas_int batch_size)
{
    #ifdef HAVE_CUBLAS
    cublasDgemmBatched( handle, transA, transB,
                        m, n, k,
                        &alpha, (const double**)dAarray, ldda, (const double**)dBarray, lddb,
                        &beta,                  dCarray, lddc, batch_size );
    #elif defined(HAVE_ROCBLAS)
    // TODO: call rocBLAS
    #endif
}
```

If a match is not found, then the C++ routines switches to a reference implementation that calls the vendor's regular BLAS routine inside a `for` loop. The current implementation loops sequentially over the operations in the batch, and, for each problem, submits a GPU kernel to the input queue. A future enhancement to this implementation includes launching GPU kernels into multiple concurrent queues.

2.4 A Complete Example for Batch GEMM

The following example shows a complete implementation of the batch `DGEMM` routine. The implementation covers all the possible combinations of the input arguments.

```
inline
void gemm(
    std::vector<blas::Op> const &transA,
    std::vector<blas::Op> const &transB,
    std::vector<int64_t>  const &m,
    std::vector<int64_t>  const &n,
    std::vector<int64_t>  const &k,
    std::vector<double >  const &alpha,
    std::vector<double*>  const &Aarray, std::vector<int64_t>  const &ldda,
    std::vector<double*>  const &Barray, std::vector<int64_t>  const &lddb,
    std::vector<double >  const &beta,
    std::vector<double*>  const &Carray, std::vector<int64_t> const &lddc,
    const size_t batch,                  std::vector<int64_t>       &info,
    blas::Queue &queue )
{

    blas_error_if( batch < 0 );
```

```
18      blas_error_if( !(info.size() == 0 || info.size() == 1 || info.size() == batch) );
19      if(info.size() > 0){
20          // perform error checking
21          blas::batch::gemm_check<double>( transA, transB,
22                                          m, n, k,
23                                          alpha, Aarray, ldda,
24                                                 Barray, lddb,
25                                          beta,  Carray, lddc,
26                                          batch, info );
27      }
28      bool fixed_size =   ( transA.size() == 1     &&
29                            transB.size() == 1     &&
30                            m.size()      == 1     &&
31                            n.size()      == 1     &&
32                            k.size()      == 1     &&
33                            alpha.size()  == 1     &&
34                            Aarray.size() == batch &&
35                            ldda.size()   == 1     &&
36                            Barray.size() == batch &&
37                            lddb.size()   == 1     &&
38                            beta.size()   == 1     &&
39                            Carray.size() == batch &&
40                            lddc.size()   == 1 );
41
42      blas::set_device( queue.device() );
43      if( fixed_size ) {
44          // call the vendor routine
45          device_trans_t  transA_ = blas::device_trans_const( transA[0] );
46          device_trans_t  transB_ = blas::device_trans_const( transB[0] );
47          device_blas_int m_      = (device_blas_int) m[0];
48          device_blas_int n_      = (device_blas_int) n[0];
49          device_blas_int k_      = (device_blas_int) k[0];
50          device_blas_int ldda_   = (device_blas_int) ldda[0];
51          device_blas_int lddb_   = (device_blas_int) lddb[0];
52          device_blas_int lddc_   = (device_blas_int) lddc[0];
53
54          // copy Aarray, Barray, and Carray to device
55          double **dAarray, **dBarray, **dCarray;
56          dAarray = (double**)queue.devPtrArray;
57          dBarray = dAarray + batch;
58          dCarray = dBarray + batch;
59          device_setvector<double*>(batch, (double**)&Aarray[0], 1, dAarray, 1, queue);
60          device_setvector<double*>(batch, (double**)&Barray[0], 1, dBarray, 1, queue);
61          device_setvector<double*>(batch, (double**)&Carray[0], 1, dCarray, 1, queue);
62          DEVICE_BATCH_dgemm( queue.handle(),
63                              transA_, transB_,
64                              m_, n_, k_,
65                              alpha[0], dAarray, ldda_, dBarray, lddb_,
66                              beta[0],  dCarray, lddc_,
67                              batch);
68      }
69      else{
70          for(size_t i = 0; i < batch; i++){
71              Op transA_    = blas::batch::extract<Op>(transA, i);
72              Op transB_    = blas::batch::extract<Op>(transB, i);
73              int64_t m_    = blas::batch::extract<int64_t>(m, i);
74              int64_t n_    = blas::batch::extract<int64_t>(n, i);
75              int64_t k_    = blas::batch::extract<int64_t>(k, i);
76              int64_t lda_  = blas::batch::extract<int64_t>(ldda, i);
77              int64_t ldb_  = blas::batch::extract<int64_t>(lddb, i);
78              int64_t ldc_  = blas::batch::extract<int64_t>(lddc, i);
79              double alpha_ = blas::batch::extract<double>(alpha, i);
80              double beta_  = blas::batch::extract<double>(beta, i);
81              double* dA_   = blas::batch::extract<double*>(Aarray, i);
82              double* dB_   = blas::batch::extract<double*>(Barray, i);
83              double* dC_   = blas::batch::extract<double*>(Carray, i);
```

```
84             blas::gemm(
85                 Layout::ColMajor, transA_, transB_, m_, n_, k_,
86                 alpha_, dA_, lda_,
87                         dB_, ldb_,
88                 beta_,  dC_, ldc_, queue );
89         }
90     }
91 }
```

The routine starts with two trivial checks of the batch size and the `info` vector size. The latter is very important to verify that the user has selected a valid size that matches one of the available error checking modes.

The second step is to perform the error checking if `info.size()` returns a non-zero value. The `gemm_check` performs the required BLAS error checking. It also verifies the integrity of the input vector sizes. The third stage is to decide whether there is a match between the input use case and the vendor's batch routine. As mentioned before, all GPU vendor libraries support only fixed size batches. The flag `fixed_size` tests such a match. If it is true, then the C++ routine directly invokes the vendor's batch routine. Note that all the pointer arrays have to copied from the CPU memory space to the GPU memory space. This is where the workspace inside the `Queue` object becomes very useful. If the flag `fixed_size` is false, then the reference implementation is invoked. Note that the loop heavily relies on the `extract` function, which reads the input arguments for each individual operation, after which the non-batch `blas::gemm` is called.

2.5 Summary

This chapter presented the implementation details of the C++ Batch BLAS. The chapter focuses on batch matrix multiplication as an example. It shows an implementation of a software stack that abstracts the hardware details at the very bottom, wraps necessary utility functions and regular BLAS routines, and then moves up to provide a very flexible API for batch BLAS at the top of the stack. Considering performance, the C++ implementation has the ability to call optimized vendor routines if the input use case is vendor-supported. Otherwise, a reference implementation is provided to maintain a complete coverage of all the possible use cases of a batch routine. Future improvements on the vendor size, regarding either performance optimization or wider coverage, can be reflected very easily and transparently without any change to the APIs exposed to the user.

CHAPTER 3

Performance Numbers

In this chapter, we present the performance of selected C++ batch routines with respect to the overheads associated with maintaining a very flexible interface.

3.1 Analysis of Overheads

In general, there are two categories of overheads for the implemented C++ API:

(1) **Permanent Overhead**. This overhead is mainly associated with hardware accelerators, where a memory copy is required from the host memory space to the device memory space. Since the C++ API uses `std::vector` containers, which reside in the CPU memory space, a device memory copy is required for all array arguments of the batch routines. As of today's vendor libraries, this corresponds to the pointer arrays of the data arguments (e.g. matrices and vectors). Such source of overhead exist in all use cases. It can be avoided in the future by adding a thorough support for the `std::vector` containers in the device runtime. In such a case, a vector can be directly initialized in the GPU memory space, and no copy is required.

(2) **Controllable Overhead**. This source of overhead is associated with error checking, which is not trivial in batch BLAS, especially with an API that support numerous use cases. The overhead can be totally avoided by turning error checking off. If error checking is turned on, then the overall performance of the batch routine depends on whether the check is set to be argument-wise or problem-wise.

The following experiments illustrate the impact of such overheads on the performance of batch

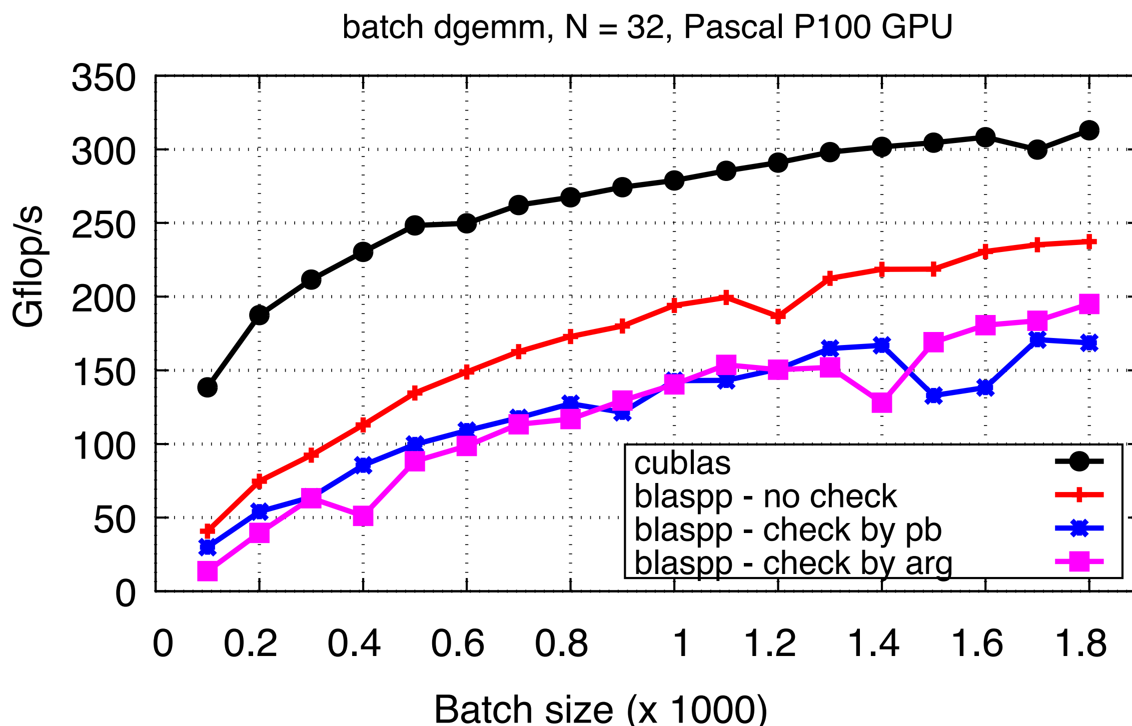

Figure 3.1: Performance analysis of batch DGEMM on square matrices of size 32

matrix multiplication in double precision DGEMM. Each experiment considers square matrices of a given size, and shows the observed performance while varying the size of the batch.

3.2 Small-to-medium Sizes

Figures 3.1 and 3.2 show the performance for batch DGEMM on square matrices of size 32 and 128, respectively. The first observation is that, even with the error checking turned off, calling cuBLAS natively is faster than calling the C++ API. The reason behind such behavior is permanent overhead of sending the pointer arrays from the `std::vector` containers on the CPU memory to the pointer arrays that are resident on the GPU memory. Such overhead is significant in Figure 3.1, because the matrix size is very small. However, even for small sizes, the communication overhead decreases consistently as the batch size grows. As an example, Figure 3.1 shows that cuBLAS is $3.4\times$ faster than BLAS++ with no error checking for a small batch of 100 operations. Such speedup shrinks significantly to $1.3\times$ for a batch of 1800. It is also clear that turning on any of the two error checking modes results in more overhead, and therefore, less performance.

Figure 3.2 shows that, as the size of the problem grows, all sources of overheads become less significant. The native cuBLAS call is only $1.10\times$-$1.16\times$ faster than BLAS++ with no error checking. The overhead can become almost negligible if the batch contain more operations. Another interesting observation is that the error checking is significantly faster when it configured to

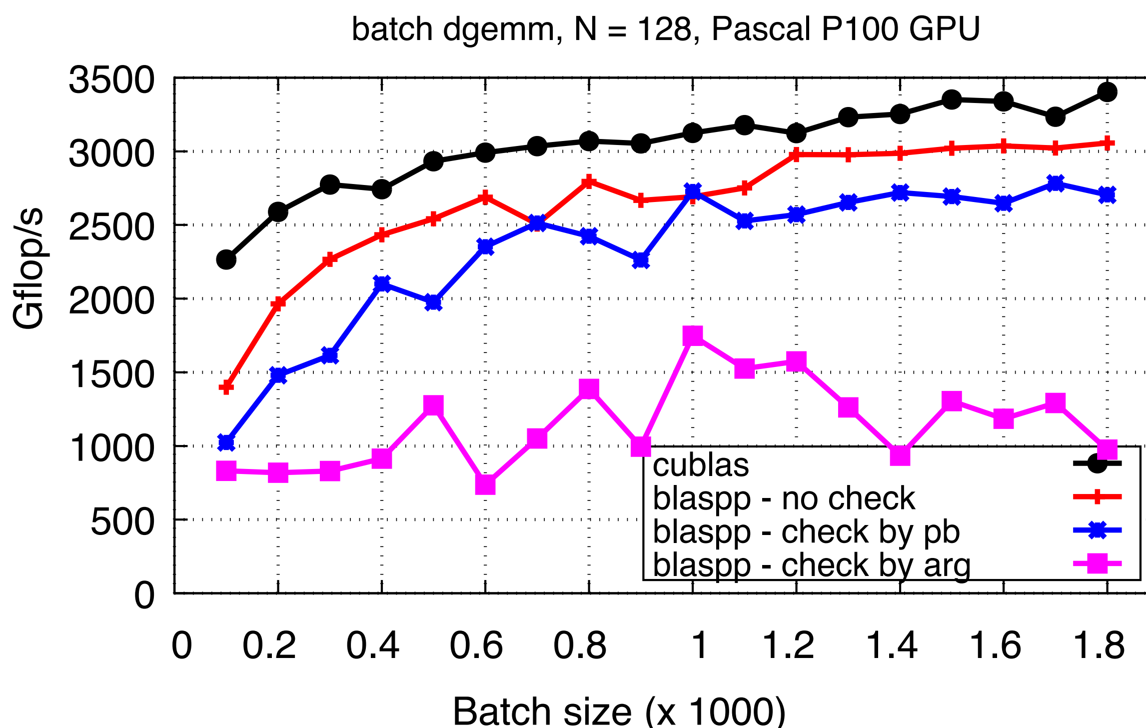

Figure 3.2: Performance analysis of batch DGEMM on square matrices of size 128

be problem-wise rather than argument-wise. The former contains more parallel work. Each operation is independently checked in parallel, and its respective entry in the `info` vector is set. On the other hand, the argument-wise error checking is costly because a reduction operation is required after checking every argument, unlike the problem-wise mode, which require exactly one reduction at the very end to test if any entry of `info` is non-zero.

3.3 Relatively Large Sizes

Figures 3.3 and 3.4 show more promising results for BLAS++. In general, the overheads are less significant, and the problem-wise error checking is always faster than the argument-wise error checking. For sizes 256×256, the performance of BLAS++ is almost identical to cuBLAS for large batches. More importantly, turning on error checking by problem is not a source of large overhead, as the performance stays close to its native upper-bound.

In Figure 3.4, calling BLAS++ with either no error checking or with problem-wise checking is almost identical to natively calling cuBLAS, which means that the overheads are negligible. The argument-wise error checking still suffers from performance drops that are mostly associated with the frequent reduction operations.

In typical use cases, it is recommended to turn on any of the error checking modes during development runs only to check for potential errors or development bugs. Production runs should always turn off any error checking in order to get the best performance out of BLAS++.

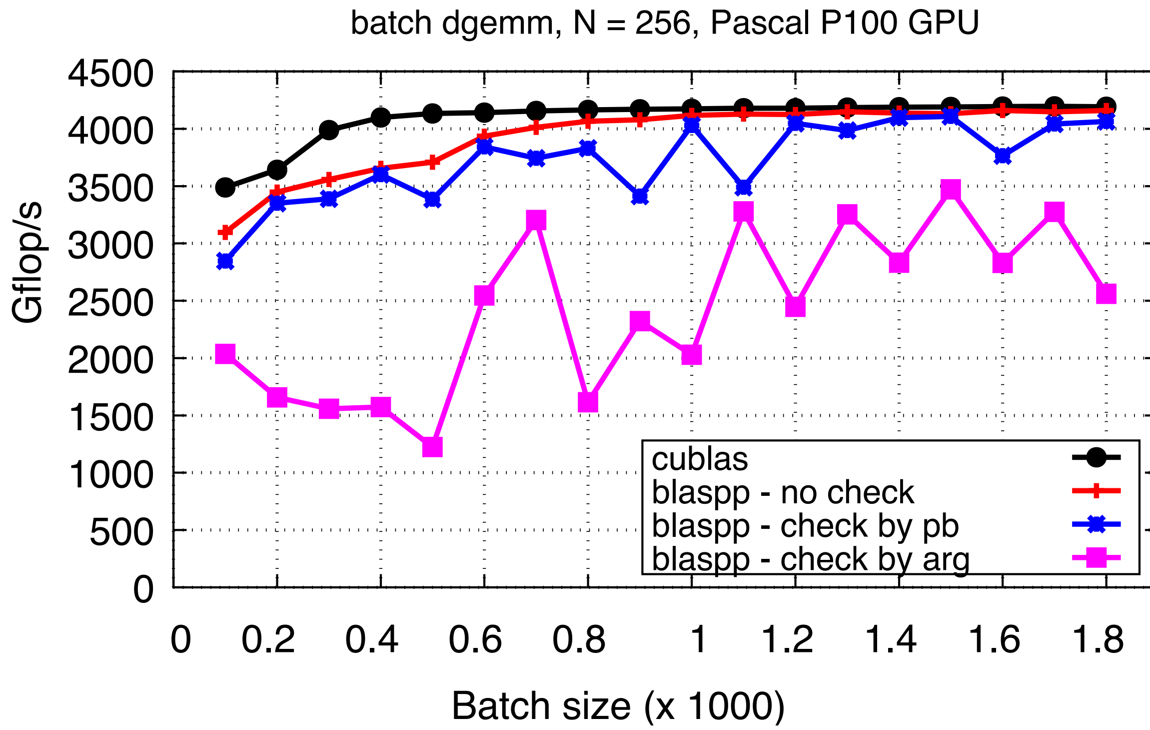

Figure 3.3: Performance analysis of batch DGEMM on square matrices of size 256

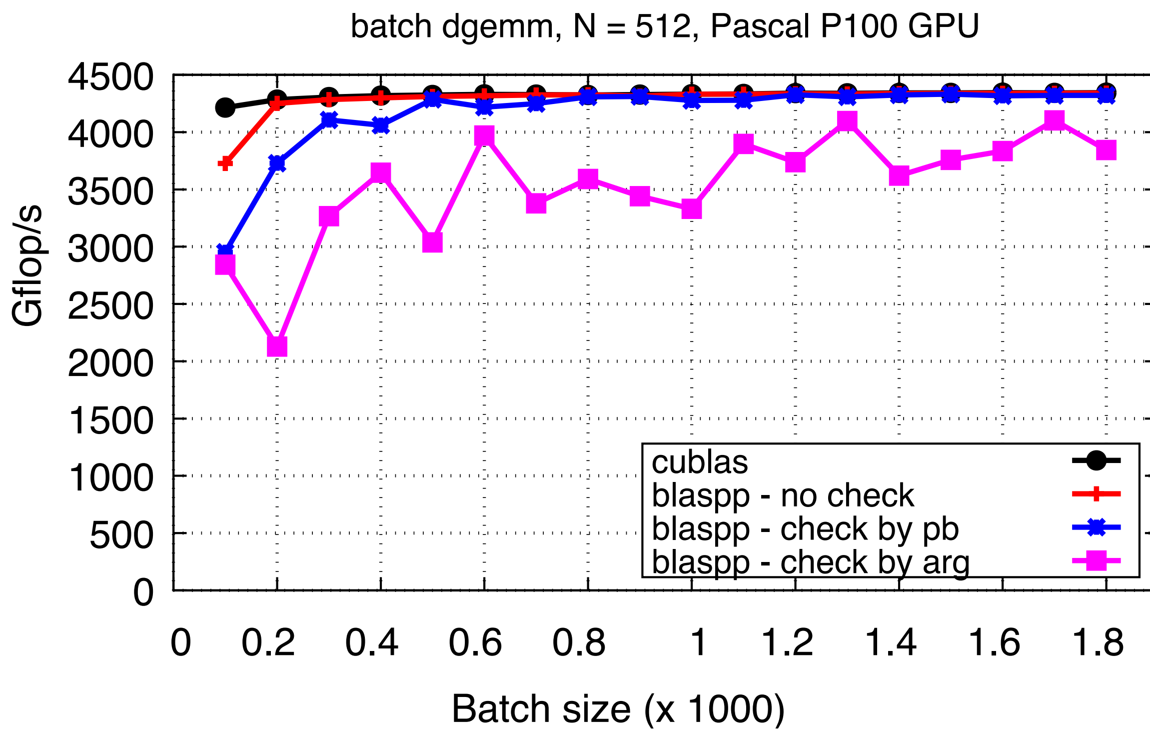

Figure 3.4: Performance analysis of batch DGEMM on square matrices of size 512

Recall that such reconfiguration is very easy in BLAS++. By resizing the `info` vector, the behavior of BLAS++ is controlled. The size of the `info` can be even configured as a runtime argument, so that no change to the developed code is required.

3.4 Summary

This chapter introduced some performance tests to Batch BLAS++, taking batch matrix multiplication as an example. In general, calling the C++ API is equal or slower than natively calling the vendor-supplied API. Relatively large problems encounter nearly negligible overheads, since most of the execution time is spent performing the actual operation. Small to medium size, however, suffer from some overheads that are associated with error checking, as well as the necessary host-to-device communication to set the array arguments of the batch API.

Bibliography

[1] Ahmad Abdelfattah, Konstantin Arturov, Cris Cecka, Jack Dongarra, Chip Freitag, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Panruo Wu. C++ API for Batch BLAS. Technical Report 4, ICL-UT-17-12, 12-2017 2017.

[2] Jakub Kurzak, Panruo Wu, Mark Gates, Ichitaro Yamazaki, Piotr Luszczek, Gerald Ragghianti, and Jack Dongarra. Designing SLATE: Software for Linear Algebra Targeting Exascale. SLATE Working Notes 3, ICL-UT-17-06, 10-2017 2017.

[3] Mark Gates, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, Konstantin Arturov, Cris Cecka, and Chip Freitag. C++ API for BLAS and LAPACK. Technical Report 2, ICL-UT-17-03, 06-2017 2017. revision 06-2017.