H2020-FETHPC-2014: GA 671633

NLAFET Working Note 5

A Comparison of Potential Interfaces for Batched BLAS Computations

Samuel D. Relton, Pedro Valero-Lara, and Mawussi Zounon

August, 2016

Document information

This preprint report is also published as MIMS EPrint 2016.xx, Manchester Institute for Mathematical Sciences School of Mathematics, The University of Manchester. ??

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

A Comparison of Potential Interfaces for Batched BLAS Computations*

Samuel D. Relton Pedro Valero-Lara Mawussi Zounon†

August 3, 2016

Abstract

One trend in modern high performance computing (HPC) is to decompose a large linear algebra problem into thousands of small problems which can be solved independently. There is a clear need for a batched BLAS standard, allowing users to perform thousands of small BLAS operations in parallel and making efficient use of their hardware. There are many possible ways in which the BLAS standard can be extended for batch operations. We discuss many of these possible designs, giving benefits and criticisms of each, along with a number of experiments designed to determine how the API may affect performance on modern HPC systems. Related issues that influence API design, such as the effect of memory layout on performance, are also discussed.

1 Introduction

The past few decades have seen a tremendous amount of effort expended in the design and implementation of efficient linear algebra software, primarily aiming to solve larger problems in less time. Numerous libraries have been written to take advantage of advances in computer architecture and exploit the parallelism both within a single node (using hardware accelerators), and between nodes (communicating using MPI, for example). Such libraries include Intel MKL, ATLAS [21], NVIDIA cuBLAS, the NAG Library, Chameleon, PLASMA [7], and MAGMA [2] etc.

One current trend in high performance computing is that, as opposed to solving one large linear algebra problem, one solves thousands of smaller subproblems, which are combined to form the solution of the larger problem. One example of this is given by multifrontal

*Version of August 3, 2016.

†School of Mathematics, The University of Manchester, Manchester, M13 9PL, UK (samuel.relton@manchester.ac.uk, pedro.valerolara@manchester.ac.uk, mawussi.zounon@manchester.ac.uk),
This work was funded by the European Unions Horizon 2020 research and innovation programme under the NLAFET grant agreement (671633).

solvers for sparse linear systems [11]. The libraries mentioned above do not target this situation specifically (though some initial support is available) and therefore these applications generally perform suboptimally.

The solution to this inefficiency is to develop a new standard set of routines for carrying out linear algebra operations on batches of small matrices, building on the well-known Basic Linear Algebra Subproblems (BLAS) standard [9], [8], [14]. The idea behind the Batched BLAS (BBLAS) is to perform multiple BLAS operations in parallel on many small matrices, making more efficient use of hardware than a simple for loop would allow. For example, if we consider a GEMM operation over a batch of N matrices then we would like to compute, in parallel,

$$C_i \leftarrow \alpha_i A_i B_i + \beta_i C_i, \qquad i = 1 : N. \tag{1.1}$$

We can keep the matrix sizes and the values of α_i and β_i constant throughout the entire batch or allow them to vary, depending upon the application that we have in mind.

One particular application which can benefit dramatically from performing many small matrix multiplications in parallel is deep learning: this functionality is already utilized in popular machine learning libraries such as Theano [4] and TensorFlow [1]. Other examples of such applications where the solution of many small problems are required include metabolic networks [13], astrophysics [16], the rendering of 3D graphics on the internet [12], and computational fluid dynamics [22].

A BBLAS standard could have a number of possible interfaces. A draft specification for BBLAS operations has already been implemented, but remains open to suggestions for change from the scientific community [10]. The primary goal of this work is to discuss three different approaches to design an API for BBLAS operations, giving some benefits and criticisms of each. The three approaches are discussed in section 2. For each approach we show the resulting calling sequence for three different BLAS routines, DGEMM, DGEMV, and DDOT to cover each level of BLAS operation. We perform a number of experiments designed to determine how the choice of API can affect the performance of the computation on modern hardware. As a secondary goal we also discuss some related issues which need to be considered during the API design, primarily the impact of the memory layout on hardware accelerators (e.g. GPUs) and the self-booting Xeon Phi models.

The remainder of this work is organized as follows. In section 2 we outline three different interfaces to BBLAS and give some example of their use. Then, in section 3, we discuss the relative benefits and criticisms of each approach in terms of, amongst other things, their ease of use in application areas. This section also contains some experiments to compare the overheads that the different APIs introduce to the computation. Section 4 contains some discussion and experiments on the effect that the memory layout has on the performance of BBLAS operations, which need to be considered when designing an efficient API. Finally, we give some concluding remarks in section 5.

Furthermore a repository containing sample code for each API that we discuss is available at https://github.com/sdrelton/bblas_api_test.

2 BBLAS APIs

In this section we outline three possible API designs for BBLAS operations. For each API we give some examples of the resulting calling sequence and some brief comments on their respective benefits. A detailed analysis of each approach is presented in section 3. The APIs are presented here in C code which, by default, stores matrices in row major order. It is a trivial modification to add support for column major order and to develop the corresponding Fortran APIs (see section 2.5).

Before discussing the APIs directly, we briefly explain a number of parameters that are common to the different APIs, but are not a part of the standard BLAS specification.

- `int batch_count` - This input parameter specifies the number of matrices in the batch that we are operating on. It is used by the first two APIs whilst the third takes a different approach.

- `int *info` - This output parameter allows the user to check whether the computation completed successfully once the routine exits. Whilst there is no info parameter in the standard BLAS it appears in the LAPACK routines.

Finally, in order to understand the thought process underpinning the design of each API, it is important that we discuss the two main categories of batch computation: fixed and variable sized batches.

2.1 Fixed and variable batches

When performing a BBLAS operation, the batch can be classified into two main types:

- a "fixed batch", where the matrix sizes and leading dimensions are constant throughout the entire batch, or,

- a "variable batch", where the matrix sizes and leading dimensions can vary for each matrix in the batch.

For example, if we wanted to compute a batch of three DGEMM operations in parallel where all the matrices A_i and B_i are of size 100×100 then we require a fixed batch computation. However, if A_1, A_2, and A_3 have different sizes then we would need to use a variable batch computation.

The reason for differentiating batch operations in this way is clear: using only one matrix size and leading dimension for the entire batch results in less overhead when performing error checks on the input parameters. In a variable batch operation we must check all of the (potentially thousands of) input values, as opposed to merely a few parameters for the fixed batch case.

Note that, in the types of batch computation above, the terms "fixed" and "variable" refer only to the size and leading dimension of the matrices. The other parameters of each BBLAS function can be treated separately in a similar manner: for instance in a batched

GEMM ($C_i \leftarrow \alpha A_i B_i + \beta C_i$) there are some applications where we may wish to vary α and β too. At the time of writing there is no consensus among the community on this matter, but the current draft standard for BBLAS computation proposes that:

- fixed batch computations will have all parameters fixed for the entire batch.
- variable batch computations will allow all parameters to vary over the entire batch.

This means that, if one wanted to compute a batch DGEMM with matrices of the same size but varying α and β parameters, then this would be classified as a variable batch operation.

To further complicate matters, we might also consider the situation where α is variable but β is fixed etc. Clearly it is difficult to support all possible combinations of parameters being fixed and variable: the API would become overly complicated and there would be a combinatorial explosion of highly optimized, low-level routines needed to deal with each case optimally.

The API designs that we investigate in more detail are:

- separate functions for fixed and variable batch computations.
- a flag-based API where the same function deals with both fixed and variable computation.
- a "group" API which performs multiple fixed batch computations within the same function call.

Note that, in the remainder of this section, all APIs use double pointers for the variables `arrayA`, `arrayB`, and `arrayC`. That is, we pass an array containing the address of each matrix in the batch. We call this the pointer-to-pointer (P2P) approach to batched BLAS. Other possible memory layouts are discussed in section 4.

2.2 Separate fixed and variable batch API

This API has two functions for each BLAS routine, one for fixed batch operations and another for variable batch operations. The functions are named by prefixing either `batchf_` or `batchv_`—for fixed and variable batches, respectively—to the name of the standard BLAS function.

For the fixed batch functions all of the parameters, except for the arrays of matrices, are constant for the entire batch; meanwhile all parameters (except for `batch_count`) are arrays in the variable batch version. An example for fixed and variable batch DGEMM is given in Table 2.1.

Another main difference between these two versions is in the `info` parameter, which affects the way that errors are handled. For the fixed batch version we propose to have just one `info` for the entire batch and, if some incorrect parameters are detected, the entire batch computation is halted. On the other hand, in the variable batch case, if the parameters for one subproblem are wrong the other subproblems can still continue and hence we propose

Table 2.1: Fixed (left) and variable (right) DGEMM function prototypes using the separate fixed and variable batch API.

```
batchf_dgemm(
const enum CBLAS_TRANSPOSE transA,
const enum CBLAS_TRANSPOSE transB,
const int m, const int n,
const int k,
const double alpha,
double const *const *arrayA,
const int lda,
double const *const *arrayB,
const int ldb,
const double beta,
double **arrayC,
const int ldc,
const int batch_count, int *info
);
```

```
batchv_dgemm(
const enum CBLAS_TRANSPOSE *transA,
const enum CBLAS_TRANSPOSE *transB,
const int *m, const int *n,
const int *k,
const double *alpha,
double const *const *arrayA,
const int *lda,
double const *const *arrayB,
const int *ldb,
const double *beta,
double **arrayC,
const int *ldc,
const int batch_count, int *info
);
```

that `info` is an array of length `batch_count`. The community is still undecided on the `info` parameter and error handling behaviour; one solution for the latter issue may be to have a `xerbla_batch` function, similar to `xerbla` for the standard BLAS, which can be modified by users and vendors as they see fit.

In Tables 2.2 and 2.3 we show the corresponding batch versions of the DGEMV and DDOT routines, respectively. These are largely as expected, though the level-1 BLAS routine DDOT has the extra parameter `res` to store the result: the standard level-1 BLAS routine returns the result directly but—in the batch case—we would need to return a pointer to an array of results, which could be considered poor programming practice. Therefore we add this output as a parameter to the function call, which is the same approach taken by the complex and double complex precision versions (CDOTU and ZDOTU etc., respectively) in CBLAS.

2.3 Flag-based API

The flag-based API uses only one function to handle both fixed and variable batches. The functions are named by prefixing `batch_` to the standard BLAS function name. The calling sequence is almost identical to the separate fixed and variable API, except for the addition of an extra parameter to specify the batch type. This can be defined in a header file as follows.

```
enum BBLAS_OPTS = {BBLAS_FIXED=0, BBLAS_VARIABLE=1};
```

In the fixed batch case, the variables `m`, `n`, `k`, etc. are passed by address, so that the same interface can be used for the variable batch case. This could be a minor annoyance

Table 2.2: Fixed (left) and variable (right) DGEMV function prototypes using the separate fixed and variable batch API.

```
batchf_dgemv(
const enum CBLAS_TRANSPOSE trans,
const int m, const int n,
const double alpha,
double const *const *arrayA,
const int lda,
double const *const *arrayx,
const int incx,
const double beta,
double **arrayy, const int incy,
const int batch_count, int *info
);
```

```
batchv_dgemv(
const enum CBLAS_TRANSPOSE *trans,
const int *m, const int *n,
const double *alpha,
double const *const *arrayA,
const int *lda,
double const *const *arrayx,
const int *incx,
const double *beta,
double **arrayy, const int *incy,
const int batch_count, int *info
);
```

Table 2.3: Fixed (left) and variable (right) DDOT function prototypes using the separate fixed and variable batch API.

```
batchf_ddot(
const int n,
double const *const *x,
const int incx,
double const *const *y,
const int incy,
double *res,
const int batch_count, int *info
);
```

```
batchv_ddot(
const int *n,
double const *const *x,
const int *incx,
double const *const *y,
const int *incy,
double *res,
const int batch_count, int *info
);
```

Table 2.4: DGEMM function prototype using the flag-based API.

```
batch_dgemm(
const enum CBLAS_TRANSPOSE *transA,
const enum CBLAS_TRANSPOSE *transB,
const int *m, const int *n, const int *k,
const double *alpha,
double const *const *arrayA, const int *lda,
double const *const *arrayB, const int *ldb,
const double *beta,
double **arrayC, const int *ldc,
const int batch_count, const enum BBLAS_OPTS batch_opts,
int *info
);
```

Table 2.5: DGEMV function prototype using the flag-based API.

```
batch_dgemv(
const enum CBLAS_TRANSPOSE *trans,
const int *m, const int *n,
const double *alpha,
double const *const *arrayA,
const int *lda,
double const *const *arrayx,
const int *incx,
const double *beta,
double **arrayy, const int *incy,
const int batch_count,
const enum BBLAS_OPTS batch_opts,
int *info
);
```

Table 2.6: DDOT function prototype using the flag-based API.

```
batch_ddot(
const int *n,
double const *const *x,
const int *incx,
double const *const *y,
const int *incy,
double *res,
const int batch_count,
const enum BBLAS_OPTS batch_opts,
int *info
);
```

Table 2.7: DGEMM function prototype using the group API.

```
batchg_dgemm(
const enum CBLAS_TRANSPOSE *transA,
const enum CBLAS_TRANSPOSE *transB,
const int *m, const int *n, const int *k,
const double *alpha,
double const *const *arrayA, const int *lda,
double const *const *arrayB, const int *ldb,
const double *beta,
double **arrayC, const int *ldc,
const int group_count, const int *group_size,
int *info
);
```

if only fixed case batches are required in a given application, since one would continually need to pass pointers such as `&m` and `&n` instead of passing the values directly. However, for the variable batch operations, the calling sequence is almost identical to the previous API. Tables 2.4, 2.5, and 2.6 show the resulting calling sequences for batch versions of DGEMM, DGEMV, and DDOT, respectively.

2.4 Group API

The final API design that we consider is the group API. This lies somewhere between the two other APIs, by which we mean that it is primarily designed to compute multiple fixed batch operations simultaneously as opposed to a single fixed or variable batch. However, it is still possible to compute a purely fixed or variable batch when required.

Each fixed batch that we want to operate on is called a group and each group can have completely different parameters, from the size of the matrices to the number of subproblems in the group. Therefore a variable batch computation is equivalent to having multiple groups of size one.

The functions are named by prefixing `batchg_` to the standard BLAS routines. The calling sequence of the functions is slightly different to the other APIs, in that `batch_count` and `batch_opts` are replaced by `group_count` and `group_size`, respectively. The `group_count` parameters gives the number of groups whilst the elements of the array `group_size` store the size of each group.

In our implementation the `info` parameter is an array of length `group_count`, with one return value for each group. However, the community has not yet reached a consensus on this matter and it may be preferable to set the length of the `info` array equal to the total number of subproblems.

Tables 2.7, 2.8, and 2.9 show the resulting function prototypes for batch versions of DGEMM, DGEMV, and DDOT, respectively.

Table 2.8: DGEMV function prototype using the group API.

```
batchg_dgemv(
const enum CBLAS_TRANSPOSE *trans,
const int *m, const int *n,
const double *alpha,
double const *const *arrayA,
const int *lda,
double const *const *arrayx,
const int *incx,
const double *beta,
double **arrayy, const int *incy,
const int group_count,
const int *group_size,
int *info
);
```

Table 2.9: DDOT function prototype using the group API.

```
batchg_ddot(
const int *n,
double const *const *x,
const int *incx,
double const *const *y,
const int *incy,
double *res,
const int group_count,
const int *group_size,
int *info
);
```

Table 2.10: Fortran interface for a fixed batch DGEMM utilizing the separate fixed and variable batch API.

```
SUBROUTINE
batchf_dgemm(
transA, transB,
m, n, k,
alpha,
arrayA, lda,
arrayB, ldb,
beta,
arrayC, ldc,
batch_count, info)
character, intent(in) :: transA
character, intent(in) :: transB
integer, intent(in)   :: m
integer, intent(in)   :: n
integer, intent(in)   :: k
double precision, intent(in) :: alpha
integer(kind=c_intptr_t), intent(in) :: arrayA(*)
integer, intent(in)   :: lda
integer(kind=c_intptr_t), intent(in) :: arrayB(*)
integer, intent(in)   :: ldb
double precision, intent(in) :: beta
integer(kind=c_intptr_t), intent(inout) :: arrayC(*)
integer, intent(in)   :: ldc
integer, intent(in)   :: batch_count
integer, intent(out)  :: info
END SUBROUTINE
```

2.5 Fortran APIs

Each of the APIs discussed within this section can be readily converted into Fortran. In Table 2.10 we show an example of how one might call a fixed batch DGEMM using the separate fixed and variable API discussed in section 2.2. It is clear that all the other APIs can be readily converted into Fortran without any difficulties.

3 Critical analysis of the APIs

Each API described in the previous section has its own strengths and weaknesses. At the time of writing Intel MKL supports the group API for batched GEMM computation. Meanwhile NVIDIA cuBLAS provides functionality for fixed batch versions of GEMM and TRSM,

along with batched LAPACK functionality for solving linear systems using LU and QR factorizations, plus a batched least squares solver [5].

One of the major differences between the proposed APIs is their complexity. It appears that the separate fixed and variable API is the simplest from the user's perspective: there is a clear distinction between the two types of batch computation. This API also avoids the need to remember that, when using the flag-based or group APIs, only the first element of the arrays `m`, `n`, etc. are required. It may be rather frustrating, from the user's perspective, to pass all the parameters by address when a fixed batch operation is requested using the other APIs. It is also worth noting that most of the applications of BBLAS identified so far rely primarily on fixed batch computations and this API makes those calls simpler for the user.

On the other hand, the flag-based and group APIs add batch functionality to the standard BLAS using just one function per BLAS routine, as opposed to two. The price paid for this compactness is the additional complexity in the calling sequences. The flag-based API uses an extra parameter to determine the batch type whilst the group API uses the `group_count` and `group_size` parameters. Requiring the user to set these extra parameters correctly increases the chance of errors occurring.

The main reason that one might prefer the group API is that, compared with running multiple fixed batches, we might expect to obtain a smaller runtime by avoiding overhead costs. The next subsection contains experiments designed to show the size of these overheads.

However, one major criticism of the group API is the additional user effort and memory requirements involved for a variable batch computation. To perform a variable batch computation one must set `group_count` equal to the number of subproblems and create the array `group_size` with all elements equal to one. The time to allocate this array and loop through it, setting all the elements equal to one, can be significant. For an extremely large batch this can also utilize a large amount of memory unnecessarily.

Furthermore, it is unclear if there are any applications where the group API appears naturally. Consider a multifrontal solver [11] where, at some stage of the computation, we have many small matrices (of size 2×2 and 3×3, for example,) which form two groups. In order to actually use these two groups we would need to loop over each matrix, inspect their size, and collate them into the two groups before calling the relevant BBLAS (or batched LAPACK) routine. Looping over the individual matrices in this way and inspecting their size can cause a significant overhead in comparison to treating the set of matrices as a variable type batch and starting the computation immediately using one of the alternative APIs.

3.1 Experiments using the Group API

In this subsection we perform a number of experiments in order to clarify what benefit might be gained from adopting the Group API as opposed to making multiple calls to the fixed batch API, assuming that the matrices have already been allocated into groups (see the discussion in the previous paragraph). The experiments are based upon those shown by Sarah

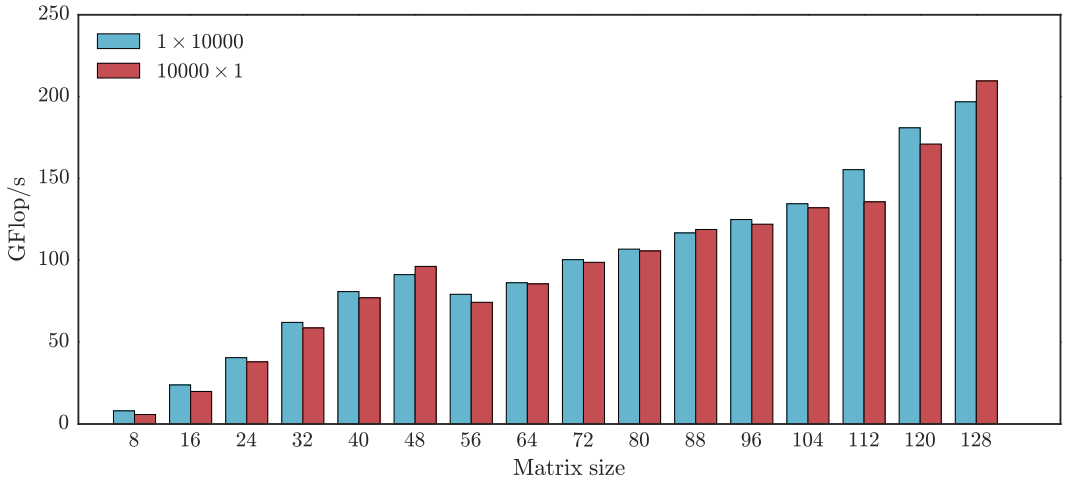

Figure 3.1: Performance of the MKL batched DGEMM on batches of 10000 matrices using either one group of 10000 (1×10000) or 10000 groups of one (10000×1). All the matrices are square with the size denoted on the x-axis.

Knepper (Intel) at the recent workshop on BBLAS, held at the University of Tennessee[1].

All experiments in this section are run on a NUMA node with 2 sockets, using Intel Xeon CPU E5-2650 v3 chips (2.3GHz, Haswell architecture), for a total of 20 cores. Hyperthreading is enabled and the memory is interleaved between the two processors[2] and we create one OpenMP thread per core. All the matrices are chosen to have elements taken from a random uniform distribution on the interval $[0, 1]$. Note that we ensure the cache of each processor is flushed between every computation to avoid obtaining misleading performance results; by neglecting this step we can obtain performance results up to 4 times faster than those reported here in some cases when the data fits into cache memory. This is consistent with observations by Whaley and Castaldo [20].

Our first experiment in Figure 3.1 compares the difference in performance between running a batch DGEMM using one group of 10000 matrices versus using 10000 groups with one matrix each, for a variety of matrix sizes. The Intel MKL function `dgemm_batch` is used to perform both of these operations.

We see that the performance increases with the matrix size and that, generally, running the computation using just one group is slightly faster due to the reduced number of parameter checks. In most cases the difference in performance between the two routines is not very dramatic: the mean difference between the performance of the two approaches is 7.7%. We note that running 10000 groups containing one matrix each is essentially the worst possible case for the group API so that, with a less dramatic difference between the two groupings,

[1]PDF slides available at http://www.netlib.org/utk/people/JackDongarra/WEB-PAGES/Batched-BLAS-2016/. Accessed on 19th July 2016.

[2]Run with "`numactl --interleave=all`".

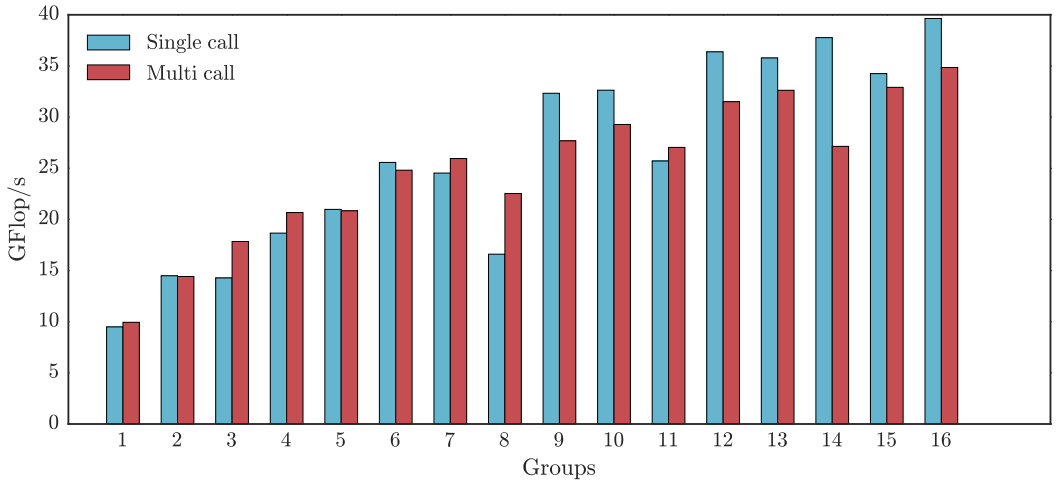

Figure 3.2: Performance of MKL batched DGEMM on groups of various sizes computed either as using multiple groups in a single call or using separate calls for each group. Group p contains ceil$(4096/p^3)$ matrices of size $8p \times 8p$.

we would expect the performance difference to be negligible.

Our second experiment in Figure 3.2 uses groups with varying matrix and group sizes to compare the performance using a single call to the DGEMM group interface as opposed to making a separate call for each group. We use up to 16 groups where the pth group contains ceil$(4096/p^3)$ matrices of size $8p \times 8p$ so that, approximately, each group requires the same number of flops to compute.

On the x-axis we show the number of groups used: we begin by using only group one and successively add more groups to the computation until all 16 are included. We see that, for small numbers of groups, there is little difference between the two approaches whilst for larger number of groups it is beneficial to use a single call to the group API.

4 Data layout in memory

One major concern, affecting all the APIs, is the layout of the data in memory. As we will show later, the memory layout is critical to obtaining good performance on modern hardware and should therefore influence the design of the API.

In section 2 we outlined all the APIs using a pointer-to-pointer (P2P) data layout. This involves passing to the BBLAS function an array of pointers (e.g. `double **arrayC` in a batched DGEMM) where each element of the array is a pointer to a memory location containing a matrix. The main benefit of this approach is its flexibility: it is very easy to add more matrices into the batch by simply appending their respective memory locations onto the `arrayC` variable, for example.

However, this approach also has a major drawback: allocating memory for each matrix

Table 4.1: DGEMM function prototype using the separate fixed and variable memory API with the strided memory layout.

```
batchf_dgemm_stride(
const enum CBLAS_TRANSPOSE *transA,
const enum CBLAS_TRANSPOSE *transB,
const int m, const int n, const int k,
const double alpha,
const double *arrayA, const int lda, const int strideA
const double *arrayB, const int ldb, const int strideB
const double *beta,
double *arrayC, const int ldc, const int strideC,
const int batch_count, int *info
);
```

separately means that the data will be scattered throughout the RAM. When performing the computation this means that the CPU will need to load memory from many different locations, which is much slower than loading contiguous memory. This difference in memory access speed is much more apparent when we consider offloading computation to hardware accelerators such as GPUs and the Xeon Phi. In this case, since the matrices are spread throughout the RAM, they must be sent separately and each communication incurs a (comparatively large) latency cost.

One solution to this problem is to use a "strided" memory layout. For storing the matrices A_i in a fixed batch DGEMM this involves having one large array containing all the A_i in contiguous memory along with (in addition to the `lda` parameter) a `strideA` parameter which gives the amount of storage space between the matrices. For example, if `ptr` was a variable pointing to the first element of A_1 then `ptr + strideA` would be a pointer to the first element of A_2 and, in general, `ptr + (i-1)*strideA` would be a pointer to the first element of A_i. An example of the API for a fixed batch DGEMM using the separate fixed and variable API and the strided memory layout is given in Table 4.1 (c.f. Table 2.1) and could easily be extended to the other APIs.

An alternative solution is to use an "interleaved" memory layout. In this case we create one large array and fill it as follows. Firstly, store the first element of each matrix in turn, followed by the second element of each matrix etc. If we have a batch of three matrices

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}, \quad C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix},$$

and work in column major order, then the strided and interleaved memory layouts will store the elements as follows.

- Strided – $[a_{11}, a_{21}, a_{12}, a_{22}, b_{11}, b_{21}, b_{12}, b_{22}, c_{11}, c_{21}, c_{12}, c_{22}]$
- Interleaved – $[a_{11}, b_{11}, c_{11}, a_{21}, b_{21}, c_{21}, a_{12}, b_{12}, c_{12}, a_{22}, b_{22}, c_{22}]$

The interleaved format is particularly interesting for utilizing vectorization, a key component in GPU computation. For example, suppose we wish to compute a batch TRSM (triangular solve) with 5000 matrices of size 4×4. In the P2P and strided formats each multiprocessor of the GPU has access to one (or perhaps several) matrices. As we progress through the columns of the matrix we need to perform one division and multiple subtractions per column. However, each streaming multiprocessor can only perform one type of operation per cycle, meaning that one division uses an entire cycle and therefore does not take full advantage of the vector instructions. By contrast, using the interleaved format, the GPU will need to perform 5000 divisions in parallel, followed by 5000 subtractions etc. This is enough to completely fill the streaming multiprocessors and therefore take full advantage of the vector instructions.

Both the strided and interleaved memory layouts store the matrices in contiguous memory and will therefore be more efficient to load into the CPU cache and offload to hardware accelerators. However, we have lost the flexibility of the P2P approach. In order to add extra matrices to our batch we need to allocate an additional large block of contiguous memory and copy all the data across, which can be extremely expensive relative to the actual computation time.

The choice of which memory layout to use may depend largely upon the specific application within which BBLAS is being used. Using the, potentially more efficient, strided and interleaved layouts requires prior knowledge about the number and size of the matrices in the batch so that a sufficient amount of memory can be allocated; and this information is not necessarily available in all applications. For example, when solving separable elliptic equations using the algorithm by Swarztrauber [18] (parallelized in [19]) the matrices are spread throughout the RAM, meaning that only the P2P memory layout can be used. The P2P memory layout is also applicable to matrix-free finite element methods [15].

However in domain decomposition on a parallel distributed architecture, each node contains many matrices contiguous in memory and therefore the strided memory layout is more appropriate [3]. Another application in which matrices are stored in contiguous memory is image processing [6], [17], for example.

4.1 Experiments on memory allocation and transfer time

This subsection contains a number of experiments designed to determine the relative cost of the memory allocation and transfer time using the P2P, strided, and interleaved memory layouts. Note that, in terms of memory allocation and transfer time, the strided and interleaved layouts are identical: they are simply permutations of one another. Therefore we will focus on the differences between the P2P and strided layouts.

As before, our intial experiments in this section are run on a NUMA node with 2 sockets, using Intel Xeon CPU E5-2650 v3 chips (2.3GHz, Haswell architecture), for a total of 20 cores. The cache is flushed between each invocation of the functions [20] and the memory on the NUMA node is interleaved between the processors. In addition, to measure the time for allocating and transferring memory to a hardware accelerator, we use an NVIDIA K40c GPU connected via PCIe 3.0. Note that, although a GPU is used in these experiments the

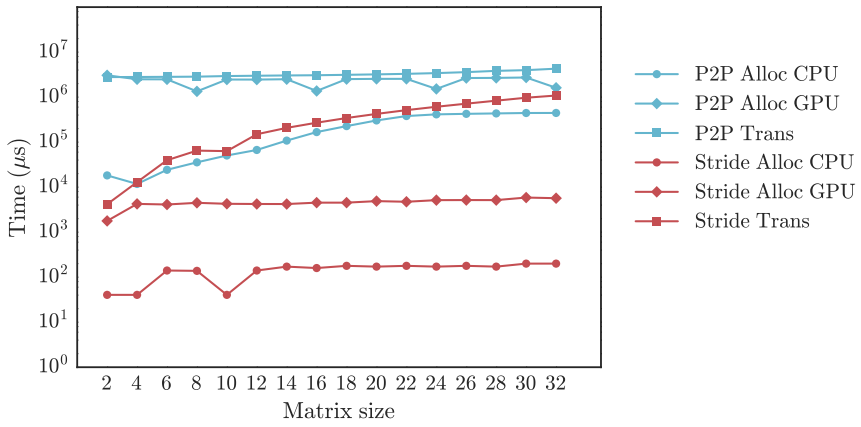

Figure 4.1: Time (in microseconds) for the allocation of the memory on the host and GPU, and for the memory transfer between the host and GPU, using both the P2P and strided memory layouts, for a variety of small matrix sizes. In each case a fixed batch of 10000 matrices is used.

conclusions regarding memory transfers should also be applicable to models of the Xeon Phi which are also connected via PCIe 3.0 in coprocessor mode.

In addition, we present the corresponding results for the self-hosted Xeon Phi (Knights Landing). The results above do not apply to this architecture since it has a faster connection to the main RAM along with upto 16GB of high-bandwidth MCDRAM. The Xeon Phi is the first commercially available product to use MCDRAM, which offers high-bandwidth memory access (400GB/s) due to its 3D memory design. In particular we use the Intel Xeon Phi 7250 processor with 68 cores running at 1.4GHz. It is booted in the default "hybrid" mode where 8GB of the fast MCDRAM is allocated by the user and the remaining 8GB is used as a cache by the system.

Our experiment consists of computing a batch of 10000 DGEMM operations using square matrices of varying size. All the matrices are chosen to have elements taken from a random uniform distribution on the interval $[0, 1]$. We run this experiment using the P2P and strided memory layouts on both the CPU, GPU, and Xeon Phi whilst timing the memory allocations, the memory transfer time (between the host and GPU), and the time to compute the actual result using both MKL and cuBLAS versions of batched DGEMM. The reported GPU transfer time includes both the time to send the parameters and receive the result back on the host, including the latency.

In Figure 4.1 we show the time required to allocate the memory in RAM on the host and GPU, along with the time required to transfer memory between the host and GPU, using both the P2P and strided memory layouts. It is clear that using the strided memory layout is much more efficient in terms of both memory allocation and transfer. In particular, the difference in memory allocation time shows that allocating one large block of memory is more efficient than allocating multiple small blocks scattered throughout the RAM and can be more than three orders of magnitude faster for the larger matrix sizes on both the CPU

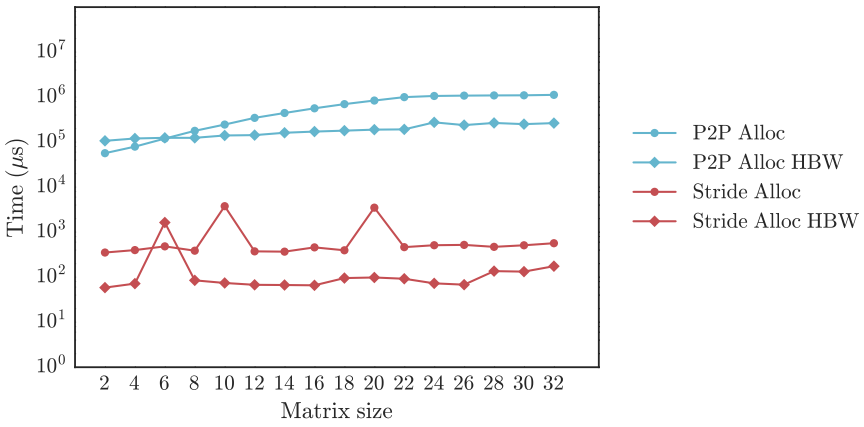

Figure 4.2: Time (in microseconds) for the allocation of the memory on the system RAM and the fast MCDRAM, using both the P2P and strided memory layouts, for a variety of small matrix sizes. In each case a fixed batch of 10000 matrices is used. In the legend MCDRAM allocations are denoted by HBW (high-bandwidth).

and GPU.

For the memory transfer to the GPU, it is clearly more efficient to transfer large blocks of data using the stride layout as opposed to many small matrices individually in the P2P layout: using the P2P layout we need to communicate with the GPU 10000 times (once for each matrix in the batch) and each of these communications is subject to the same latency cost, which dominates the transfer time. This explains why the transfer time for the P2P layout is almost constant in Figure 4.1. For the smaller matrix sizes we see that performing this memory transfer using the stride layout can be up to three orders of magnitude faster.

Unfortunately, it is also very expensive to convert from the P2P layout to the strided memory layout. Our experiments showed that, even when using OpenMP with 20 cores, the time to convert from the P2P to the stride layout was on average 2.4 times more expensive than performing the computation using the P2P layout with Intel MKL.

In Figure 4.2 we show the corresponding results for allocating the memory using the P2P and strided memory layouts on the Intel Xeon Phi. We analyze both cases: when the matrices are allocated on the main RAM, or when the matrices are allocated directly on the high-bandwidth MCDRAM. Similarly to the host and GPU memory allocation in Figure 4.1, we see that the strided memory layout is much more efficient in both cases. We also see that using the high-bandwidth MCDRAM is significantly cheaper than allocating in the main RAM. Note also that the drivers used to interface with the MCDRAM are less mature than those used in DDR4 (which constitutes the main RAM), so we can expect this allocation time to decrease in the coming years.

Our next set of results gives the performance obtained using the CPU, GPU, and Xeon Phi for batched GEMM both including and excluding the memory transfer time.

In Figure 4.3 we show the performance obtained when executing the batched DGEMM on the CPU cores using Intel MKL and on the GPU using NVIDIA cuBLAS. For the GPU

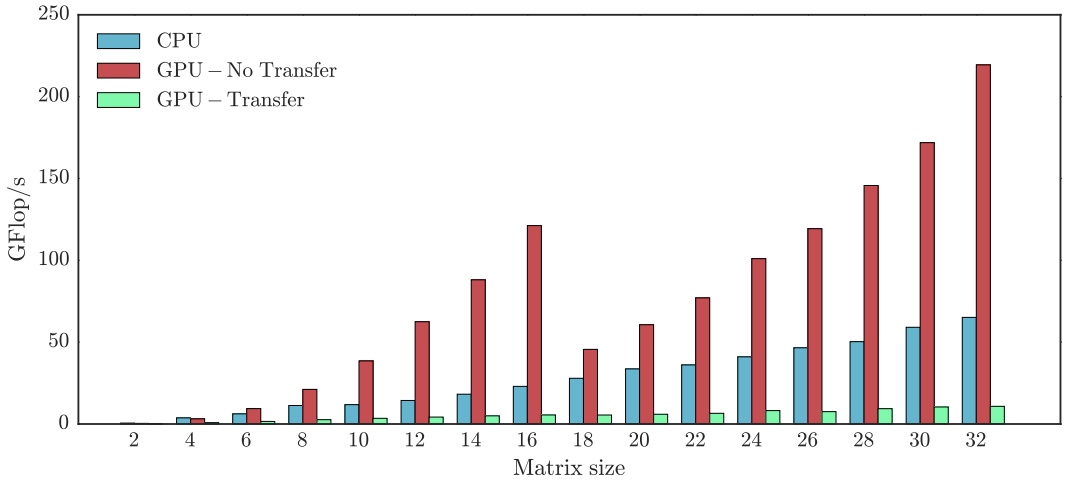

Figure 4.3: Performance for computing a fixed batch DGEMM with 10000 matrices, of various sizes, on 20 CPU cores using Intel MKL and an NVIDIA K40c GPU using NVIDIA cuBLAS. For the GPU we report the performance when both including and excluding the memory transfer time between the host and device, using the strided memory layout.

we provide the performance when both including and excluding time required to transfer the data between the host and device. This experiment was performed using the strided memory layout, which we have shown provides better memory allocation and transfer times.

From this plot it is clear that there is a stark difference between the GPU performance when we include and exclude the memory transfer time. When this transfer is included the performance drops dramatically and it is preferable to perform all computation on the CPU cores, avoiding the memory transfers to and from the device.

If the data is already loaded onto the GPU, or if the memory transfer can be done concurrently with some unrelated CPU computation, then there is a clear benefit to using the GPU; otherwise it may be preferable to simply use the CPU cores for the computation. Whether or not the memory transfer time can be hidden by CPU computation depends entirely upon the specific application that the user is working on, which highlights the tight connection that exists between the BBLAS and its use in specific applications. One might also consider a hybrid approach to computing a BBLAS routine in which the CPU cores and devices (such as GPUs) split the batch between themselves, allowing the CPU to work on part of the batch whilst the remainder is being transferred to the devices.

In Figure 4.4 we show the corresponding performance results for the Intel Xeon Phi. We show the performance when the matrices are initially stored on the main RAM and in the MCDRAM. It is clear that, in most cases, it is beneficial to use the MCDRAM instead of the main RAM, as there is a higher bandwidth connection between the memory and computational cores. The reason for the loss in performance using the MCDRAM for matrices of size 28 is currently under investigation.

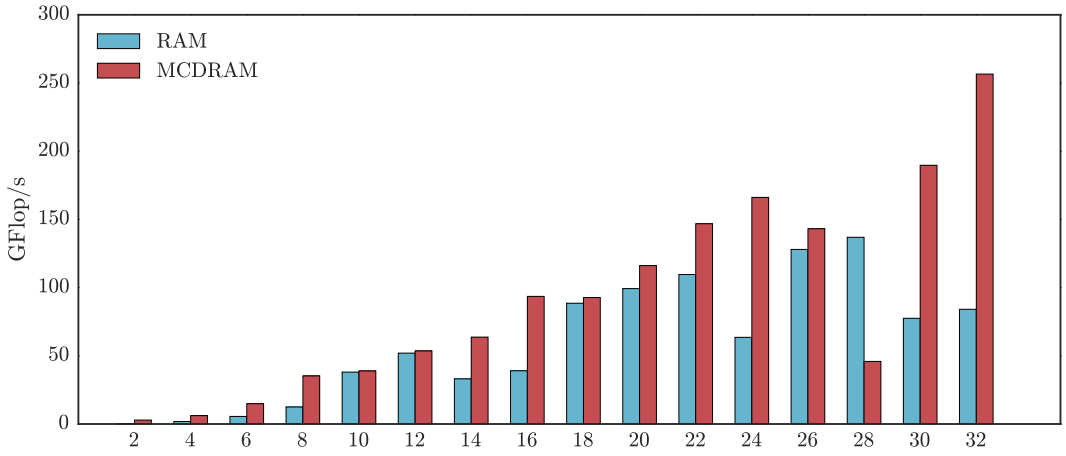

Figure 4.4: Performance for computing a fixed batch DGEMM with 10000 matrices, of various sizes, using the Intel Xeon Phi. We consider the cases where the matrices are allocated in the main RAM or directly in the MCDRAM.

5 Conclusions

In this work we have analyzed three different approaches to creating a standard API for batched BLAS operations. We have also analyzed the performance effects of each API and shown how the memory layout can impact the allocation and transfer times.

In terms of the approach taken for the API design, it seems that the simplest choice for users is to use separate functions for the fixed and variable batch computations. Most applications require fixed batch operations and this API makes this as simple as possible. The group API allows the possibility of multiple fixed batch operations in parallel but, from our experiments, seems to offer little performance benefit to compensate for its additional complexity. Meanwhile, the flag-based API simply wraps the fixed and variable interfaces into a single function, but creates a more complicated API for the user. Therefore we recommend that separate routines are created for fixed and variable batch computation and that the API shown in section 2.2 is adopted by the community.

It seems clear that the choice of which memory layout to use will depend largely upon the user's application. Our experiments show that the strided memory layout (and therefore also the interleaved layout) is much more efficient in terms of memory allocation and transfer and consequently they should be preferred whenever possible. However, these memory layouts can only be used when the number of matrices in the batch and their sizes are known a priori. In many applications this is not possible and therefore it is important to also support the pointer-to-pointer memory layout.

Whilst there are clear memory allocation and transfer benefits to the strided and interleaved memory layouts, further work is required to determine any performance benefits that they may have: an efficient and optimized implementation of each approach is required to

accurately compare their performance.

Sample API code is available at https://github.com/sdrelton/bblas_api_test.

Acknowledgments

The authors would like to thank Jack Dongarra, Mark Gates, Nicholas Higham, Sven Hammarling, and Piotr Luszczek for their helpful comments on earlier versions of this work. They would also like to thank The University of Tennessee for access to their computing resources.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[2] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, et al. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series* 180.1 (2009). DOI: 10.1088/1742-6596/180/1/012037.

[3] Emmanuel Agullo, Luc Giraud, and Mawussi Zounon. "On the Resilience of Parallel Sparse Hybrid Solvers". In: *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*. 2015, pp. 75–84. DOI: 10.1109/HiPC.2015.9.

[4] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* abs/1605.02688 (May 2016). URL: http://arxiv.org/abs/1605.02688.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback). DOI: 10.1137/1.9780898719604.

[6] John Ashburner. "A fast diffeomorphic image registration algorithm". In: *NeuroImage* 38.1 (2007), pp. 95–113. ISSN: 1053-8119. DOI: 10.1016/j.neuroimage.2007.07.007.

[7] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. "A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures". In: *Parallel Computing* 35.1 (2009), pp. 38–53.

[8] Jack Dongarra, J. Du Croz, Iain Duff, and Sven Hammarling. "A set of Level 3 Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* 16 (1990), pp. 1–28.

[9] Jack Dongarra, J. Du Croz, Sven Hammarling, and R. J. Hanson. "An extended set of FORTRAN Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* 14 (1988), pp. 1–32.

[10] Jack Dongarra, Iain Duff, Mark Gates, Azzam Haidar, et al. *A Proposed API for Batched Basic Linear Algebra Subprograms*. MIMS EPrint 2016.25. UK: The University of Manchester, 2016.

[11] Iain Duff and J. K. Reid. "The Multifrontal Solution of Indefinite Sparse Symmetric Linear Equations". In: *ACM Trans. Math. Softw.* 9.3 (1983), pp. 302–325. DOI: 10.1145/356044.356047.

[12] Nicholas J. Higham and Vanni Noferini. "An algorithm to compute the polar decomposition of a 3×3 matrix". In: *Numer. Algorithms* (2015), pp. 1–21. DOI: 10.1007/s11075-016-0098-7.

[13] Ali Khodayari, Ali R. Zomorrodi, James C. Liao, and Costas D. Maranas. "A kinetic model of Escherichia coli core metabolism satisfying multiple sets of mutant flux data". In: *Metabolic Engineering* 25 (2014), pp. 50–62. DOI: 10.1016/j.ymben.2014.05.014.

[14] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krough. "Basic Linear Algebra Subprograms for FORTRAN usage". In: *ACM Trans. Math. Softw.* 5 (1979), pp. 308–323.

[15] Karl Ljungkvist. "Matrix-Free Finite-Element Operator Application on Graphics Processing Units". In: *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*. 2014, pp. 450–461. DOI: 10.1007/978-3-319-14313-2_38.

[16] O. E. B. Messer, J. A. Harris, S. Parete-Koon, and M. A. Chertjkow. "Multicore and accelerator development for a leadership-class stellar astrophysics code". In: *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*. 2012.

[17] Pedro Valero-Lara. "Multi-GPU acceleration of DARTEL (early detection of Alzheimer)". In: *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*. 2014, pp. 346–354. DOI: 10.1109/CLUSTER.2014.6968783.

[18] Paul N. Swarztrauber. "A direct method for the discrete solution of separable elliptic equations". In: *SIAM J. Numer. Anal.* 11.6 (1972), pp. 1136–1150. DOI: 10.1137/0711086.

[19] Pedro Valero-Lara, Alfredo Pinelli, and Manuel Prieto-Matias. "Fast finite Difference Poisson solvers on heterogeneous architectures". In: *Computer Physics Communications* 185.4 (2014), pp. 1265–1272. DOI: 10.1016/j.cpc.2013.12.026.

[20] R. Clint Whaley and Anthony M. Castaldo. "Achieving accurate and context-sensitive timing for code optimization". In: *Softw. Pract. Exper.* 38.15 (2008), pp. 1621–1642. DOI: 10.1002/spe.884.

[21] R. Clint Whaley and Jack Dongarra. "Automatically Tuned Linear Algebra Software". In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. 1998, pp. 1–27.

[22] Sencer N. Yerlan, Tim. A. Davis, and Sanjay Ranka. *Sparse Multifrontal QR on the GPU*. Tech. rep. The University of Florida, 2013.