The BLAS API of BLASFEO: Optimizing Performance for Small Matrices

GIANLUCA FRISON, TOMMASO SARTOR, ANDREA ZANELLI, and MORITZ DIEHL,
University of Freiburg

Basic Linear Algebra Subroutines For Embedded Optimization (BLASFEO) is a dense linear algebra library providing high-performance implementations of BLAS- and LAPACK-like routines for use in embedded optimization and other applications targeting relatively small matrices. BLASFEO defines an application programming interface (API) which uses a packed matrix format as its native format. This format is analogous to the internal memory buffers of optimized BLAS, but it is exposed to the user and it removes the packing cost from the routine call. For matrices fitting in cache, BLASFEO outperforms optimized BLAS implementations, both open source and proprietary. This article investigates the addition of a standard BLAS API to the BLASFEO framework, and proposes an implementation switching between two or more algorithms optimized for different matrix sizes. Thanks to the modular assembly framework in BLASFEO, tailored linear algebra kernels with mixed column- and panel-major arguments are easily developed. This BLAS API has lower performance than the BLASFEO API, but it nonetheless outperforms optimized BLAS and especially LAPACK libraries for matrices fitting in cache. Therefore, it can boost a wide range of applications, where standard BLAS and LAPACK libraries are employed and the matrix size is moderate. In particular, this article investigates the benefits in scientific programming languages such as Octave, SciPy, and Julia.

CCS Concepts: • **Mathematics of computing → Mathematical software**;

Additional Key Words and Phrases: Linear algebra, libraries, high-performance, matrix, BLAS

ACM Reference format:
Gianluca Frison, Tommaso Sartor, Andrea Zanelli, and Moritz Diehl. 2020. The BLAS API of BLASFEO: Optimizing Performance for Small Matrices. *ACM Trans. Math. Softw.* 46, 2, Article 15 (May 2020), 36 pages.
https://doi.org/10.1145/3378671

1 INTRODUCTION

This article describes the implementation of a standard BLAS application programming interface (API) in the BLASFEO framework [9], and investigates its use in scientific programming languages.

BLASFEO (Basic Linear Algebra Subroutines For Embedded Optimization) is an open source dense linear algebra (DLA) library [2] aimed at providing high-performance implementations of BLAS- and LAPACK-like routines for use in embedded optimization and small-scale high-performance computing in general [6, 8, 13]. One of the most remarkable features of BLASFEO is

This research was supported by the German Federal Ministry for Economic Affairs and Energy (BMWi) via eco4wind (0324125B) and DyConPV (0324166B), and by DFG via Research Unit FOR 2401.
Authors' addresses: G. Frison, T. Sartor, A. Zanelli, and M. Diehl, Georges-Koehler-Allee 102, 79110 Freiburg, Germany; emails: {gianluca.frison, tommaso.sartor, andrea.zanelli, moritz.diehl}@imtek.uni-freiburg.de.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
0098-3500/2020/05-ART15 $15.00
https://doi.org/10.1145/3378671

the definition of a novel interface for DLA routines, where high-performance implementations of the computing routines operate on a performance-optimized matrix format (denoted panel-major, and resembling the packed format of the internal memory buffers used in many optimized BLAS implementations [14, 28]), and packing and unpacking routines are exposed to the user. From now on in this article, this interface is indicated as the BLASFEO API. The BLASFEO API allows one to significantly reduce the overhead of DLA routines, especially for small sized matrices, in which case the quadratic cost of packing and unpacking data is not negligible with respect to the cubic cost of the DLA operation itself. However, the BLASFEO API is not compatible with the standard DLA interfaces.

BLAS [19] and LAPACK [1] are the de facto standard DLA interfaces, and have been so for several decades. They are extremely widespread and form the basis of the major part of numerical and scientific software, and in this they represent a very successful example of combining high-performance and portability. Several highly optimized implementations exist, both open source (e.g., ATLAS [25], OpenBLAS [21], BLIS [27]) and proprietary (e.g., MKL [18]). From now on in this article, the BLAS and LAPACK standard interface is indicated as the BLAS API.

The aim of this research work is to investigate the addition of a BLAS API to the BLASFEO framework, alongside the original BLASFEO API. By analogy with an optimization problem, it can be said that the objective is unchanged: attaining the highest performance for the routines of a DLA library, in the case of matrices fitting in cache. However, there are two more constraints, one hard and one soft: the strict compliance with the BLAS API, and the desire of reusing as much code as possible between the BLAS API and the BLASFEO API. As happens in an optimization problem, the introduction of additional constraints can lower the optimal value attained by the objective.

The numerical experiments in Section 4 show that, compared to the BLASFEO API, the performance loss for the BLAS API of BLASFEO ranges between negligible and 20% depending on the routine, matrix size, and architecture, with an average of about 10–15%. For the matrix sizes of interest, this still puts the BLAS API of BLASFEO solidly ahead of (the single-threaded version of) other optimized BLAS API implementations (both open source and proprietary), with only a handful of cases where it is roughly on par with them. The addition of multi-threaded capabilities to the BLASFEO framework is outside the scope of the current paper.[1] The comparisons in this article are limited to standard BLAS API library implementations of DLA routines. Therefore, alternative approaches for small-scale DLA routines such as code generation [11, 12, 23], C++ templates [3, 5], or specialized compilers [22] are not considered. Likewise, approaches based on batched BLAS [17] are also not considered.

The constraint about strict compliance with the BLAS API has important consequences on the algorithmic choices, as discussed in Section 2, taking `dgemm` as an example. In particular, the two extremes of working natively in column-major on all input and output matrices, or alternatively packing/unpacking them all in BLASFEO native panel-major format, both have major performance flaws. Two better algorithmic variants are identified, which only pack some matrices (panels from the left factor A, and additionally the right factor B for one variant) while operating natively in column-major for all other matrices. The need to perform packing "online," within each routine call, is a major change compared to the original BLASFEO API implementation scheme. However, if carefully performed and implemented as discussed in Section 2, this packing does not introduce excessive overhead, and it even represents the tool now used to complete the BLASFEO API itself with the routine variants disregarded in the original implementation scheme in [9] (such as the

[1]Note that one strategy to get multi-threaded linear algebra routines for relatively large matrices is to use libraries such as PLASMA [4] or libflame [26], which employ a task scheduler with sequential BLAS and LAPACK for tasks on modest sized tiles. Thanks to its good performance for small matrices, BLASFEO is an excellent choice to provide the sequential BLAS and LAPACK routines.

"TN" and "TT" versions of `dgemm`, where the left factor matrix A is transposed). A further consequence of the compliance with the BLAS API is the need to stick to a destructive routine interface, whereas the BLASFEO API adds an additional matrix argument reserved for the output and therefore it is not destructive. In many applications, a non-destructive interface can avoid the need to perform additional matrix copies, which in the case of small matrices are comparably costly.

The desire of reusing as much code as possible affects the details of implementation, as discussed in Section 3. In particular, in the BLASFEO framework linear algebra kernels are defined as the innermost loop coded as a stand-alone routine, and are implemented in a modular fashion by exploiting the flexibility of the assembly language. These kernels are split into smaller sub-routines which perform elementary operations such as loading or storing of a matrix to/from registers, multiplication of two panels, or factorization of a register-resident sub-matrix. The sub-routines make use of a custom function calling convention, which allows one to pass data in floating point (FP) registers and reduces the calling overhead. This framework naturally allows the implementation of linear algebra kernels operating on matrix arguments with different storage formats (in this case, column-major and panel-major), by simply "gluing together" sub-routines operating on different storage formats. A high degree of code reuse is attained, since the actual operations are implemented in the shared sub-routines, while the tailored kernels are simply a sequence of calls to the sub-routines themselves. More details are found in Section 3.2.

One of the most particular features of the BLASFEO kernels with respect to other optimized BLAS implementations like OpenBLAS or BLIS is the fact that the BLASFEO kernels operate on the left and right factors packed into panels of identical height p_s. In case of non-square kernels, this implies that multiple panels are swept at once from the left or the right factor, as opposed to a single higher panel in other optimized BLAS implementations. In the BLASFEO API, this is a requirement, since the same panel-major matrix must be freely employed as a left or right factor (and as output matrix too). In the BLAS API, this feature is maintained, mainly to maximize the code shared with the BLASFEO API. Even if the sweeping of multiple shorter panels is possibly slightly sub-optimal compared to the sweeping of a single higher one (due to cache associativity), in practice the BLASFEO API shows excellent performance on all tested architectures [9]. Conversely, in the BLAS API this feature turns out to be an intrinsic advantage in the implementation of routines with symmetric nature (like, e.g., `dsyrk` or `dpotrf`), since the mathematical symmetry of the operation can be carried over to the working memory buffers, which are shared between left and right factors, reducing the amount of packing. More details are found in Section 3.1.

Other design choices compatible with the BLAS API compliance constraint are carried on unchanged from the BLASFEO API. No cache blocking is performed, and level 3 routines are implemented as nested triple loops, whose innermost loop is coded as a stand-alone routine, the kernel. The modular nature of the assembly kernels allows one to easily code kernels tailored to the different DLA routines and fully exploiting vectorization also in the corner cases. LAPACK routines like factorizations are implemented similarly to the level 3 BLAS routines (i.e., as nested triple loops whose innermost loop is a tailored kernel), and not on top of them. As in the case of the BLASFEO API, it can be said that the block size in blocked LAPACK routines shrinks to the kernel size, while unblocked routines coincide with the tailored kernels and operate on fixed-size register-resident sub-matrices.

Since BLAS and LAPACK are at the foundation of modern scientific programming languages, Section 5 of this article evaluates the use of the BLAS API of BLASFEO in Octave, Python SciPy, and Julia, and investigates to what extent a DLA library optimized for small matrices can speed up computations in high-level languages. These languages are chosen because of their widespread use and their open source nature. As such, by default, they are distributed with open source BLAS and LAPACK implementations such as OpenBLAS. Being open source software too, BLASFEO can

contribute to improve the performance of such languages possibly more than proprietary BLAS and LAPACK implementations do, while keeping a fully open source software stack.

2 COMPARISON OF DGEMM ALGORITHMS FOR SMALL MATRICES

This section proposes and compares several algorithms to implement the BLAS routine dgemm in the BLASFEO framework, with a focus on optimizing performance for small matrices. Similar findings apply in the implementation of other BLAS and LAPACK routines.

2.1 Notation

The general matrix-matrix multiplication routine dgemm is certainly the most famous and important BLAS routine, and therefore it is used here as an example. In order to keep the notation more readable, the dgemm routine is here assumed to implement the operation $C \Leftarrow A \cdot B + C$. Therefore a call in the C language is assumed to be in the form (where the function arguments of the transposition options, the scalar factors α and β, and the leading size of the matrices are omitted)

```
void dgemm_(int *m,
            int *n,vv
            int *k,
            double *A,
            double *B,
            double *C);
```

where $m \times n$ is the size of the result matrix and k is the length of each dot product, A and B denote the left and right matrix factors, and C denotes the result matrix. In the standard BLAS API, all matrices A, B, and C are stored in column-major format.

It is well known that in double precision all level 3 BLAS and many LAPACK routines such as factorizations can be implemented using dgemm [15, 16]. In the BLASFEO framework, this is exploited in the design of kernels for level 3 BLAS and LAPACK routines, which all share the common structure (where for compactness the function arguments of the scalar factors α and β and the size of the buffers are omitted)

```
void kernel_dgemm(int k_max,
                  double *A_k,
                  double *B_k,
                  double *C_k,
                  double *D_k);
```

with the only exception of dtrsm kernels, which take as an extra argument the triangular matrix of the system to be solved,

```
void kernel_dtrsm(int k_max,
                  double *A_k,
                  double *B_k,
                  double *C_k,
                  double *D_k,
                  double *E_k);
```

Here $k_{\max}$ is the length of the innermost loop, A_k and B_k are the sub-matrices of the left and right factors A and B, C_k and D_k are the read and written sub-matrices of the result matrix C, and E_k is the triangular sub-matrix of the system to be solved in the case of dtrsm. Note the subscript $_k$, which denotes that these quantities refer to the kernel sub-matrices (as opposed to the routine

matrices, which are denoted without the subscript $_k$). These kernel sub-matrices may be stored in column-major or in panel-major formats depending on the kernel implementation.

2.2 Algorithmic Variants

This section describes five algorithmic variants for the implementation of the standard BLAS API `dgemm` routine leveraging the existing BLASFEO framework. It is assumed that the high-performance backend of BLASFEO is employed, such that the panel-major matrix format is the memory layout of the BLASFEO matrices `blasfeo_dmat`.

Algorithm "A". It is straightforward to add a standard BLAS API `dgemm` routine in the BLASFEO framework by only using existing BLASFEO routines for packing, unpacking, and the BLASFEO API `blasfeo_dgemm_nt` routine. This is the algorithmic variant named "A."

In more detail, the routine `blasfeo_memsize_dmat` returns the amount of memory that should be provided to create a `blasfeo_dmat` matrix. The memory for all arguments A_b, B_b, C_b of the BLASFEO `dgemm_nt` routine is allocated dynamically and aligned to cache line boundaries, and the `blasfeo_dmat` matrices A_b, B_b, C_b are created with the `blasfeo_create_dmat` routine. The whole input matrices A, B, C are packed into the `blasfeo_dmat` matrices A_b, B_b, C_b using the routines `blasfeo_pack_dmat` and `blasfeo_pack_tran_dmat`. At this point, the `blasfeo_dgemm_nt` routine can be employed to compute the matrix product. Finally, the result is stored back in the matrix C using the routine `blasfeo_unpack_dmat`. Overall, the amount of packing is of $mk + nk + 2mn$ matrix elements.

All parametric variants of `dgemm` can be handled by transposing while packing the factor matrices A and B. The "NT" variant of the `dgemm` routine is chosen as the computational kernel, as this is the best-performing variant in the BLASFEO API.

This algorithm is easy to implement without coding any new linear algebra routine or kernel, but building entirely on top of the existing BLASFEO API. However, it unnecessarily performs packing and unpacking of C (which is not needed for performance), and it does not exploit the order of linear algebra loops within the BLASFEO `dgemm` routine. Furthermore, for very small matrices the dynamic memory allocation can severely affect performance.

Algorithm "B". By opening the lid on the internals of the BLASFEO API `dgemm` routine, it is possible to obtain better performing algorithms. The algorithmic variant named "B" operates natively on the matrix C in column-major, reducing by a factor 2 (in the case of square matrices) the amount of packing, amounting to $mk + nk$ matrix elements. This requires the implementation of new kernels, in which A_k and B_k are in panel-major and C_k and D_k are in column-major. The factor matrices A and B can still be freely transposed while packed, and therefore also in this algorithmic variant only the "NT" variant of the `dgemm` kernel is employed.

Additionally, the fact that the `dgemm` routine is implemented as a triple loop, with the outer loop over m, is exploited. At each iteration of the loop over m, a sub-matrix of A is determined, where the number of rows is fixed and equal to the height m_r of the computational kernel, and the number of columns is equal to k. The two innermost loops repeatedly stream this sub-matrix of A while sweeping over the entire matrix B. Therefore, A can be packed m_r rows at a time in the outermost loop, in a memory buffer of size $O(k)$. Regarding B, it is packed as in algorithmic variant "A," outside the triple loop.

The main drawbacks of this algorithm are the overhead due to the packing of both A and B, and the usage of dynamic memory allocation, which adversely affect performance for small matrices. Dynamic memory allocation is employed because the buffer size for the B matrix grows as $O(nk)$, and in the case of square matrices it quickly gets large with the matrix size. Therefore, its size can easily exceed the stack size, which can be rather small in the case of embedded devices.

Algorithm "C". The aim of algorithmic variant "C" is to further reduce the amount of packing and to avoid the dynamic memory allocation and therefore to improve performance for small matrices. In order to do so, this variant operates natively on both matrices C and B, each stored in column-major order. The matrix A is still packed (and possibly transposed) as in algorithmic variant "B," in a memory buffer of size $O(k)$. The amount of packing reduces to mk matrix elements, which is independent of n. Since the matrix B is used natively and therefore not explicitly transposed, the two variants "NN" and "NT" of the `dgemm` kernels are employed (handling the `dgemm` variants "NN" and "TN," and "NT" and "TT," respectively).

Automatic memory allocation (that allocates memory on the stack) is employed for the A_k buffer, of size $O(k)$, avoiding any dynamic memory allocation. The main drawback of this algorithmic variant is that the column-major memory layout of B adversely affects cache usage, especially in the "NT" and "TT" `dgemm` variants where B is accessed across columns.

Algorithm "Ct". Algorithmic variant "Ct" is analogous to algorithmic variant "C," and loosely speaking it can be considered its transposition (hence the "t" in the name). Compared to algorithmic variant "C," the variant "Ct" packs the B matrix instead of the A matrix, the order of outer loops is swapped (outermost over n), and finally the kernel sizes and types are transposed.[2] In particular, the matrix A is used natively and therefore not explicitly transposed, and the two variants "NT" and "TT" of the `dgemm` kernels are employed, handling the `dgemm` variants "NN" and "NT," and "TN" and "TT," respectively. The number of matrix elements packed by this variant is nk, which is independent of m. The B_k buffer, of size $O(k)$, is allocated on the stack using automatic memory allocation.

Algorithm "D". Algorithmic variant "D" natively operates on all matrices in column-major, and therefore it does not need any memory allocation or packing. As long as all matrix arguments are accessed natively and therefore not explicitly transposed, this variant requires a different kernel for each of the `dgemm` variants "NN," "NT," "TN," and "TT." In the column-major format, the variants "TN" and "TT" (which are the variants where the matrix A has to be transposed in the multiplication) are less friendly to vectorization, since contiguous A elements are used at different iterations of the loop over k. Therefore, on modern architectures, native kernels for these variants perform worse, with the performance penalty increasing with the vector size. In the case of the "TT" variant, a workaround consists in implementing the kernel based on a kernel for the "NN" variant as $C \Leftarrow \beta \cdot C + \alpha \cdot (B \cdot A)^T$. In the case of the "TN" variant, the same workaround cannot be employed, as in the transposition $(A^T \cdot B)^T = (B^T \cdot A)$ the left factor is still transposed. Therefore, the implementation of this variant requires the explicit transposition of either matrix factor, and therefore it is better replaced by algorithmic variants "C" or "Ct," where the explicit transposition is combined with packing into a better performing matrix format.

Therefore, this algorithmic variant is only implemented in the `dgemm` variants "NN," "NT," and "TT."

2.3 Comparison of Algorithmic Variants

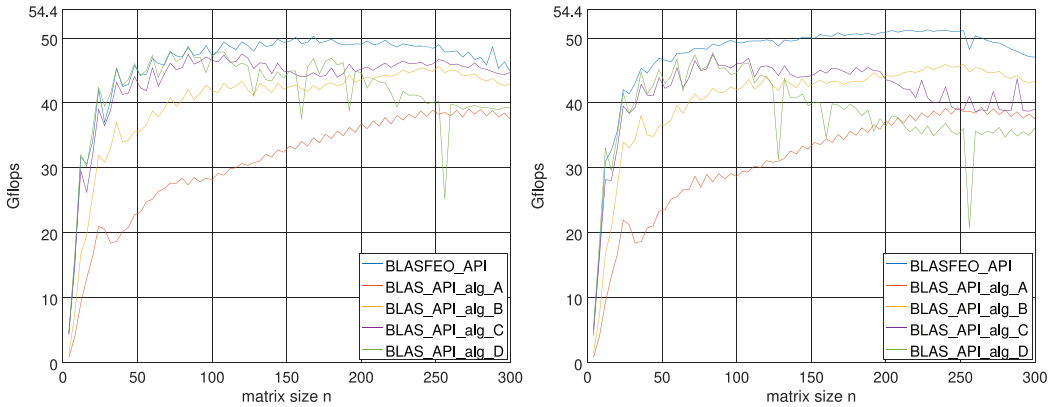
Table 1 summarizes the key features of the five algorithmic variants for the implementation of `dgemm`, as described in Section 2.2. It reports the memory layout of A_k, B_k, and C_k as employed in the computational kernel, and the number and type of computational kernels needed to compute all four `dgemm` variants. The table also reports the amount of memory used for internal buffers (and therefore excludes the memory needed to store the A, B, and C arguments of `dgemm`) and the type

[2]For example, `kernel_dgemm_tt_4x12_libc4cc` for algorithmic variant "Ct" is the equivalent of `kernel_dgemm_nn_12x4_lib4ccc` for algorithmic variant "C."

Table 1. Key Features of Algorithmic Variants for the Implementation of the BLAS Routine dgemm

alg.	A_k	B_k	C_k	mem. use	alloc.	packing	kernels
"A"	p	p	p	$O(mk + nk + mn)$	heap	$mk + nk + 2mn$	nt_ppp
"B"	p	p	c	$O(nk)$	heap	$mk + nk$	nt_ppc
"C"	p	c	c	$O(k)$	stack	mk	{nn, nt}_pcc
"Ct"	c	p	c	$O(k)$	stack	nk	{nt, tt}_cpc
"D"	c	c	c	$O(1)$	none	none	{nn, nt, tn*, tt}_ccc

In the kernel names, the first two characters encode the transposition of A and B ("n," non-transposed; "t," transposed); the last three characters encode the memory layout of A_k, B_k, and C_k ("c," column-major; "p," panel-major). Note that in the BLAS API dgemm implementation, D_k coincides with C_k. Kernels denoted with the exponent * are not vectorization friendly and generally perform poorly on modern architectures.



(a) dgemm_nn (b) dgemm_nt

Fig. 1. Comparison of algorithmic variants for the implementation of the BLAS routine dgemm in the case of square matrices: performance on Intel Haswell Core i7 4810MQ. Left picture: dgemm with `transa='n'` and `transb='n'`. Right picture: dgemm with `transa='n'` and `transb='t'`.

of memory allocation employed: automatic memory allocation on the stack, or dynamic memory allocation on the heap.[3] Finally, the number of matrix elements packed is also listed.

2.3.1 Comparisons for Square Matrices. In case of square matrices, algorithmic variants "C" and "Ct" are equivalent: therefore, only variant "C" is considered here. Figure 1 shows the performance plot for the four algorithmic variants "A." "B," "C," and "D," and for the "NN" and "NT" dgemm variants, on an Intel Haswell Core i7 4810MQ. The performance of the BLASFEO API routines is also added for comparison.

Algorithmic variant "A" shows a large overhead, especially for small matrices, being at least two times slower than the BLASFEO API routines for matrices smaller than 50. Algorithmic variant "B" significantly reduces overhead, and it always performs better than variant "A" in the matrix size range of interest. Both variants perform similarly across all four dgemm variants, as they employ a single computational kernel and the transpositions are performed while packing at no extra cost.

Algorithmic variant "C" further reduces overhead for matrices smaller than about 200, approaching the performance of the BLASFEO API routines. However, when the matrix B is employed transposed, the performance of this variant degrades when L1 cache size is exceeded. Finally, compared

[3]Static memory allocation is not a viable option, as it would make the routines not thread-safe.

with algorithmic variant "C," variant "D" slightly reduces overhead only for very small matrices, and its performance degrades more when L1 cache size is exceeded. Software prefetching can mitigate but not solve these performance degradations.

2.3.2 Considerations for Rectangular Matrices. In the case of rectangular matrices, the data in Table 1 leads to some additional considerations. Section C in the Appendix contains some benchmarks in the case of both square and skinny matrices.

When the result matrix C is rectangular, algorithmic variants "C" and "Ct" are no longer equivalent. They perform the same number of flops and they have the same memory usage, but the amount of packing is different. In particular, algorithmic variant "C" requires packing the mk elements of the matrix A (amount independent of n), and therefore it is advantageous when C is a fat matrix with $m < n$. Conversely, algorithmic variant "Ct" requires packing the nk elements of the matrix B (amount independent of m), making it the best choice when C is a thin matrix with $m > n$.

When m and n are both very small, for any value of k the cost of packing any of the matrices A or B cannot be well amortized over the number of flops, namely $2mnk$. Therefore, in this case algorithmic variant "D" (if available for the `dgemm` variant) is likely the best choice.

More generally, the case of square matrices maximizes the ratio between the number of flops and the number of matrix elements, and thus minimizes the impact of packing on the computation time. Therefore, in the case of rectangular matrices, algorithmic variants involving less packing become more advantageous compared to the findings in Section 2.3.1.

2.4 Proposed Approach

The implementation of a generic DLA routine in the BLAS API of BLASFEO makes use of algorithmic variant "C" for moderately sized matrices, and algorithmic variant "B" for larger matrices. Other algorithmic variants can possibly supplement these variants in special cases. As an example, in the case of non-symmetric routines like `dgemm`, algorithmic variant "Ct" can replace algorithmic variant "C" in the case of thin result matrices. Algorithmic variant "D" (if a high-performance implementation is available, like for the "NN," "NT," and "TT" variants, but not for the "TN" variant of `dgemm`) can be employed in the case of very small matrices.

The optimal switching point differs for different linear algebra routines and architectures. In the examples in Figure 1, the switching point between algorithmic variants "B" and "C" is for $m = n = k \approx 400$ for the "NN" `dgemm` variant[4] and for $m = n = k \approx 200$ for the "NT" `dgemm` variant. The switching point can be manually limited to a maximum value using a compilation flag to ensure that the memory employed by the algorithmic variant "C" does not exceed the stack size on the specific OS and architecture; algorithmic variant "B" employs dynamic memory allocation and therefore safely works for much larger matrix sizes.

In the case of the `dgemm` routine, the proposed generic approach requires the implementation of three kernel types: `nt_ppc`[5] for algorithmic variant "B" and `nn_pcc` and `nt_pcc` for algorithmic variant "C." These add to the existing kernel types `nn_ppp` and `nt_ppp` used in the BLASFEO API. Regarding the supplementary variants, algorithmic variant "Ct" requires the implementation of the kernels `nt_cpc` and `tt_cpc`, which are internally based on the kernels for algorithmic variant "C" thanks to the matrix multiplication transposition rule (so no intrinsically new kernel type is required). Algorithmic variant "D" requires the implementation of the kernels `nn_ccc`, `nt_ccc`, and `tt_ccc`, out of which the first two are intrinsically new kernels, while the `tt_ccc` kernel is

[4]Switching point not visible in Figure 1(a).
[5]The kernel naming convention is explained in the caption of Table 1.

internally based on the nn_ccc kernel, again thanks to the matrix multiplication transposition rule. Section 3.2 explains how all these kernels are coded in assembly in a modular fashion.

3 DETAILS OF IMPLEMENTATION

The BLAS API of BLASFEO has the same aim as the BLASFEO API: providing a linear algebra library performance-optimized for matrices fitting in cache. Therefore, at the high level many implementation choices are unchanged: lack of cache blocking, ordering of loops, and tailoring of custom assembly kernels for each linear algebra routine.

As discussed in Section 2.2, the main difference is the packing of data: in the BLASFEO API this is split from the linear algebra routines, while in the BLAS API this has to happen within the linear algebra routines.

3.1 Implementation of BLAS and LAPACK Routines

As a general rule, in the BLAS API of BLASFEO all level 3 BLAS and LAPACK routines are implemented using the algorithmic variants "C" (for small matrices) and "B" (for larger matrices) described in Section 2.2. The exact switching point depends on the specific routine and computer architecture, but typically in the case of square matrices it is for matrix sizes in the range 64–400.

This section describes the details of some implementation choices for selected BLAS and LAPACK routines in the BLAS API of BLASFEO.

3.1.1 dgemm. The implementation of dgemm was discussed in detail in Section 2.

3.1.2 dsyrk. In the case of dsyrk, the A and B matrices of dgemm coincide. However, generally, in the dsyrk implementation in optimized BLAS libraries, the memory buffers for A_k and for B_k cannot coincide, since in the case of non-square kernels they are packed with different panel sizes.

Conversely, in the BLASFEO framework, the panel sizes for A_k and for B_k are equal even for non-square kernels, since this is a requirement to make the panel-major the default matrix format for all BLASFEO API routines. Therefore, in the implementation of dsyrk, the same memory buffer is employed for A_k and B_k. For the "B" algorithmic variant, the A matrix is packed at once into B_k, and then A_k simply points to a panel in B_k, without re-packing it. This reduces the amount of necessary packing by a factor 2. For the "C" algorithmic variant, the kernels used to compute the diagonal blocks have both A_k and B_k (which coincide) available in panel-major, and therefore use the faster ppc kernel variants. This gives the BLASFEO framework an inherent advantage over other optimized implementations of dsyrk.

3.1.3 dtrsm. In the BLASFEO framework, the dtrsm routines are implemented using custom kernels, which consist of the dgemm loop followed by a custom and fully unrolled sub-routine performing the solution of a fixed-size register-resident triangular system. In the implementation of these kernels, there is a fundamental difference between left and right variants of the dtrsm routine. Before proceeding, it is useful to note that in BLASFEO optimal kernels are generally square or tall, meaning that $m_r \geq n_r$, where $m_r \times n_r$ is the kernel size (i.e., the size of the result sub-matrix computed by the kernel).

Left variants of the dtrsm kernel require one to solve a few (n_r) large ($m_r \times m_r$) triangular systems of equations, each computing an independent $m_r \times 1$ sub-column of D. The implementation of this operation is not vector friendly, as the vectorization happens in the computation of a single sub-column of D. The triangular matrix is the left factor and therefore accessed with vector loads, requiring masking at the edges and performing useless flops. Furthermore, the solution of larger $m_r \times m_r$ triangular systems requires bigger and more complex custom routines, and it implies that less flops are processed by the efficient dgemm loop and more by these custom routines.

Right variants of the dtrsm kernel require one to solve many (m_r) small ($n_r \times n_r$) triangular systems of equations, each computing an independent $1 \times n_r$ sub-row of D. The implementation of this operation is vector friendly. The computation of a single sub-row of D is not vectorized, but n_r of such sub-rows can fit in a vector register (where n_r is the size of the vector register) and be processed in parallel using vector instructions. The triangular matrix is at the right factor and its elements are broadcast, and therefore no useless flops or memory loads need to be performed. Furthermore, the solution of smaller $n_r \times n_r$ triangular systems requires smaller and simpler routines, and more flops are processed by the more efficient dgemm loop.

Given the above considerations, which get increasingly important as the SIMD width increases, in BLASFEO only kernels tailored to the right variants of dtrsm are implemented. The left variants dtrsm are implemented using transposition and the kernels for the right dtrsm variant. For example, the triangular system solution $A^{-1}B$ (with A a triangular matrix) is implemented as $(B^T A^{-T})^T$, with the transposition of B performed at no extra cost, since it has to be packed anyway being now the left factor.

3.1.4 dtrmm. In the implementation of dtrmm, considerations analogous to dtrsm apply.

3.1.5 dpotrf. In the implementation of dpotrf, considerations analogous to dsyrk apply. In fact, also in this case the symmetry of the operation can be exploited to reduce the amount of required data packing. This gives to the BLASFEO framework an inherent advantage over other optimized implementations of dpotrf.

3.1.6 dgetrf. The implementation of the dgetrf routine is rather different compared to other routines in BLASFEO. At the high level, it is computed by column-blocks, as opposed to row-blocks of the other routines. Internally, however, the column-blocks stored in column-major are packed and transposed into row-blocks stored in panel-major, which is the format used for computing. This enables using the more efficient dgemm_nt kernel, and to perform triangular system solutions with the triangular matrix as the right factor. Furthermore, in the case of tall kernels this enables reusing the same L1-resident tall factor across calls to the dtrsm and dgemm kernels operating at the same iteration of the outer loop.

3.2 Custom Function Calling Convention: Modular Assembly Kernels

The linear algebra implementation as proposed in Sections 2.4 and 3.1 requires the implementation of a large number of custom kernels, which generally share a large fraction of the code. In BLASFEO, these kernels are written in assembly (with the exception of the generic target, coded in C). The flexibility of the assembly language is exploited to code the kernels in a modular fashion enhancing code reuse, and yet attaining high performance.

3.2.1 Modular Kernel Design. In the BLASFEO framework, a kernel is a function with global scope, implementing the innermost loop of the linear algebra routines, and operating on a sub-matrix of the result matrix D_k. The size $m_r \times n_r$ of the kernel is the size of the sub-matrix of the result matrix D_k computed by the kernel itself.

A kernel algorithm is split into more elementary operations, each of which is coded in a separate function (named inner function) with module scope. Therefore, the kernels themselves do not contain code for any numerical operation, and they are simply implemented as a sequence of calls to inner functions. Typically, the same inner function is called by many different kernels, ensuring a high degree of code reuse. For example, the BLASFEO API kernel for the "NT" version of dgemm,

and the BLAS API ppc kernel for the "L" version of `dpotrf` are decomposed into inner functions as

```
GLOB_FUN_START(kernel_dgemm_nt_4x4_lib4444)
PROLOGUE

ZERO_ACC

// set some GP registers ...
CALL(inner_kernel_dgemm_nt_4x4_lib44)

// set some GP registers ...
CALL(inner_scale_ab_4x4_lib4)

// set some GP registers ...
CALL(inner_store_4x4_lib4)

EPILOGUE
RETURN
GLOB_FUN_END(kernel_dgemm_nt_4x4_lib4444)



GLOB_FUN_START(kernel_dpotrf_nt_l_4x4_lib44cc)
PROLOGUE

ZERO_ACC

// set some GP registers ...
CALL(inner_kernel_dgemm_nt_4x4_lib44)

// set some GP registers ...
CALL(inner_scale_m11_4x4_libc)

// set some GP registers ...
CALL(inner_edge_dpotrf_4x4_libc)

// set some GP registers ...
CALL(inner_store_l_4x4_libc)

EPILOGUE
RETURN
GLOB_FUN_END(kernel_dpotrf_nt_l_4x4_lib44cc)
```

Here, each "4" after "lib" means that the corresponding matrix argument is in panel-major layout with panel size $p_s = 4$, while each "c" after "lib" means that the corresponding matrix argument is in column-major layout. For example, `inner_kernel_dgemm_nt_4x4_lib44` takes for its two matrix arguments A_k and B_k two panels of size $p_s = 4$, while `inner_store_l_4x4_libc` stores

the lower triangular of a 4×4 sub-matrix in column-major format. Note that the two functions share the same innermost loop, coded in the inner function `inner_kernel_dgemm_nt_4x4_lib44`.

Generally, in the standard function calling conventions it is not possible to pass data in FP vector registers. Therefore, if the proposed approach is coded using C code, similarly to the ν-BLACs in `LGen` [22] and `SLinGen` [23], the inner functions need to be inlined, otherwise the overhead would be unacceptably large. Even in the case where all inner functions are inlined, the compilers seem to struggle to find a globally optimal register allocation strategy across the numerous inlined functions. In any case, inlining makes only the source code modular, but not the object code, whose size grows considerably.

The approach proposed in BLASFEO employs assembly code instead. The inner functions have module scope, and therefore they do not need to follow the standard function calling convention. In particular, some FP registers are designated to pass numerical data between several inner functions in a consistent way. Similarly, some general purpose (GP) registers are designated to pass arguments (like loop sizes and pointers) from the kernels to the inner functions. These registers are different from the registers used to pass arguments in the standard function calling convention (which remain unmodified), ensuring that inner functions can be called in any order.

In this way, both modularity and performance are attained. Overhead is kept at a minimum, as it only consists of setting some GP registers and performing some unconditional jumps for the inner function calls. Optionally, BLASFEO also gives the possibility to treat the entire inner functions as macros and inline them, removing these unconditional jumps at the expense of an increase in object code size.

In BLASFEO, the `dgemm` inner kernels are the backbone of the kernels for all level 3 BLAS and LAPACK routines. For performance reasons, these inner kernels are only implemented for the "NN" and "NT" `dgemm` variants, where the left factor is non-transposed. Conversely, and recalling that matrix elements are stored contiguously on the same column in both the considered column-major and panel-major matrix formats, when the left factor is transposed then contiguous matrix elements on the same column are employed at different iterations of the inner loop over k. The vectorization of such "TN" and "TT" `dgemm` inner kernels is intrinsically less efficient since it is dot-product based: it uses register space less effectively (it requires an entire vector register for each kernel matrix element, to hold a partial dot-product in each vector element); it requires a final reduction step (to sum up the partial dot-products across vector elements); it needs larger values of k to perform well (and the value increases with the vector width); it has a smaller degree of reuse of factor matrix elements in registers (since the kernel size is smaller, and this is especially important since BLASFEO does not employ cache blocking). As a replacement for the functionality of such inner kernels, BLASFEO employs only the more efficient "NN" and "NT" `dgemm` inner kernels together with the explicit transposition of the left A_k or right B_k factors, or with the application of the matrix multiplication transposition rule $(A \cdot B)^T = (B^T \cdot A^T)$ to the register-resident sub-matrix of the result matrix computed by the kernel.

3.2.2 Custom Function Calling Convention for Inner Functions. This section describes the custom calling convention used for the inner functions in BLASFEO for the currently supported targets. For compactness reasons, the usage of FP registers is only described for real double precision kernels.

Note that in the BLASFEO kernels, integer numbers are passed by value and FP numbers by reference. Therefore, BLASFEO kernels only have arguments of integer type (integer numbers or pointers), and do not employ the FP part of the standard function calling convention.

x86_64. In `x86_64` architectures, the standard function calling convention is different in Linux and macOS (where the first six integer function arguments are passed in the GP registers `rdi`, `rsi`,

rdx, rcx, r8, and r9, and the remaining on the stack), and Windows (where the first four integer function arguments are passed in the GP registers rdx, rcx, r8, and r9, and the remaining on the stack).

The x86_64 architecture has 16 64-bit GP registers. In BLASFEO, the registers rdi, rsi, rdx, rcx, r8, and r9 are reserved for the arguments of the kernel function, and never modified; the register esp is reserved for the stack pointer. The registers r10, r11, r12, r13, r14, r15, rax, rbx, and rbp are available for the BLASFEO custom function calling convention: they are used, in this order, to pass integer and pointer arguments to the inner functions, and for internal computations within the inner functions. As an example, the inner function inner_kernel_dgemm_nt_4x4_lib44 takes the arguments

```
// r10d <= int k_max
// r11 <= double *A_k
// r12 <= double *B_k
```

The 64-bit x86_64 architecture with SSE and AVX ISAs (Instruction Set Architectures) has 16 FP registers. The number of FP registers used as accumulator registers depends on the kernel size and on the instruction set. In double precision, in the case of the AVX 256-bit registers, the accumulation registers are the registers ymm0-ymm3 for the 4×4 kernels, the registers ymm0-ymm7 for the 8×4 kernels, and the registers ymm0-ymm11 for the 12×4 kernels. In the case of the SSE 128-bit registers, they are the registers xmm0-xmm7 for the 4×4 kernels. The remaining FP registers can be freely used for internal computations in the inner functions.

x86. Currently, for the 32-bit x86 architecture BLASFEO has only been ported to Linux with gcc and clang compilers. In this case, the standard function calling convention is to pass all function arguments on the stack.

The x86 architecture has eight 32-bit GP registers. The register esp is reserved for the stack pointer. The remaining seven registers eax, ebx, ecx, edx, esi, edi, and ebp are used in the BLASFEO custom function calling convention.

The x86 architecture with SSE and AVX ISAs has eight FP registers. At most four can be used as accumulator registers: the 128-bit registers xmm0-xmm3 for the 4×2 kernels targeting SSE ISAs and the 256-bit registers ymm0-ymm3 for the 4×4 kernels targeting the AVX ISAs.

ARMv8A. Currently, for the 64-bit ARMv8A architecture BLASFEO has only been ported to Linux with gcc and clang compilers. The standard function calling convention is to pass the first eight integer function arguments in the GP registers x0-x7 (which in the case of 32-bit arguments are named w0-w7) and the remaining on the stack.

The ARMv8A architecture has 30 GP registers (x0-x29), one procedure link register (x30), and one zero register (x31). In BLASFEO, the registers x0-x7 are reserved for the arguments of the kernel function, and never modified; the registers x30 and x31 are not employed. The remaining registers x8-x29 are used in the BLASFEO custom function calling convention.

The ARMv8A architecture with NEONv2 ISA has 32 128-bit FP registers. The FP registers used as accumulator registers are the registers q0-q7 for the 4×4 kernels, the registers q0-q15 for the 8×4 kernels, and the registers q0-q23 for the 12×4 kernels.

ARMv7A. Currently, for the 32-bit ARMv7A architecture, BLASFEO has only been ported to Linux with gcc and clang compilers. The standard function calling convention is to pass the first four integer function arguments in the GP registers r0-r3 and the remaining on the stack.

The ARMv7A architecture has 13 GP registers (r0-r12), a stack pointer register (r13), a link register (r14), and a program counter register (r15). In BLASFEO, the registers r0-r3 are reserved for

the arguments of the kernel function and never modified; the registers r13-r15 are not employed. The registers r4-r12 are used in the BLASFEO custom function calling convention.

The ARMv7A architecture has two extensions for FP computations, VFP (offering single and double precision scalar FP instructions), and NEON (offering single precision SIMD FP instructions). VFP comes in two versions, offering 32 or 16 64-bit FP registers. In BLASFEO, only the more common version with 32 FP registers is targeted. In double precision, the FP registers used as accumulator registers are the registers d0-d15 for the 4×4 kernels.

4 PERFORMANCE EVALUATION

In this section, the performance of the BLAS API of BLASFEO is evaluated against other optimized BLAS libraries. Namely, we consider the open source implementations OpenBLAS version 0.3.4-dev, BLIS version 0.5.0 (coupled with LAPACK 3.5.0 to provide the dpotrf and dgetrf routines), and the proprietary implementation MKL version 2019.1.144. Additionally, the performance of the BLASFEO API is added as a reference, even if this does not implement the standard BLAS API. The sequential version of all libraries is employed.[6] All open source libraries are compiled with standard options. MKL is used with the MKL_DIRECT_CALL_SEQ option. The same setup is used for all experiments in the article.

It should be stressed that these comparisons are limited to library implementations of linear algebra routines. Therefore they do not consider program generated, just-in-time (JIT) compiled or template-based implementations like, for example, LIBXSMM, Eigen, Blaze, LGen, and SLinGen. The interested reader can find some comparisons also involving these approaches in the original BLASFEO article [9].

The performance of the linear algebra routines is evaluated for matrices of size up to 300, as this is large enough for the embedded optimization applications of interest, and it is a good size to show the behavior of the routines. In fact, since BLASFEO does not implement cache blocking, 300 as a matrix size is large enough to show some performance degradation when the data footprint exceeds the cache size. Matrix sizes are not known at compile time and all experiments are performed with hot cache.[7]

The performance is evaluated on three microarchitectures: Intel Haswell,[8] ARM Cortex A57, and ARM Cortex A53. Intel Haswell is a high-performance microarchitecture commonly employed in laptops, workstations, and servers, and high-end embedded devices. The ARM microarchitectures are commonly employed in mobile and embedded devices, with a focus on performance as well as cost and power consumption. In particular, the ARM Cortex A53 is a rather small and cheap core, and it supports in-order execution.

For all microarchitectures, the same six routines are evaluated. The dgemm routine is evaluated in the variants "NN" and "NT."[9] The "UT" (upper-transposed) variant of the dsyrk routine makes use of the "NN" inner kernel in the BLASFEO API, and the "NT" inner kernel in the BLAS API. Therefore, this is an example of a routine performing similarly in the two APIs, since the faster kernel in the BLAS API offsets the lower overhead of the BLASFEO API. The "RLTU" (right-lower-transposed-unit_diagonal) variant of the dtrsm routine makes use of the "NT" inner kernel for both

[6]Generally, the multi-threaded version with only one thread enabled has worse performance due to the additional overhead of the parallelization framework.

[7]This is generally the case in embedded optimization, where optimization problems are repeatedly solved at each sampling time by using iterative algorithms.

[8]The exact same optimized code is employed also on the client version of the more recent Intel Skylake microarchitecture, with analogous results.

[9]In BLASFEO, the kernels employed in these routines are the backbone for all algorithmic variants of the BLAS API, as well as the BLASFEO API.

the BLAS API and the BLASFEO API. Therefore, this is an example of a routine performing better in the BLASFEO API. Finally, the Cholesky and the row-pivot LU factorizations (implemented in `dpotrf` ("L" variant) and `dgetrf`, respectively) are two routines widely employed in the solution of linear systems of equations, and their performance is critical in many embedded optimization applications.

4.1 `x86_64` Intel Haswell

Intel Haswell is a deeply out-of-order microarchitecture that performs aggressive hardware prefetch. Regarding the implementation of linear algebra routines, the Haswell core can perform two 256-bit-wide FP fused-multiplication-accumulate every clock cycle, giving a throughput of 16 flops per cycle in double precision. The client version of Intel Skylake, the Haswell successor microarchitecture, does not show significant differences in this regard, and the same optimized code is employed.

It is relatively easy to write `dgemm` kernels achieving high performance provided that at least 10 accumulation registers are employed. In the implementation of BLASFEO, the panel size p_s is 4 in double precision. The optimal `dgemm` kernel size is 12×4. Hardware prefetch can detect the streaming of data along panels or along columns.

The test processor is the Intel Core i7 4810MQ, a quad-core Haswell processor running at 3.4 GHz when the 256-bit execution units are employed (3.8 GHz when they are idle). The maximum double-precision throughput per core is 54.4 Gflops. The memory is 8 GiB of DDR3L-1600 RAM in dual-channel configuration, giving a bandwidth of 25.6 GB/s. Each core has 32 KB 8-way set associative data and 32 KB 8-way set associative instruction L1 caches, and 256 KB 8-way set associative L2 cache. All cores share 6 MB 12-way set associative L3 cache. The cache line size is 64 bytes.

Performance plots are in Figure 2. For the matrix sizes of interest, the BLAS API of BLASFEO is competitive with Intel MKL in the implementation of `dgemm`, and it outperforms MKL for the other level 3 BLAS and especially for LAPACK routines. It also outperforms the other open source implementations OpenBLAS and BLIS for all tested routines, and by a factor of 2 or 3 for matrix sizes in the order of tens. The use of the BLASFEO API gives an additional 10–15% speedup in most routines (except `dsyrk_ut`).

The drop in performance visible especially for the BLASFEO API implementation of `dgemm` is due to the fact that four panels (three for A_k and one for B_k simultaneously streamed by the `dgemm` kernel) of height $p_s = 4$ exceed the L1 data cache size for $n \geq 256$.

4.2 `ARMv8A` ARM Cortex A57

The ARM Cortex A57 is a relatively low-power microarchitecture, and it is the 64-bit successor of the ARM Cortex A15. It is a 3-way superscalar microarchitecture with out-of-order execution. The NEON ISA in the ARMv8A architecture supports vectorization in both single and double precision, with four- and two-wide vectors, respectively. Regarding the implementation of linear algebra routines, the Cortex A57 core can perform one 128-bit-wide FP fused-multiplication-accumulate at every clock cycle, giving a throughput of eight and four flops per cycle in single and double precision, respectively. The Cortex A72 and A75, the successor microarchitectures, do not show a significant difference in this regard, and the same optimized code is employed.

In the implementation of BLASFEO, the panel size p_s is four in double precision. The optimal `dgemm` kernel size is 8×4. Software prefetch is employed for both the left and the right factors in all matrix formats, slightly improving performance.

The test processor is the NVIDIA Tegra TX1 SoC in the Shield TV featuring four Cortex A57 cores, each paired with one of the four low-power Cortex A53 cores. The tests are performed

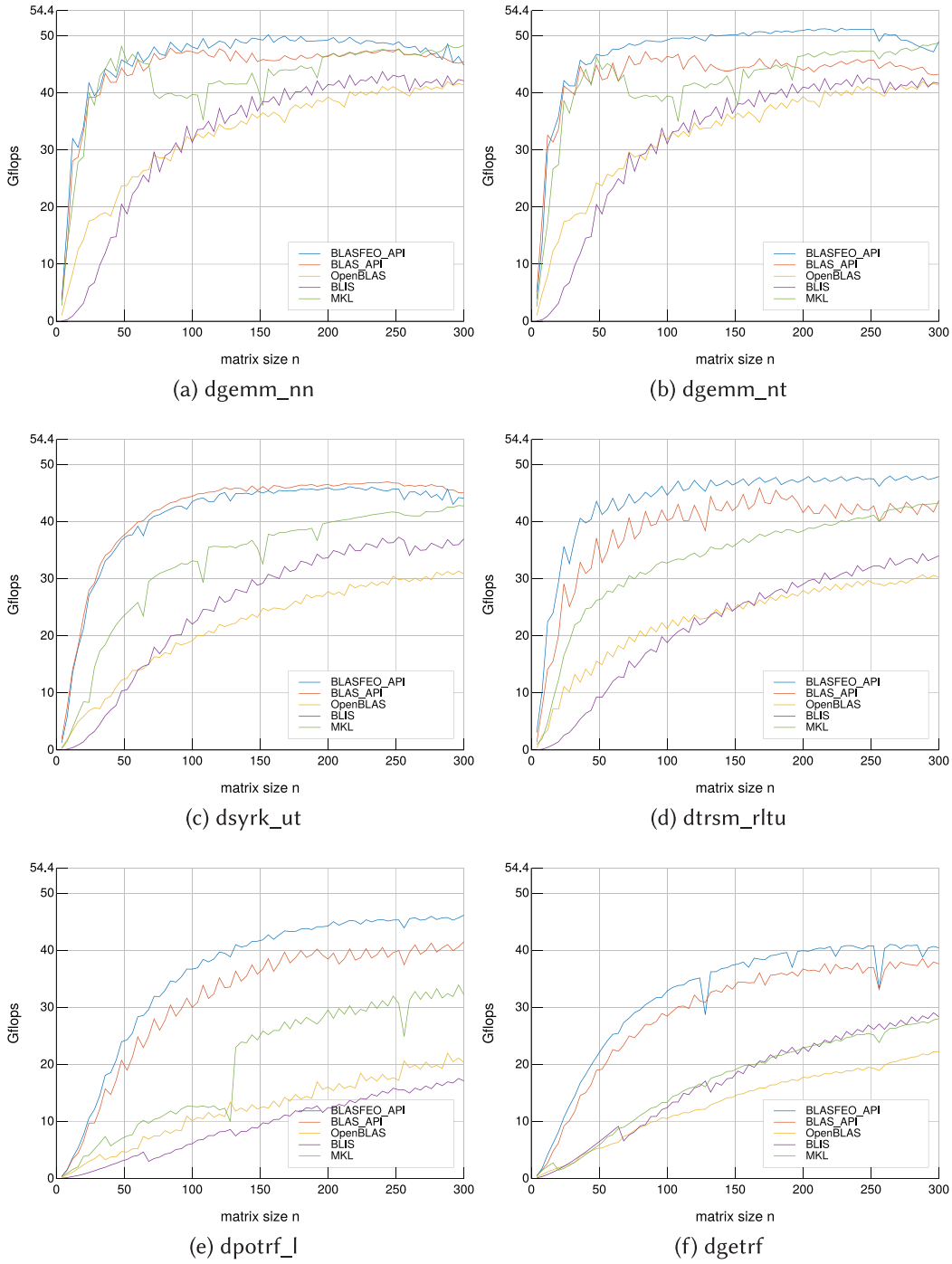


Fig. 2. Performance of some BLAS and LAPACK routines on one core of the Intel Haswell Core i7 4810MQ @ 3.4 GHz.

on a single Cortex A57 core, running at 2.15 GHz during all tests. The maximum double-precision throughput per core is 8.6 Gflops. The memory interface is 64-bit LPDDR4-3200 giving 25.6 GB/s of bandwidth. The amount of memory is 3 GiB. Each Cortex A57 core has 48 KB 3-way set associative instruction L1 cache and 32 KB 2-way set associative data L1 cache. All Cortex A57 cores share a 2 MB 16-way set associative unified L2 cache. The cache line size is 64 bytes.

Performance plots are in Figure 3. In the case of dgemm, the BLAS API of BLASFEO is competitive or outperforms OpenBLAS (whose performance shows significant drops for some sizes) and BLIS (which shows considerable overhead for small matrices). The performance advantage for the other level 3 BLAS and especially LAPACK routines is larger, and in the range of a speedup factor of 2 for matrix sizes in the order of tens. The use of the BLASFEO API gives an additional 10–15% speedup in most routines.

The drop in performance visible especially for the BLASFEO API implementation of dgemm is due to the fact that three square matrices (A, B, and C) of doubles exceed the L2 cache size for $n \geq 295$ (with cache associativity and share of cache between data and instructions slightly lowering this value). The BLAS API employs more memory (in particular, the matrix B is entirely packed at once into a buffer), and therefore the drop happens for smaller matrix sizes.

4.3 ARMv8A ARM Cortex A53

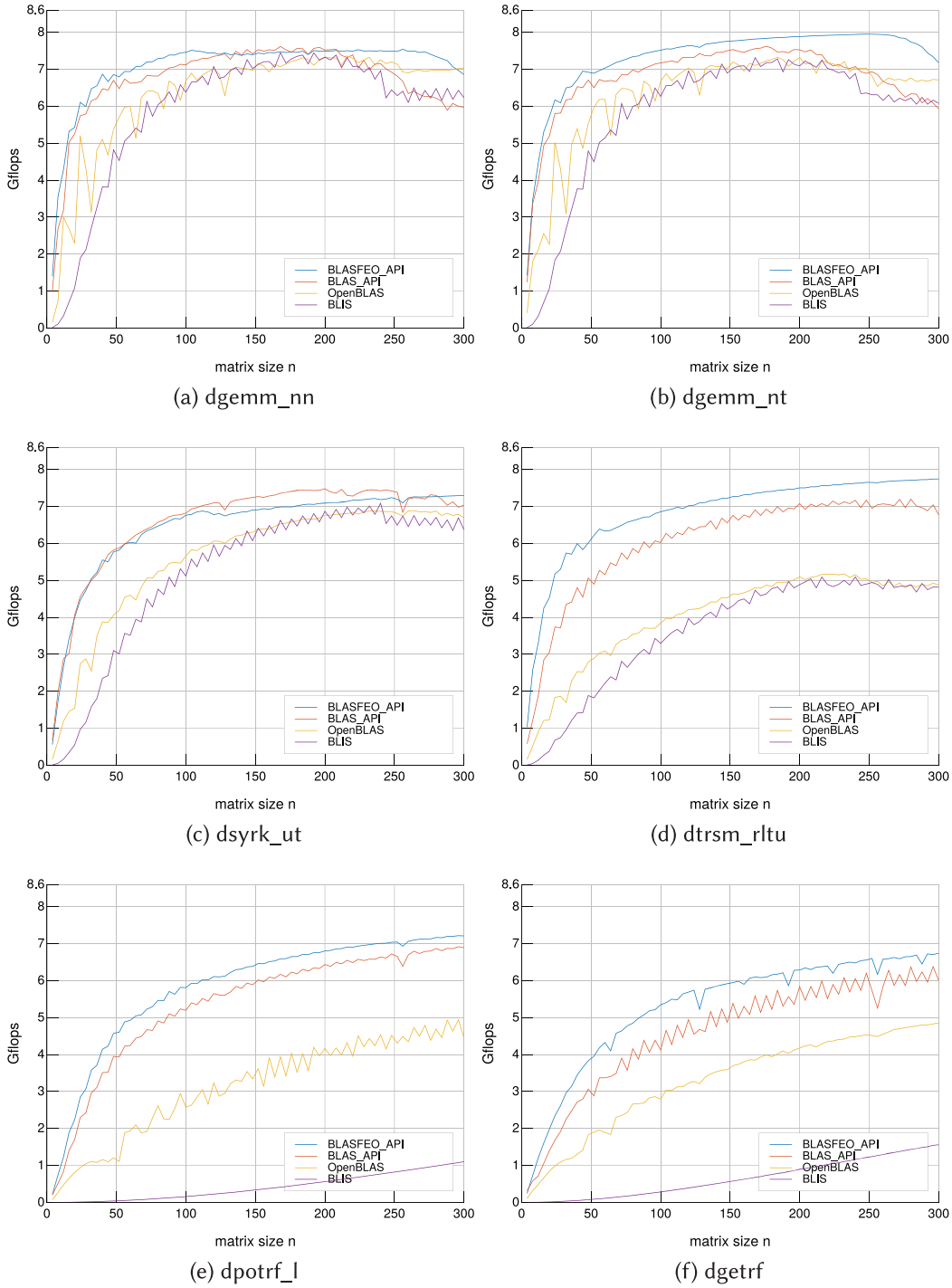
The ARM Cortex A53 is a low-power core, intended to be used alone or as the little companion core of the big cores like Cortex A57 or A72. It can be considered the 64-bit successor of the ARM Cortex A7. It is a partially 2-way superscalar microarchitecture (only certain combinations of instructions can be dual-issued) with in-order execution. The Cortex A53 is ISA-compatible with the Cortex A57, and therefore it also supports the NEON ISA. Regarding the implementation of linear algebra routines, the Cortex A53 core can perform one 128-bit-wide FP fused-multiplication-accumulate at every clock cycle, giving a throughput of eight and four flops per cycle in single and double precision, respectively (same as the Cortex A57).

In the implementation of BLASFEO, the panel size p_s is 4 in double precision. The optimal dgemm kernel size is 12×4. The low-power features of the ARM Cortex A53 (in-order execution, partial dual issue, small number of execution ports) make it difficult to write high-performing code, and, for example, the code optimized for the Cortex A57 performs poorly. In particular, as described in [10] a 64-bit FP load and an FMA cannot be co-issued, and no 64-bit FP load can be performed on the fourth cycle after any FMA; 64-bit loads to GP registers can be co-issued with FMA, and loads of more than 64-bit cannot be issued in one single cycle. Therefore, an optimal scheme alternates three FMAs and one 64-bit FP load, with the remaining data loaded using 64-bit loads in GP registers and data transfers between GP and FP registers. The maximum attainable performance with this scheme is three 128-bit FMAs every four clock cycles, which is 75% of peak FMA throughput.

The test processor is the Amlogic S905 SoC in the ODROID-C2 board, featuring four Cortex A53 cores. The tests are performed on a single Cortex A53 core, running at 1.536 GHz during all tests. The maximum double-precision throughput per core is 6.144 Gflops. The memory interface is 32-bit DDR3-1824 giving 7.296 GB/s of bandwidth. The amount of memory is 2 GiB. Each Cortex A53 core has 32 KB 2-way set associative instruction L1 cache and 32 KB 4-way set associative data L1 cache. All cores share 512 KB 16-way set associative unified L2 cache. The cache line size is 64 bytes.

Performance plots are in Figure 4. In the case of BLIS, the code for the Cortex A53 target in version 0.5.0 terminates with a segfault,[10] so the code for the Cortex A57 had to be used instead. Note that, in BLIS, the Cortex A53 and A57 targets share the same kernel code, and only differ

[10]The issue is fixed in later version 0.6.0.

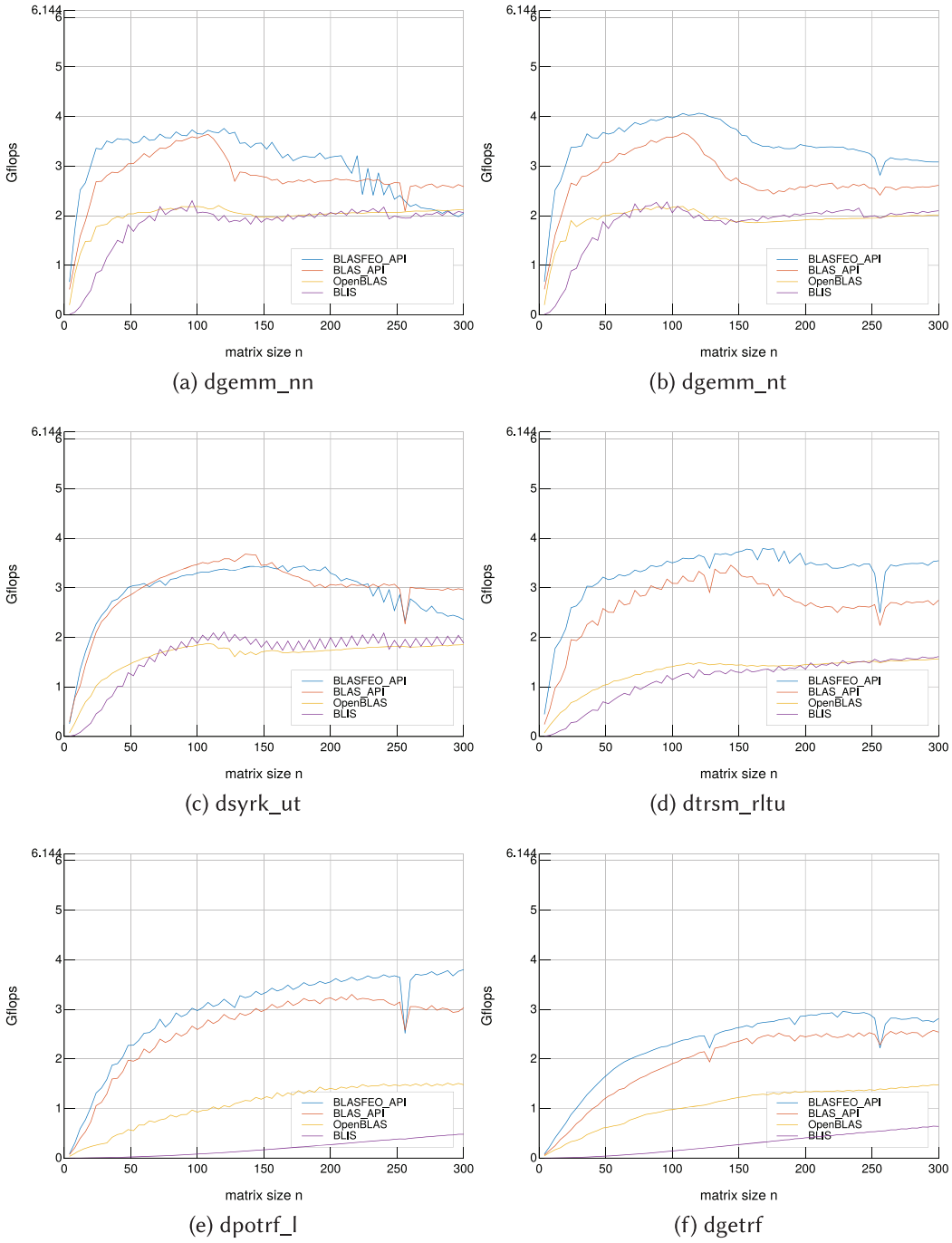


(a) dgemm_nn (b) dgemm_nt

(c) dsyrk_ut (d) dtrsm_rltu

(e) dpotrf_l (f) dgetrf

Fig. 3. Performance of some BLAS and LAPACK routines on one core of the ARM Cortex A57 @ 2.15 GHz.



(a) dgemm_nn (b) dgemm_nt

(c) dsyrk_ut (d) dtrsm_rltu

(e) dpotrf_l (f) dgetrf

Fig. 4. Performance of some BLAS and LAPACK routines on one core of the ARM Cortex A53 @ 1.536 GHz.

by the choice of compilation flags. This implies that there is not a large performance difference between the two targets. The first thing worth noticing is that, in general, BLASFEO outperforms both OpenBLAS and BLIS by a wide margin due to its better-performing assembly kernels. The speedup is considerably larger than in the case of the Cortex A57, in which case the speedup is solely due to the low overhead of the BLASFEO framework. Compared to the BLAS API of BLASFEO, the use of the BLASFEO API gives an additional 10–20% speedup in most routines.

As in the case of the Cortex A57, the drop in performance for the BLASFEO API implementations of `dgemm` is due to the fact that three square matrices of doubles exceed the L2 cache size for $n \geq 147$ (again slightly lowered due to cache associativity and share of L2 cache between data and instruction). The drop happens for smaller matrix sizes in the case of the BLAS API since this employs more memory internally. The performance of the `dgemm` kernel itself exceeds 70% of peak FMA throughput, therefore getting very close to the 75% maximally attainable by the scheme.

5 APPLICATION IN SCIENTIFIC PROGRAMMING LANGUAGES

The BLAS API of BLASFEO currently supports a sub-set of the entire BLAS and LAPACK libraries, focusing on the routines with the largest impact on the intended application areas. The main interest is in real double precision routines, particularly various types of matrix-matrix multiplications and solution of linear systems of equations. Complex routines are currently not of interest. Therefore, the BLAS API of BLASFEO does not aim at replacing existing BLAS implementations, but at complementing them, improving the performance of selected routines for small matrices. This can be obtained even without any modification to the existing BLAS implementations.

In the case of compiling a program from source, this can be achieved by simply choosing the linking order of libraries, placing the BLASFEO library before the other BLAS and LAPACK libraries. In this way, the linker looks for the symbols first in BLASFEO and then it looks for the unresolved symbols in the BLAS and LAPACK libraries.

In the case of an existing program making use of dynamic libraries (as, e.g., programs installed with package managers in many Linux distributions, looking for libraries in the standard search directories), the Unix command `LD_PRELOAD` can be employed to resolve all symbols in the BLASFEO dynamic library first, and then to resolve any missing symbol in the default libraries. This `LD_PRELOAD` trick is employed in the experiments with Octave and SciPy presented next.

5.1 Performance Plots

This section compares the computational performance of some key linear algebra routines as employed in the scientific programming languages Octave, Python SciPy, and Julia,[11] and with native calls to the BLAS or LAPACK routines from C code. More detailed information about these scientific programming languages and their use of BLAS and LAPACK routines is reported in Section D in the Appendix. The performance of the BLAS API of BLASFEO is evaluated against OpenBLAS, as this is currently the best open source implementation for the matrix sizes of interest, and generally the default BLAS and LAPACK versions in open source scientific software on Linux distributions. The results are presented in Figure 5.

The first fact worth noticing is that both Octave and (to a smaller extent) SciPy show some noticeable overhead. On the other hand, once the JIT compilation is terminated, the performance plots for Julia are almost indistinguishable from the ones for the native calls from C. Since for the matrix sizes of interest BLASFEO routines are faster than the OpenBLAS counterparts, the impact

[11]For each language, when multiple commands call the same BLAS or LAPACK routine, the one with less overhead is employed.

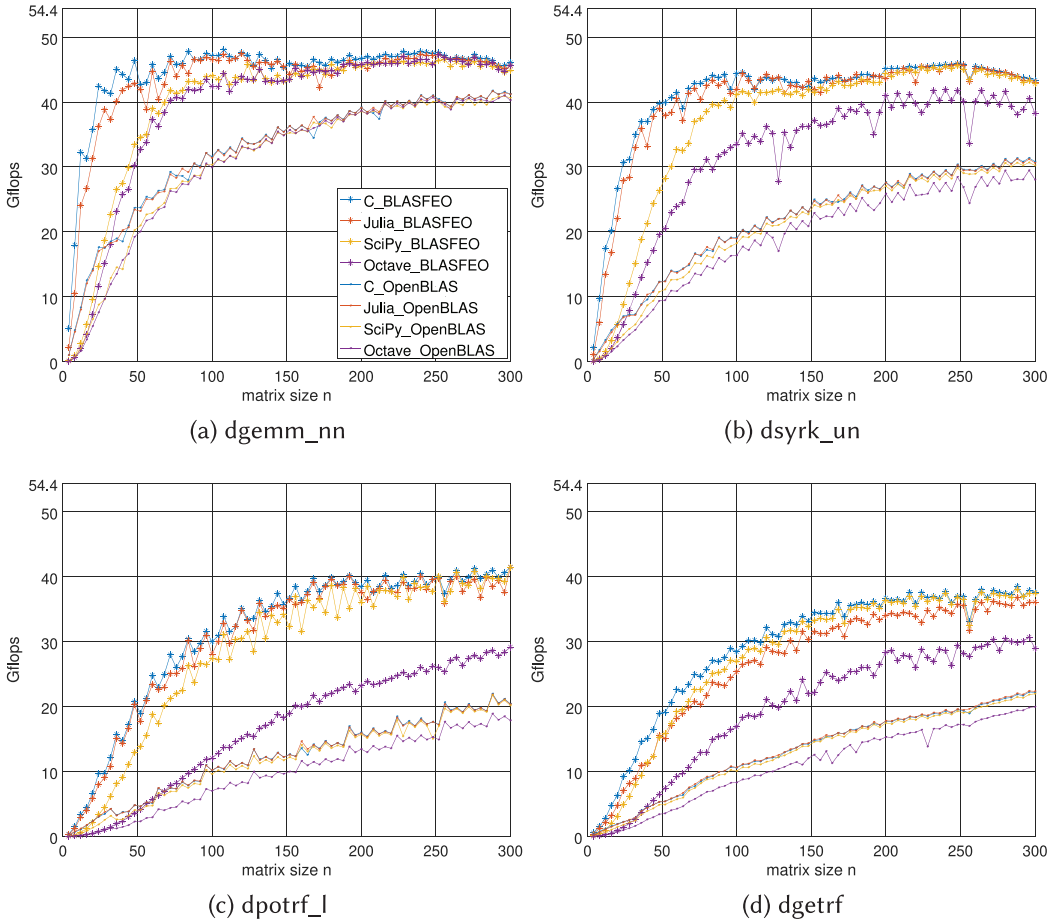


Fig. 5. Performance of some BLAS or LAPACK routines in Octave 4.2.2, Python SciPy 1.2, Julia 1.0.1, and native calls in C, on Intel Haswell Core i7 4810MQ @ 3.4 GHz. BLAS and LAPACK provided by single-threaded versions of either the BLAS API of BLASFEO or OpenBLAS 0.3.4.dev.

of this overhead on performance is much more visible in the case of BLASFEO. Nonetheless, the use of BLASFEO gives a significant speedup for all languages, especially for rather small matrices of size up to 100 where the speedup ranges from about 50% in the case of `dgemm`, up to doubling or more the performance in the case of the `dpotrf` and `dgetrf` factorizations.

In more detail, the overhead introduced by Octave is relatively small in the case of `dgemm`, but it affects performance more in the case of `dsyrk` (since the additional operation of copying the upper triangular part in the lower triangular part of the result matrix is performed) and in the case of the factorizations `dpotrf` and `dgetrf` (since additional operations are performed, like the copy in a new matrix before factorization and computation of the condition number after factorization). In particular, all matrix copies are implemented with double nested loops in C++, and therefore they do not take advantage of optimized and vectorized level 1 BLAS routines. Conversely, the low-level SciPy wrappers to BLAS and LAPACK routines show considerably smaller overhead, which affects performance only for matrix sizes up to a few tens. The Julia wrappers are JIT compiled and show almost unnoticeable overhead.

5.2 Riccati Recursion

This section compares the running time of the Riccati recursion algorithm as implemented using BLAS and LAPACK routines through different programming languages, and it investigates to what extent the choice of the BLAS and LAPACK implementation affects performance.

The Riccati recursion algorithm is widely used in control applications, and, for example, it is encountered in the computation of the optimal state feedback in finite horizon linear quadratic regulator (LQR) or in the Kalman filter. It can also be used to efficiently factorize the Karush-Kuhn-Tucker (KKT) matrix arising in constrained optimal control problems. The backward Riccati recursion reads

$$P_n = Q + AP_{n+1}A^T - (S + AP_{n+1}B^T)(R + BP_{n+1}B^T)^{-1}(S^T + BP_{n+1}A^T),$$

where the matrices P_{n+1} and $\begin{bmatrix} R & S \\ S^T & Q \end{bmatrix}$ are assumed to be symmetric positive definite. The matrices A, Q, P have size $n_x \times n_x$, the matrices B and S^T have size $n_x \times n_u$, and the matrix R has size $n_u \times n_u$, where n_x is the number of states and n_u is the number of controls in the dynamical system. In finite horizon LQR or in the factorization of the KKT matrix in optimal control problems, the recursion is repeated N times, where N is the control horizon length. In the Kalman filter, one recursion step is performed at each sampling instant.

The Riccati recursion can be implemented efficiently as [7]

$$C \leftarrow \begin{bmatrix} B^T \\ A^T \end{bmatrix} \cdot \mathcal{L}_{n+1},$$

$$\begin{bmatrix} \Lambda_n & 0 \\ L_n & \mathcal{L}_n \end{bmatrix} \leftarrow \left(\begin{bmatrix} R & S \\ S^T & Q \end{bmatrix} + C \cdot C^T \right)^{\frac{1}{2}},$$

where $\mathcal{L}_n$ is the lower Cholesky factor of P_n and the exponent $\frac{1}{2}$ denotes the lower triangular Cholesky factorization. The algorithm can be implemented using the `dtrmm_rlnn` and `dsyrk_ln` BLAS routines and the `dpotrf_l` LAPACK routine. All these routines are available natively in C and through thin wrappers in Julia 1.0.1 and SciPy 1.2 (after the fix to `dtrmm`, see Section D.2 in the Appendix). The `dtrmm` routine is unused in Octave 4.2.2. If not fixed, the `dtrmm` routine in SciPy 1.2 gives the wrong result and it needs to be substituted with `dgemm` in the Riccati recursion implementation.

Table 2 contains the solution time for the Riccati recursion implemented using native BLAS and LAPACK calls in C and through wrappers in Octave 4.2.2, SciPy 1.2 and Julia 1.0.1. The BLAS and LAPACK routines are provided by either the BLAS API of BLASFEO, OpenBLAS 0.3.4.dev, or MKL 2019.1.144. The performance of the Riccati recursion implemented using the BLASFEO API is added as an additional comparison. Figure 6 presents graphical representations of the data in Table 2.

In the case of Octave and SciPy, the choice of the BLAS and LAPACK implementation barely affects performance for the smallest example (with matrix sizes in the order of the units), as the overhead of these interpreted scripting languages dominates the computation time. For the other examples (with matrix sizes in the order of tens), the BLAS and LAPACK implementation has a significant effect on the performance, and the BLAS API of BLASFEO gives a speedup ranging between 1.5 and 2 times over OpenBLAS and between 1.2 and 1.4 times over MKL. Julia employs JIT compilation and therefore, once the Riccati recursion module is compiled, the overhead is several times lower than interpreted scripting languages, and also in the smallest example the BLAS API of BLASFEO gives a speedup of about 1.7 times over both OpenBLAS and MKL. The C implementation of the Riccati recursion has even lower overhead, and therefore the BLAS API of BLASFEO can show its full potential, with a speedup ranging between 2 and 3 times over

Table 2. Computation Time (in Seconds) of Riccati Recursion Implemented Natively in Octave 4.2.2, SciPy 1.2 (with Fix to `dtrmm`), Julia 1.0.1, and in C, on Intel Haswell Core i7 4810MQ @ 3.4 GHz

API	BLAS implemen.	$n_x = 8$ $n_u = 4$	$n_x = 24$ $n_u = 12$	$n_x = 40$ $n_u = 20$	$n_x = 64$ $n_u = 32$
Octave 4.2.2	OpenBLAS	$3.20 \cdot 10^{-4}$	$4.87 \cdot 10^{-4}$	$7.85 \cdot 10^{-4}$	$1.46 \cdot 10^{-3}$
Octave 4.2.2	MKL	$3.28 \cdot 10^{-4}$	$4.19 \cdot 10^{-4}$	$6.15 \cdot 10^{-4}$	$1.15 \cdot 10^{-3}$
Octave 4.2.2	BLASFEO	$3.00 \cdot 10^{-4}$	$3.71 \cdot 10^{-4}$	$5.28 \cdot 10^{-4}$	$9.75 \cdot 10^{-4}$
Python SciPy 1.2	OpenBLAS	$7.10 \cdot 10^{-5}$	$2.04 \cdot 10^{-4}$	$4.65 \cdot 10^{-4}$	$1.04 \cdot 10^{-3}$
Python SciPy 1.2	MKL	$7.09 \cdot 10^{-5}$	$1.48 \cdot 10^{-4}$	$3.01 \cdot 10^{-4}$	$7.73 \cdot 10^{-4}$
Python SciPy 1.2	BLASFEO	$5.79 \cdot 10^{-5}$	$1.07 \cdot 10^{-4}$	$2.15 \cdot 10^{-4}$	$5.65 \cdot 10^{-4}$
Julia 1.0.1	OpenBLAS	$1.82 \cdot 10^{-5}$	$1.47 \cdot 10^{-4}$	$4.19 \cdot 10^{-4}$	$1.05 \cdot 10^{-3}$
Julia 1.0.1	MKL	$1.77 \cdot 10^{-5}$	$9.36 \cdot 10^{-5}$	$2.81 \cdot 10^{-4}$	$8.31 \cdot 10^{-4}$
Julia 1.0.1	BLASFEO	$1.04 \cdot 10^{-5}$	$6.82 \cdot 10^{-5}$	$2.05 \cdot 10^{-4}$	$6.20 \cdot 10^{-4}$
BLAS from C	OpenBLAS	$1.35 \cdot 10^{-5}$	$1.11 \cdot 10^{-4}$	$3.40 \cdot 10^{-4}$	$8.26 \cdot 10^{-4}$
BLAS from C	MKL	$1.33 \cdot 10^{-5}$	$6.27 \cdot 10^{-5}$	$1.95 \cdot 10^{-4}$	$6.09 \cdot 10^{-4}$
BLAS from C	BLASFEO	$5.11 \cdot 10^{-6}$	$3.60 \cdot 10^{-5}$	$1.25 \cdot 10^{-4}$	$4.19 \cdot 10^{-4}$
BLASFEO from C	BLASFEO	$3.24 \cdot 10^{-6}$	$2.29 \cdot 10^{-5}$	$7.95 \cdot 10^{-5}$	$2.87 \cdot 10^{-4}$

BLAS and LAPACK provided by single-threaded versions of either BLASFEO, OpenBLAS 0.3.4-dev or MKL 2019.1.144. The BLASFEO API is also added as a reference. The control horizon length is fixed to $N = 10$; the number of states n_x and the number of controls n_u vary.

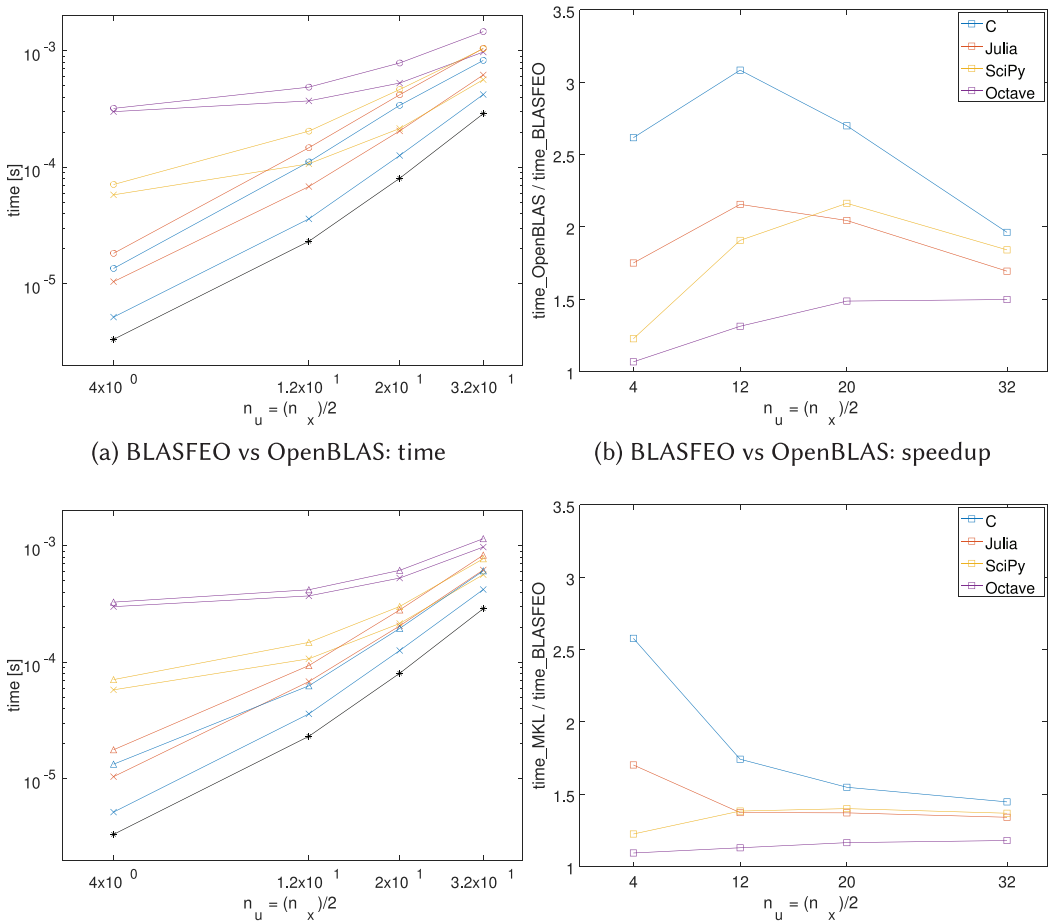
OpenBLAS and between 1.5 and 2.5 times over MKL (which in the C implementation is used with the MKL_DIRECT_CALL_SEQ option).

The BLASFEO API shows an additional speedup of 1.5 over the BLAS API of BLASFEO. Note that this speedup is larger than the speedup of the single routines evaluated in the performance plots in Section 4.1. The additional speedup comes from the fact that the BLASFEO API is non-destructive (there is an additional matrix argument reserved for the output, often avoiding the need for an explicit matrix copy); the fact that the BLASFEO API provides specialized routines (e.g., fusing `dsyrk` with a subsequent `dpotrf` in a single routine); and the fact that the BLASFEO API employs none or very limited internal memory (the reduction in the memory footprint increases the amount of memory which is kept in cache in between iterations of the Riccati recursion). This experimentally shows that the BLASFEO API can be a better fit for embedded applications than a similarly implemented BLAS API.

6 CONCLUSION

This article describes the implementation of a standard BLAS API in the BLASFEO framework, which optimizes the performance for small matrices fitting in cache. The proposed implementation scheme for a generic DLA routine switches between two algorithmic variants: the first one focuses on reducing overhead for small matrices (by limiting packing and avoiding dynamic memory allocation), and the other attains a performance level scaling better with the matrix size (by improving data layout and movement across the cache hierarchy). In special cases such as some `dgemm` variants and when dealing with rectangular matrices, two supplementary algorithmic variants can provide additional speedups. The flexibility of the assembly sub-routines framework is leveraged to conveniently implement kernels which operate on matrix arguments of different storage format.

Numerical experiments show the effectiveness of the approach. In the BLASFEO framework, the routines of the BLAS API are on average 10–15% slower than the BLASFEO API counterparts, but



(a) BLASFEO vs OpenBLAS: time

(b) BLASFEO vs OpenBLAS: speedup

(c) BLASFEO vs MKL: time

(d) BLASFEO vs MKL: speedup

Fig. 6. Graphical representation of the data in Table 2. Computation time (in seconds) and speedup of Riccati recursion implemented natively in Octave 4.2.2 (plotted in purple), SciPy 1.2 (with fix to `dtrmm`) (plotted in yellow), Julia 1.0.1 (plotted in red), and in C (plotted in blue), on Intel Haswell Core i7 4810MQ @ 3.4 GHz. BLAS and LAPACK provided by single-threaded versions of either BLASFEO (plotted with the cross "-x-"), OpenBLAS 0.3.4-dev (plotted with the circle "-o-"), or MKL 2019.1.144 (plotted with the triangle "-△-"). The BLASFEO API is also added as a reference (plotted in black with the asterisk "-*-"). The control horizon length is fixed to $N = 10$; the number of states n_x and the number of controls n_u vary.

for the matrix sizes of interest they are nonetheless significantly faster than state-of-the-art open source BLAS and LAPACK implementations, and in most cases they also outperform commercial implementations. The performance advantage is large especially in the case of LAPACK routines such as factorizations, where the speedup factor is about 2 for matrices of size up to 100.

These results are in a way more significant than the BLASFEO API presented in [9], since the implementation described in the current article sticks to the standard BLAS API, and therefore it is directly comparable to the other BLAS and LAPACK implementations.

Future research will focus on adding routines operating on additional matrix formats (e.g., the general stride proposed in [27]) by leveraging the flexibility of the assembly sub-routines

framework. Furthermore, the implementation of multi-threaded versions of DLA routines in the BLASFEO framework will be investigated, maintaining the focus on performance for small matrices.

APPENDICES

A IMPLEMENTING BLASFEO FOR A NEW ARCHITECTURE

This section discusses the effort required to implement BLASFEO for a new architecture, and shows the performance attained for different optimization levels for the Intel Haswell and ARM Cortex A57 assumed to be a new architecture in BLASFEO.

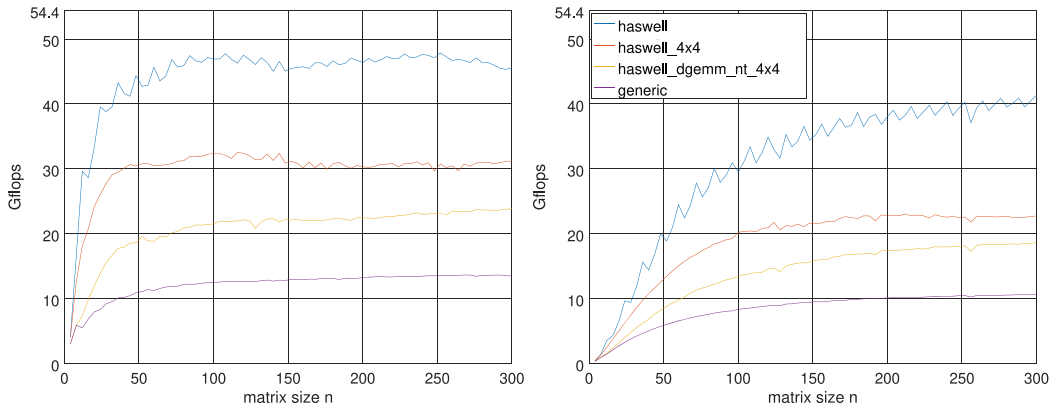
BLASFEO has been designed with the goal of enhancing performance for small matrices. In this it differs from other BLAS implementations such as BLIS, which has been designed with the explicit goal of being a "framework for rapidly instantiating BLAS functionality" [27]. As discussed in detail in the current article as well as in the original BLASFEO paper [9], in order to attain its goal, BLASFEO employs a large number of assembly coded kernels, tailored to each linear algebra routine and optimized for each hardware target. The use of the custom function calling convention described in Section 3.2 helps to manage this large amount of assembly code. However, the question remains, of how large is the effort required to implement BLASFEO for a new architecture.

In BLASFEO, only the innermost loop is coded as an assembly kernel. The framework around the kernels is coded in C, and it is tailored to a fixed panel size p_s. All hardware targets characterized by the same p_s share the same framework. Furthermore, the "generic" target of BLASFEO is coded in C in such a way that all kernels for the level 3 BLAS and LAPACK routines of the BLASFEO API and algorithmic variant "B" of the BLAS API are built on top of the single `kernel_dgemm_nt_4x4_lib4444` kernel (this is analogous to what happens in the corresponding assembly kernels for other targets, which are all built on top of the `inner_kernel_dgemm_nt_XxX_lib44` inner kernel). Therefore, if an optimized implementation of this single kernel is available, all these routines automatically get optimized to some degree.

Figures 7 and 8 show the performance of different optimization efforts of the `dgemm_nn` and `dpotrf_l` routines on the Intel Haswell and ARM Cortex A57 (which, here, serve as new architectures for illustration purposes). The "generic" target (purple line) performs rather poorly on Intel Haswell (since it cannot exploit the wide FP execution units of this microarchitecture), but it exceeds 50% of the maximum performance on the lower power ARM Cortex A57. The optimization of the single `kernel_dgemm_nt_4x4_lib4444` kernel (yellow line) gives most of the performance improvement for the linear algebra routines based on `4x4` kernels, and as the matrix size increases, most of the computations are performed by this single kernel. The optimization of all the other `4x4` kernels (red line) mainly improves performance for small matrices. Finally, by optimizing all kernels for the optimal kernel size (blue line) the performance is further improved, especially in the case of Intel Haswell, which has an optimal kernel size of `12x4`.

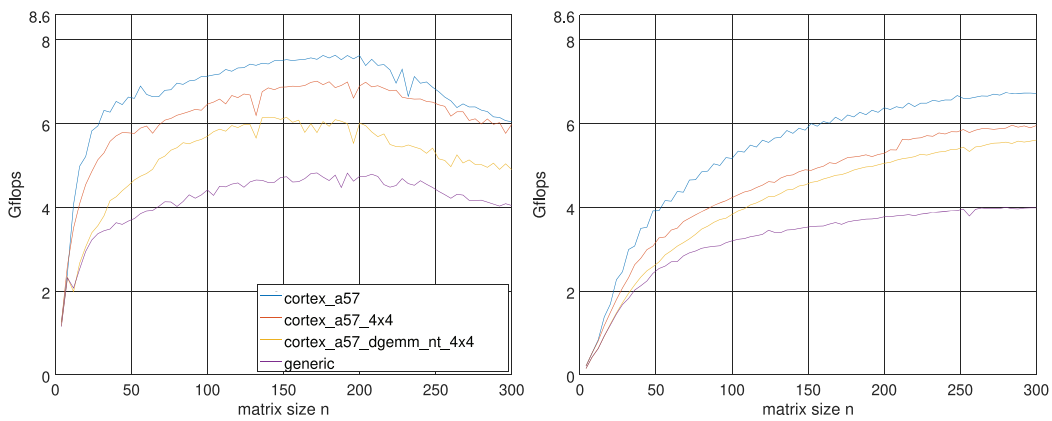
These findings are representative of a more generic trend. In the case of low-power microarchitectures with narrow FP execution units, the "generic" target of BLASFEO generally provides a good basis, and the optimization of a few key kernels gives most of the performance improvement. In the case of high-performance microarchitectures with wide FP execution units, the "generic" target gives rather poor performance, and it may not even use the optimal p_s value.

In the case of an architecture with a new value of p_s, a new version of the entire framework as well as of the assembly kernels needs to be implemented. As an example, this is the case for the AVX512 ISA (which at the time of writing is not supported yet in BLASFEO) since doubling



(a) dgemm_nn (b) dpotrf_l

Fig. 7. Comparison of different implementation efforts for the routines dgemm_nn and dpotrf_l: performance on Intel Haswell Core i7 4810MQ @ 3.4 GHz. The optimization levels are as follows: "haswell" target with optimal 12×4 kernel size (*haswell*); "haswell" target with 4×4 kernel size (*haswell_4x4*); "generic" target with the only kernel_dgemm_nt_4x4_lib4444 optimized for the "haswell" target (*haswell_dgemm_nt_4x4*); "generic" target (*generic*).



(a) dgemm_nn (b) dpotrf_l

Fig. 8. Comparison of different implementation efforts for the routines dgemm_nn and dpotrf_l: performance on ARM Cortex A57 @ 2.15 GHz. The optimization levels are: "cortex_a57" target with optimal 8×4 kernel size (*cortex_a57*); "cortex_a57" target with 4×4 kernel size (*cortex_a57_4x4*); "generic" target with the only kernel_dgemm_nt_4x4_lib4444 optimized for the "cortex_a57" target (*cortex_a57_dgemm_nt_4x4*); "generic" target (*generic*).

the FP registers width from 256 to 512 bits requires doubling the value of p_s from 4 to 8 in double precision and from 8 to 16 in single precision.

B PERFORMANCE EVALUATION OF ADDITIONAL PARAMETRIC VARIANTS

The previous sections of this article evaluated the performance of the BLASFEO implementation of some BLAS and LAPACK routines against state-of-the-art open source and proprietary implementations. For compactness reasons, the comparison had to be limited to selected parametric variants

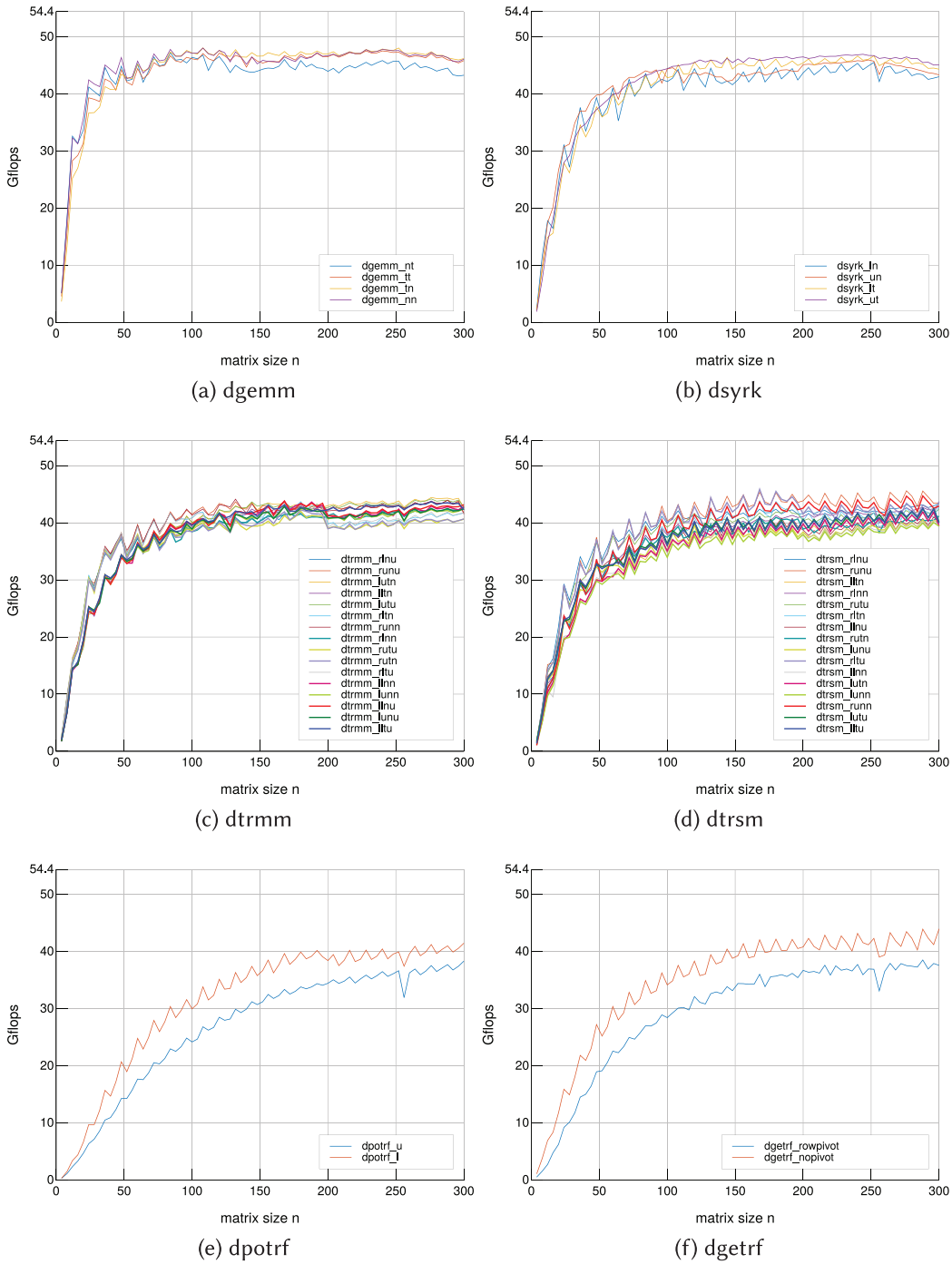


Fig. 9. Performance of all parametric variants of some BLAS and LAPACK routines on one core of the Intel Haswell Core i7 4810MQ @ 3.4 GHz. All matrices are square of size n.

of the routines like transposition, side, upper, or lower triangular, and so on. These variants were chosen as representative of the performance of the routines in the general case.

Conversely, this section aims at comparing the BLASFEO BLAS API implementation of all parametric variants of some BLAS and LAPACK routines against each other. In Figure 9, the performance of the BLAS routines `dgemm`, `dsyrk`, `dtrmm`, and `dtrsm`, and the LAPACK routines `dpotrf` and `dgetrf` is evaluated for the case of square matrices on the Intel Haswell microarchitecture. From the plots it is clear that the implementation approach presented in this article performs consistently well on all parametric variants, with small performance variations due to the nature of the variants themselves. No inherent performance drawback is found in the proposed implementation approach.

C `DGEMM` PERFORMANCE EVALUATION ON SKINNY MATRICES

Some considerations about the case of rectangular matrices are collected in Section 2.3.2. However, in the previous sections of this article, the performance evaluation of BLASFEO routines had to be limited to the case of square matrices.

This section compares the performance of all four `dgemm` variants in the case of both square and skinny (i.e., where one dimension is much smaller than the other) matrices on the Intel Haswell microarchitecture. In particular, for these benchmarks the small dimension is chosen to be fixed and equal to 4. The performance is evaluated in the four cases as follows:

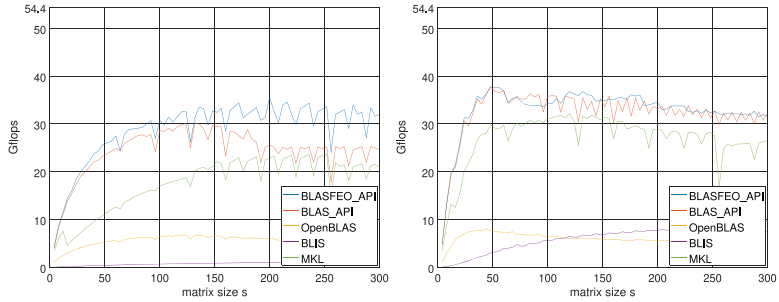
- $m = 4$, $n = 4$, k varied between 4 and 300 (Figure 10, left column).
- $m = 4$, n and k varied between 4 and 300 (Figure 10, right column).
- $n = 4$, m and k varied between 4 and 300 (Figure 11, left column).
- m, n, and k varied between 4 and 300 (Figure 11, right column).

The `dgemm` variants "NN," "NT," and "TT" are implemented using algorithmic variants "B" (for large matrices), "C" and "Ct" (for medium matrices, when $m < m$ and $m \geq n$, respectively), and "D" (for small matrices). The `dgemm` variant "TN" is implemented using algorithmic variants "B" (for large matrices), and "C" and "Ct" (for medium and small matrices), since a native algorithmic variant "D" would not perform better than the other algorithmic variants in any case.

From the plots it is clear that for most of the considered cases, the BLAS API of BLASFEO outperforms the other BLAS implementations, except a few cases where MKL is slightly faster. Therefore, the proposed approach shows a good and consistent performance also in the case of rectangular matrices, and no inherent performance drawback is found.

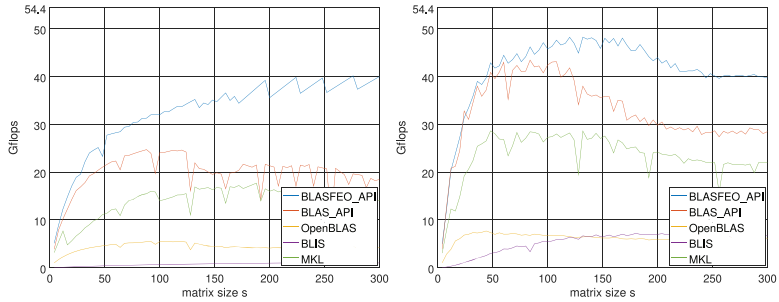
In these experiments, the BLASFEO API of BLASFEO is added as a reference. In the BLASFEO API, all input and output matrix arguments are stored in the panel-major format, so `dgemm` variants "NN," "NT," and "TT" are all implemented using only the equivalent of algorithmic variant "D" (i.e., no packing is performed). The performance is good for any matrix size, as there is no overhead from packing and the panel-major matrix format enhances data caching and movement. Therefore, it can be seen as an upper bound on the performance of the BLAS API, a bound which is approached when the packing overhead is minimized and the data memory layout allows the computational kernels to remain well fed.

Conversely, since algorithmic variant "D" would perform poorly, the `dgemm` variant "TN" is implemented using the equivalent of algorithmic variants "C" or "Ct" and packing is employed to explicitly transpose either the matrix A or B and reduce to variants "NN" or "TT," respectively. As a consequence, the performance is analogous to the performance of the BLAS API, and for every matrix size the best algorithmic variant has to be employed to minimize packing overhead.



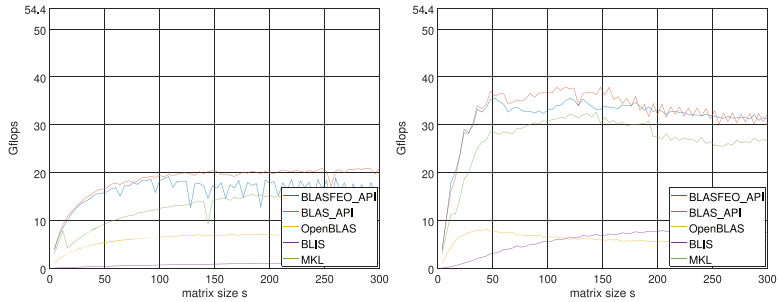
(a) dgemm_nn, m=4, n=4, k=s

(b) dgemm_nn, m=4, n=s, k=s



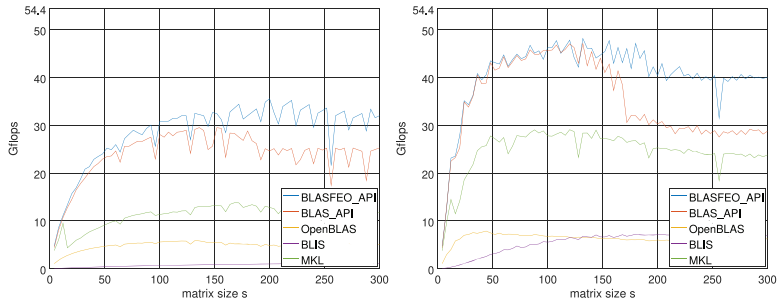
(c) dgemm_nt, m=4, n=4, k=s

(d) dgemm_nt, m=4, n=s, k=s



(e) dgemm_tn, m=4, n=4, k=s

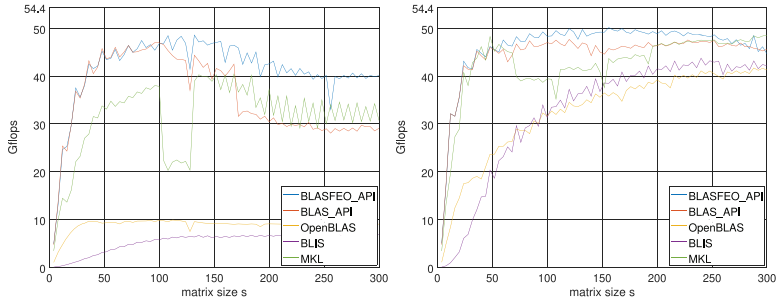
(f) dgemm_tn, m=4, n=s, k=s



(g) dgemm_tt, m=4, n=4, k=s

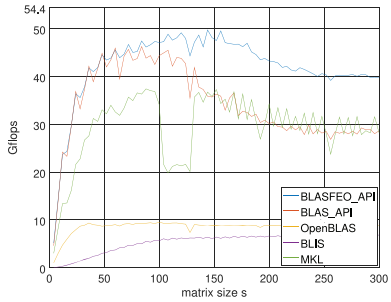
(h) dgemm_tt, m=4, n=s, k=s

Fig. 10. Performance of all parametric variants of dgemm one core of the Intel Haswell Core i7 4810MQ @ 3.4 GHz. Experiments for the cases $m = 4$, $n = 4$, k between 4 and 300 (left column), and $m = 4$, n and k between 4 and 300 (right column).

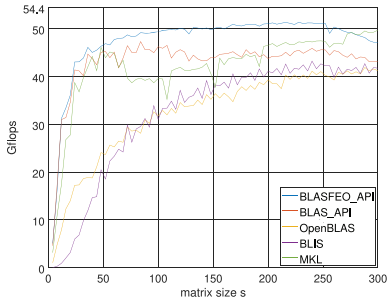


(a) dgemm_nn, m=s, n=4, k=s

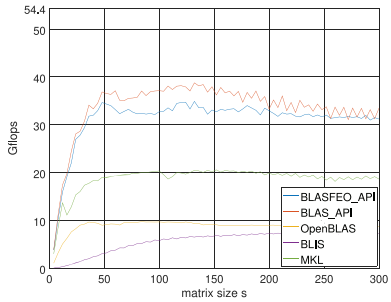
(b) dgemm_nn, m=s, n=s, k=s



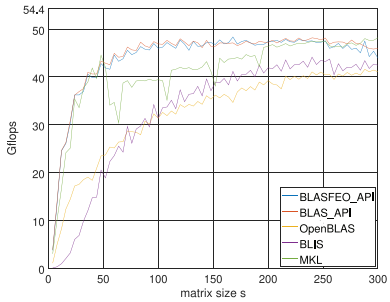
(c) dgemm_nt, m=s, n=4, k=s



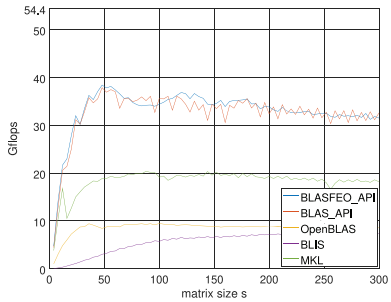
(d) dgemm_nt, m=s, n=s, k=s



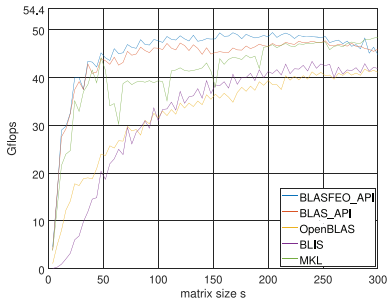
(e) dgemm_tn, m=s, n=4, k=s



(f) dgemm_tn, m=s, n=s, k=s



(g) dgemm_tt, m=s, n=4, k=s



(h) dgemm_tt, m=s, n=s, k=s

Fig. 11. Performance of all parametric variants of dgemm one core of the Intel Haswell Core i7 4810MQ @ 3.4 GHz. Experiments for the cases $n = 4$, m and k between 4 and 300 (left column), and m, n, and k between 4 and 300 (right column).

D COMMON COMMANDS IN SCIENTIFIC PROGRAMMING LANGUAGES AND THEIR IMPLEMENTATION USING BLAS AND LAPACK

This section contains tables of native commands in Octave, Python NumPy/SciPy, and Julia, and their implementation using calls to BLAS and LAPACK routines.

Note that the `char` options in the original Fortran interface are added to the routine name, after an underscore, in the same order as they appear in the routine signature. For example, `dsyrk_un` is the `dsyrk` routine called with options `uplo='u'` and `trans='n'`.

D.1 Octave

GNU Octave is open source software featuring a high-level programming language, primarily intended for numerical computing. Its language is mostly compatible with—and therefore the leading free alternative to—MATLAB.

Octave is written in C++ and it uses the C++ standard library. It can be extended with oct-files (using the native Octave API) or mex-files (for MATLAB compatibility), and it can use dynamically loaded libraries. Octave uses an interpreter to execute the Octave scripting language. The syntax, as with MATLAB, is matrix based. Octave employs calls to BLAS and LAPACK routines to implement the most common matrix and vector operations in its native language. Therefore, these operations are particularly efficient in Octave, especially if optimized BLAS and LAPACK implementations are employed.

The matrix multiplication in Octave is implemented with different versions of the BLAS routines `dgemm` and `dsyrk`. In cases where one of the two matrix factors is triangular, the general matrix-matrix multiplication routine `dgemm` is employed instead of the specialized routine `dtrmm`. The back- and forward-slash operators are implemented with a range of LAPACK routines depending on the matrix argument type, like `dpotrf`, `dpotrs`, `dgetrf`, `dgetrs`, and `dtrtrs`. The logic to choose between these routines adds considerable overhead to the routine calls, especially in the case of very small matrices. Direct calls to the BLAS routine `dtrsm` are not employed in the solution of triangular linear systems, and the LAPACK routine `dtrtrs` is employed in all cases, even if this implies the cost of an additional matrix transposition if the triangular matrix appears on the right-hand side.

Table 3 contains a list of some of the most common Octave commands and their implementation using BLAS and LAPACK routines. In Octave, matrices are stored in column-major, and therefore they can be passed directly to BLAS and LAPACK routines without the need for transposition. The findings apply to the Octave version 4.2.2.

Table 3. Version of BLAS and LAPACK Routines Employed in Octave Native Language

Octave command	BLAS and LAPACK routine	flops
`C = A * B`	`dgemm_nn`	$2 \cdot n^3$
`C = A * B'`	`dgemm_nt`	$2 \cdot n^3$
`C = A' * B`	`dgemm_tn`	$2 \cdot n^3$
`C = A' * B'`	`dgemm_tt`	$2 \cdot n^3$
`C = A * A'`	`dsyrk_un`	$1 \cdot n^3$
`C = A' * A`	`dsyrk_ut`	$1 \cdot n^3$
`C = chol(Q, 'lower')`	`dpotrf_l`	$1/3 \cdot n^3$
`C = chol(Q, 'upper')`	`dpotrf_u`	$1/3 \cdot n^3$

`A`, `B`, and `C` denote generic matrices; `Q` denotes a symmetric positive definite matrix. All flop counts refer to $n \times n$ matrices.

D.2 NumPy/SciPy

Python is an interpreted high-level language with a dynamic type system and automatic memory management. Due to its large standard library and availability of interpreters for many operating systems, it has become a popular choice in many fields of application. In particular, thanks to the presence of an extensive mathematics library and the third-party library NumPy, it is often used as a scientific scripting language. Although speed is not the main focus of the language, as opposed to readability and simplicity [24] of use, time-critical functionalities can be moved to extension modules written in C or delegated to external high-performance code, which can be accessed through foreign function libraries.

NumPy [20] is an open source library providing the Python language with multi-dimensional array objects and a large collection of high-level mathematical functions. In particular, it provides an interface for linear algebra operations on matrices and vectors, and in this it is analogous to other numerical programming languages like Octave. These operations are ultimately implemented using calls to BLAS and LAPACK routines, even if NumPy provides a more convenient interface, which hides most low-level details of BLAS and LAPACK APIs and performs additional sanity checks. This is analogous to the Octave implementation, and it adds a similar level of overhead. NumPy provides functionality to operate on matrices on both column- and row-major formats, even if the default storage scheme is row-major. Therefore, BLAS is effectively accessed through the CBLAS interface, while LAPACK routines are called directly, possibly after transposition between row- and column-major.

SciPy is an open source Python library for scientific and technical computing; it is part of the NumPy stack and builds on the NumPy array object. In particular, SciPy provides a rich library for linear algebra operations, and a set of functions that is both wider range and lower level compared to NumPy. For some linear algebra routines multiple implementations exist, with the lower level being a thin wrapper to the corresponding BLAS or LAPACK routine. These low-level routines allow one to perform specialized operations (like exploiting the fact that a matrix is triangular) and do not perform any additional sanity checks, resulting in much lower overhead. SciPy always directly calls the BLAS and LAPACK APIs. If the NumPy matrices are stored in row-major order, they are explicitly transposed into dynamically allocated memory buffers, which are then passed to the BLAS and LAPACK routines. Therefore, in order to obtain full performance from the SciPy routines, all matrices must be stored in column-major order.

The Cholesky factorization is used as an example. This factorization is implemented in the NumPy routine `numpy.linalg.cholesky`, which computes the lower triangular factor of a positive definite matrix. The SciPy routine `scipy.linalg.cholesky` additionally provides options to compute the upper/lower triangular factor, to overwrite the input matrix and to enable/disable checks for finiteness of the elements of the input matrix. The lowest level interface is provided by the SciPy routine `scipy.linalg.lapack.dpotrf`, which roughly provides the same functionality as the corresponding LAPACK routine.

As NumPy provides similar levels of functionality and overhead as Octave, only SciPy is considered in the performance evaluations performed in this article. In particular, the version 1.0 of SciPy introduces new low-level functions providing wrappers to the BLAS routines `dtrsm` and `dtrmm`. Unfortunately, up to the current version 1.2 a bug affects the functionality of `dtrmm` as needed in the implementation of the Riccati recursion in Section 5.2. The replacement of this routine with `dgemm` does not allow one to fully exploit the structure of the Riccati recursion. Since SciPy is open source software, we were able to examine the source code and implement a fix for the bug. This fixed version of SciPy 1.2 is also considered in the tests in Section 5.2.

Table 4 contains a list of SciPy/NumPy commands and their respective calls to BLAS and LAPACK routines. In particular, different ways of calling the BLAS routines `dgemm` and `dsyrk` and the

Table 4. Version of BLAS and LAPACK Routines Employed in NumPy

NumPy / SciPy command	BLAS and LAPACK routine	flops
`C = np.dot(A, B)`	`cblas` → `dgemm_nn`	$2 \cdot n^3$
`C = np.dot(A, B.T)`	`cblas` → `dgemm_tn`	$2 \cdot n^3$
`C = np.dot(A.T, B)`	`cblas` → `dgemm_nt`	$2 \cdot n^3$
`C = np.dot(A.T, B.T)`	`cblas` → `dgemm_tt`	$2 \cdot n^3$
`sp.linalg.blas.dgemm(1.0, A, B, c=C, trans_a=False, trans_b=False)`	`dgemm_nn`	$2 \cdot n^3$
`sp.linalg.blas.dgemm(1.0, A, B, c=C, trans_a=False, trans_b=True)`	`dgemm_nt`	$2 \cdot n^3$
`sp.linalg.blas.dgemm(1.0, A, B, c=C, trans_a=True, trans_b=False)`	`dgemm_tn`	$2 \cdot n^3$
`sp.linalg.blas.dgemm(1.0, A, B, c=C, trans_a=True, trans_b=True)`	`dgemm_tt`	$2 \cdot n^3$
`C = np.dot(A, A.T)`	`cblas` → `dsyrk_lt`	$1 \cdot n^3$
`C = np.dot(A.T, A)`	`cblas` → `dsyrk_ln`	$1 \cdot n^3$
`sp.linalg.blas.dsyrk(1.0, A, trans=0, lower=0 c=C, overwrite_c=True)`	`dsyrk_un`	$1 \cdot n^3$
`sp.linalg.blas.dsyrk(1.0, A, trans=1, lower=0 c=C, overwrite_c=True)`	`dsyrk_ut`	$1 \cdot n^3$
`sp.linalg.blas.dsyrk(1.0, A, trans=0, lower=1 c=C, overwrite_c=True)`	`dsyrk_ln`	$1 \cdot n^3$
`sp.linalg.blas.dsyrk(1.0, A, trans=1, lower=1 c=C, overwrite_c=True)`	`dsyrk_lt`	$1 \cdot n^3$
`C = np.linalg.cholesky(Q)`	`dpotrf_l`	$1/3 \cdot n^3$
`C = sp.linalg.cholesky(Q, lower=True, overwrite_a=False, check_finite=False)`	`dpotrf_l`	$1/3 \cdot n^3$
`C = sp.linalg.cholesky(Q, lower=False, overwrite_a=False, check_finite=False)`	`dpotrf_u`	$1/3 \cdot n^3$
`sp.linalg.lapack.dpotrf(Q, lower=1, clean=0, overwrite_a=True)`	`dpotrf_l`	$1/3 \cdot n^3$
`sp.linalg.lapack.dpotrf(Q, lower=0, clean=0, overwrite_a=True)`	`dpotrf_u`	$1/3 \cdot n^3$

`A`, `B`, and `C` denote generic matrices; `Q` denotes a symmetric positive-definite matrix. All flop counts refer to $n \times n$ matrices. Notice that calls to CBLAS in NumPy are carried out with `Order = CblasRowMajor`.

LAPACK routine `dpotrf` are considered. The findings apply to the NumPy version 1.11 and SciPy version 1.2.

D.3 Julia

Julia is an open source high-level GP programming language. It is dynamically typed and in that regard it resembles scripting languages such as Octave and Python. However, Julia makes use of JIT compilation, and its compiler is based on LLVM and therefore can generate optimized native code for multiple architectures. Once compiled, the speed of Julia code approaches the speed of statically typed and compiled languages such as C and C++. This combination of a dynamically typed language and high performance is an explicit design choice of Julia.

Table 5. Version of BLAS and LAPACK Routines Employed in Julia Native Language

Julia command	BLAS and LAPACK routine	flops
`C = A * B`	dgemm_nn	$2 \cdot n^3$
`C = A * tranpose(B)`	dgemm_nt	$2 \cdot n^3$
`C = transpose(A) * B`	dgemm_tn	$2 \cdot n^3$
`C = transpose(A) * tranpose(B)`	dgemm_tt	$2 \cdot n^3$
`mul!(C, A, B)`	dgemm_nn	$2 \cdot n^3$
`mul!(C, A, transpose(B))`	dgemm_nt	$2 \cdot n^3$
`mul!(C, transpose(A), B)`	dgemm_tn	$2 \cdot n^3$
`mul!(C, transpose(A), transpose(B))`	dgemm_tt	$2 \cdot n^3$
`BLAS.gemm!('N', 'N', alpha, A, B, beta, C)`	dgemm_nn	$2 \cdot n^3$
`BLAS.gemm!('N', 'T', alpha, A, B, beta, C)`	dgemm_nt	$2 \cdot n^3$
`BLAS.gemm!('T', 'N', alpha, A, B, beta, C)`	dgemm_tn	$2 \cdot n^3$
`BLAS.gemm!('T', 'T', alpha, A, B, beta, C)`	dgemm_tt	$2 \cdot n^3$
`C = A * transpose(A)`	dsyrk_un	$1 \cdot n^3$
`C = transpose(A) * A`	dsyrk_ut	$1 \cdot n^3$
`mul!(C, A, transpose(A))`	dsyrk_un	$1 \cdot n^3$
`mul!(C, transpose(A), A)`	dsyrk_ut	$1 \cdot n^3$
`BLAS.syrk!('L', 'N', alpha, A, beta, C)`	dsyrk_ln	$1 \cdot n^3$
`BLAS.syrk!('L', 'T', alpha, A, beta, C)`	dsyrk_lt	$1 \cdot n^3$
`BLAS.syrk!('U', 'N', alpha, A, beta, C)`	dsyrk_un	$1 \cdot n^3$
`BLAS.syrk!('U', 'T', alpha, A, beta, C)`	dsyrk_ut	$1 \cdot n^3$
`cholesky!(Q)`	dpotrf_u	$1/3 \cdot n^3$
`LAPACK.potrf!('L', Q)`	dpotrf_l	$1/3 \cdot n^3$
`LAPACK.potrf!('U', Q)`	dpotrf_u	$1/3 \cdot n^3$

`A`, `B`, and `C` denote generic matrices; `Q` denotes a symmetric positive-definite matrix. All flop counts refer to $n \times n$ matrices.

Regarding numeric computations, the Julia syntax has similarities with the Matlab and Octave syntax. Besides support for multi-dimensional arrays, Julia provides native implementations for many linear algebra operations in the `LinearAlgebra` module. Basic matrix operations are implemented with calls to BLAS and LAPACK routines; the default implementation is OpenBLAS. Calls to these routines do not need any additional transposition since the native matrix format in Julia is column-major.

Similarly to Python with NumPy and SciPy, in Julia there are several commands to perform the same linear algebra operations, trading off performance and easiness of use. For example, the general matrix-matrix multiplication can be performed using the operator `'*'` (which returns the result in a newly allocated matrix), or calling the function `mul!` (which requires a pre-allocated output matrix) of the `LinearAlgebra` module. Furthermore, wrappers to BLAS and LAPACK routines are part of the `LinearAlgebra.BLAS` and `LinearAlgebra.LAPACK` modules. The functions in these modules provide the same functionality as the BLAS and LAPACK routines. As an example, the `gemm!` function also accepts the scalar `alpha` and `beta` arguments. More importantly, specialized routines like `syrk`, `symm`, or `trmm` can be called directly through these wrappers.

When operating with sub-matrices, it is important to notice that in Julia, slicing an array creates a copy of the selected sub-array. This has important consequences on performance, since unnecessary copying can be expensive. Furthermore, this can lead to undesired behaviors: if a sliced matrix is passed as the return argument of an in-place linear algebra function, the result of the operations

gets lost as it is not copied back in the original matrix. The Julia command `view` should be used instead to get an in-place reference to a sub-matrix, which can be used to modify the original matrix. Notice that in this article, static arrays (from the module `StaticArrays`) are not considered, since basic linear algebra operations on them are implemented with JIT-compiled native Julia routines and not with calls to BLAS and LAPACK routines.

Table 5 contains a list of some common Julia commands for matrix operations, and their implementation using BLAS and LAPACK routines. The findings apply to the Julia version 1.0.1.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (3rd ed.). SIAM.

[2] BLASFEO. 2016. Retrieved from https://github.com/giaf/blasfeo.

[3] Blaze. 2012. Retrieved from http://bitbucket.org/blaze-lib/blaze.

[4] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 35, 1 (2009), 38–53.

[5] Eigen. 2010. Eigen v3. Retreved from http://eigen.tuxfamily.org/.

[6] H. J. Ferreau, S. Almer, R. Verschueren, M. Diehl, D. Frick, A. Domahidi, J. L. Jerez, G. Stathopoulos, and C. Jones. 2017. Embedded optimization methods for industrial automatic control. In *Proceedings of the IFAC World Congress*.

[7] G. Frison, H. B. Sorensen, B. Dammann, and J. B. Jørgensen. 2014. High-performance small-scale solvers for linear model predictive control. In *Proceedings of the European Control Conference (ECC'14)*. 128–133.

[8] Gianluca Frison. 2015. *Algorithms and Methods for High-Performance Model Predictive Control*. Ph.D. thesis, Technical University of Denmark (DTU).

[9] G. Frison, D. Kouzoupis, T. Sartor, A. Zanelli, and M. Diehl. 2018. BLASFEO: Basic linear algebra subroutines for embedded optimization. In *ACM Transactions on Mathematical Software* 44, 4 (2018), 42:1–42:30.

[10] M. R. Hasan. 2017. *Maintaining High Performance Across All Problem Sizes and Parallel Scales Using Microkernel-Based Linear Algebra*. Ph.D. thesis. Louisiana State University.

[11] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. 2016. LIBXSMM: Accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[12] B. Houska, H. J. Ferreau, and M. Diehl. 2011. An auto-generated real-time iteration algorithm for nonlinear MPC in the microsecond range. *Automatica* 47, 10 (2011), 2279–2285.

[13] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari. 2014. Embedded online optimization for model predictive control at megahertz rates. *IEEE Transactions on Automatic Control* 59, 12 (2014), 3238–3251.

[14] Kazushige Goto and Robert A. van de Geijn. 2008. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software* 34, 3 (2008), 12:1–12:25.

[15] Kazushige Goto and Robert A. van de Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software* 35, 1 (2008).

[16] Bo Kågström, Per Ling, and Charles Van Loan. 1998. GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. *ACM Transactions on Mathematical Software* 24, 3 (1998), 268–302.

[17] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra. 2016. High-performance matrix-matrix multiplications of very small matrices. In *European Conference on Parallel Processing, Springer*. 659–671.

[18] Intel. 2019. Math Kernel Library. Retrieved from https://software.intel.com/en-us/mkl.

[19] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software* 5 (1979), 308–325.

[20] Travis E. Oliphant. 2006. *A Guide to NumPy*. Trelgol Publishing USA.

[21] OpenBLAS. 2011. OpenBLAS: An optimized BLAS library. Retrieved from http://www.openblas.net/.

[22] Daniele G. Spampinato and Markus Püschel. 2016. A basic linear algebra compiler for structured matrices. In *International Symposium on Code Generation and Optimization (CGO)*. 117–127.

[23] D. G. Spampinato, D. Fabregat-Traver, P. Bientinesi, and M. Püschel. 2018. Program generation for small-scale linear algebra applications. In *International Symposium on Code Generation and Optimization (CGO)*. 327–339.

[24] Tim Peters. 2004. PEP 20—The Zen of Python. Retrieved from https://www.python.org/dev/peps/pep-0020/.

[25] R. C. Whaley and J. Dongarra. 1999. Automatically tuned linear algebra software. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*.

[26] F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti. 2009. The libflame library for dense matrix computations. In *IEEE Computing in Science and Engineering* 11, 6 (2009).

[27] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software* 41, 3 (2015), 14:1–14:33.

[28] Field G. Van Zee, Tyler Smith, Francisco D. Igual, Mikhail Smelyanskiy, Xianyi Zhang, Michael Kistler, Vernon Austel, John Gunnels, Tze Meng Low, Bryan Marker, Lee Killough, and Robert A. van de Geijn. 2016. The BLIS framework: Experiments in portability. *ACM Transactions on Mathematical Software* 42, 2 (2016), 12:1–12:19.

Received February 2019; revised September 2019; accepted January 2020