Noname manuscript No.
(will be inserted by the editor)

Integration and exploitation of *intra-routine malleability* in BLIS

Rafael Rodríguez-Sánchez · Francisco D. Igual · Enrique S. Quintana-Ortí

Received: date / Accepted: date

Abstract Malleability is a property of certain applications (or tasks) that, under an external request or automatically, can accommodate a dynamic modification of the degree of parallelism being exploited at runtime. Malleability improves resource usage (core occupation) on modern multi-core architectures for applications that exhibit irregular and divergent execution paths and heavily depend on the underlying library performance to attain high performance. The integration of malleability within high-performance instances of the BLAS (Basic Linear Algebra Subprograms) is nonexistent and, in addition, it is difficult to attain given the rigidity of current application programming interfaces (APIs). In this paper, we overcome these issues presenting the integration of a malleability mechanism within BLIS, a high-performance and portable framework to implement BLAS-like operations. For this purpose, we leverage low level (yet simple) APIs to integrate on-demand malleability across all Level-3 BLAS routines, and we demonstrate the performance benefits of this approach by means of a higher-level dense matrix operation: the LU factorization with partial pivoting and look-ahead.

Keywords Malleability · Linear Algebra · BLAS · Multi-core architectures

R. Rodríguez-Sánchez
Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid. 28040 Madrid (Spain)
E-mail: rafaelrs@ucm.es

F. D. Igual
Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid. 28040 Madrid (Spain)
E-mail: figual@ucm.es

E. S. Quintana-Ortí
Departamento de Informtica de Sistemas y Computadores
Universitat Politècnica de València. 46022 Valencia (Spain)
E-mail: quintana@disca.upv.es

1 Introduction

The Basic Linear Algebra Subroutines (BLAS) [8] and the Linear Algebra Package (LAPACK) [12] standardize domain-specific interfaces for Dense Linear Algebra (DLA) operations that constitute the fundamental building blocks for many scientific and engineering applications. Since the 70s, the scientific community has devoted great efforts to further develop these interfaces with the goal of ensuring performance portability across a wide range of computer architectures. This portability is mainly achieved by means of highly-optimized and architecture-specific implementations of the BLAS for almost any computing architecture.

Focusing exclusively on how parallelism is extracted in applications built on top of the BLAS, we can follow [13] to establish the following categories:

- *rigid*, in which the amount of parallelism is fixed at application level, via environment variables, or even fixed at compilation time (as, e.g., in ATLAS [16]);
- *moldable*, in which the amount of parallelism is fixed on a per-call basis, via thread-safe calls to an application programming interface (API) that controls the degree of parallelism for individual application threads running BLAS kernels (as, e.g., in Intel MKL [6] or BLIS [18]); and
- *malleable*, in which the amount of threads executing a specific BLAS call can vary *during* execution.

Typically, parallelism is controlled in modern BLAS implementations via two different mechanisms, namely:

- *Environment variables*: In this option, parallelism can only be defined at application-level, and it is fixed a priori, being unchangable throughout the complete execution. While this approach is valid for many applications, it yields *rigid* parallel BLAS implementations: the amount of threads cannot vary between different BLAS calls within applications, and obviously, the amount of threads cannot vary *during* a BLAS execution.
- *Specific runtime APIs*: Similarly to other modern BLAS distributions (e.g., OpenBLAS or Intel MKL), BLIS addresses the *moldability* problem by offering ad-hoc APIs to set the overall degree of parallelism for individual BLAS calls. For this purpose, a recent release of the BLIS framework introduced an *Expert API*, which allows parallelism to be expressed at routine-level (see Section 2.2).

Currently, no complete BLAS implementation offers native support for malleability, though some previous experiences have demonstrated the performance potential of such approach, for a few selected DLA applications, by performing a prototype integration into specific (BLIS) routines [2,3]. In general, malleability is of great appeal for applications comprising different high-level parallel execution paths and where the workload is not perfectly balanced across them. By leveraging malleability, as soon as one execution path is complete, idle cores can be rapidly leveraged by other execution paths to increase

their degree of parallelism and improve the occupation of the architecture. Previous works with malleability use custom codes extracted from BLIS in order to accommodate malleability. In this paper, we propose an innovative and integral integration strategy to accommodate malleability in all Level-3 BLIS routines, and we evaluate the potential overhead and application benefits for a specific LAPACK operation: the LU factorization with look-ahead.

The remainder of the paper is organized as follows. Section 2 provides a general overview of the mechanisms included in the standard version of BLIS to extract sequential and parallel performance of BLAS routines. Section 3 describes the modifications proposed in order to integrate malleability in BLIS. Section 4 evaluates the performance and overhead attained by a fully malleable BLIS implementation, and its benefits when applied to LU factorization with look-ahead. Finally, Section 5 closes the paper with some concluding remarks and proposal for further application of malleability in BLIS.

2 BLIS background

The BLAS-Like Instantation Software (BLIS) is an open-source framework that unveils the internal implementation of the BLAS kernels, facilitating the exploration of new optimizations/methodologies [21, 20, 17, 19]. In contrast, implementations such as ATLAS and OpenBLAS expose their internals with a more reduced detail, while commercial libraries such as Intel MKL only offer a "black box" service. An additional important aspect of the BLIS framework is that it is competitive with many well-known high-performance BLAS implementations.

In this section we review the design principles that underlie BLIS, using the implementation of GEMM as a particular case study.

Consider the matrices $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, and $C \in \mathbb{R}^{m \times n}$. BLIS mimics GotoBLAS to implement the GEMM kernel as three nested loops around a macro-kernel plus two packing routines (see Loops 1–3 in Fig. 1). The macro-kernel is then implemented in terms of two additional loops around a *micro-kernel* (Loops 4 and 5 in that figure). The loop ordering in BLIS, together with the packing routines and an appropriate choice of the BLIS cache configuration parameters (n_c, k_c, m_c, n_r and m_r), orchestrate a regular pattern of data transfers across the levels of the memory hierarchy, and aim to amortize the cost of these transfers with enough computation from within the micro-kernel [18] in order to attain near-peak performance. In most architectures, m_r, n_r are in the range 4–16; m_c, k_c are in the order of a few hundreds; and n_c can be up to a few thousands [18, 21].

2.1 Multithreading management

The parallelization of BLIS GEMM for multi-threaded architectures has been analyzed for conventional symmetric multicore processors [21], modern many-

```
Loop 1   for j_c = 0, ..., n − 1 in steps of n_c
Loop 2     for p_c = 0, ..., k − 1 in steps of k_c
             B_c := B(p_c : p_c + k_c − 1, j_c : j_c + n_c − 1)              // Pack into B_c
Loop 3       for i_c = 0, ..., m − 1 in steps of m_c
               A_c := A(i_c : i_c + m_c − 1, p_c : p_c + k_c − 1)            // Pack into A_c
Loop 4         for j_r = 0, ..., n_c − 1 in steps of n_r                    // Macro-kernel
Loop 5           for i_r = 0, ..., m_c − 1 in steps of m_r
                   C_c(i_r : i_r + m_r − 1, j_r : j_r + n_r − 1)  // Micro-kernel
                     += A_c(i_r : i_r + m_r − 1, 0 : k_c − 1)
                      · B_c(0 : k_c − 1, j_r : j_r + n_r − 1)
                 endfor
               endfor
             endfor
           endfor
         endfor
```

Fig. 1 High performance implementation of GEMM in BLIS. In the code, $C_c \equiv C(i_c : i_c + m_c - 1, j_c : j_c + n_c - 1)$ is a notation artifact, introduced to ease the presentation, while A_c, B_c correspond to actual buffers that are involved in data copies.

threaded architectures [14], and asymmetric multicore processors [4]. The general conclusion that can be extracted from these works is that the most suitable parallelization strategy strongly depends on the underlying architecture. Fortunately, an important feature of BLIS is that the five nested-loop organization of GEMM (and of all other Level-3 BLAS routines) is exposed to the programmer by means of environment variables or by some API calls, so that the programmer can decide at which level(s) parallelism should be set (i.e., exploited). In addition, the programmer can delegate this decision to the library as it is done in many other libraries.

In general, a convenient option in most single socket systems with shared L3 cache and core-private level L1 and L2 caches is to parallelize Loop 3. With this strategy, all threads collaborate to pack the macro-panel B_c into the L3 cache, and each thread packs a different macro-panel A_c into its L2 cache as well as executes a different instance of the macro-kernel. In addition, if there are not sufficient A_c macro-panels to "feed" all available physical cores, a second or third level of parallelism can be extracted from Loop 4 and Loop 5 in which different threads will operate on independent instances of the micro-kernel, but access the same macro-panel A_c in the L2 cache.

At the start of the execution of a Level-3 BLAS routine, BLIS checks the parallelism set for all five loops and spawns as many threads as the multiplication of these five values. Consider, for example, a Level-3 BLAS routine in which 2-way parallelism is set (to be exploited) for Loop 4 and 1-way parallelism for the remaining four loops (i.e., no parallelism is to be extracted for those other loops); see Fig. 2. In this configuration, two threads iterate over the full iteration space of Loop 1, ranging from 0 to n. Then, in the 2-way parallel Loop 4, each thread iterates over (roughly) half of the iteration range. (For example, Th_0 iterates from 0 to $\frac{n_c}{2}$, while Th_1 iterates from $\frac{n_c}{2}$ to n_c.) Finally, in the sequential Loop 5, the threads again iterate over the full iteration space of this loop, ranging from 0 to m_c. For brevity, Loop 2 and Loop 3 are omitted in the figure, but they are equivalent to Loop 1 or Loop 5 as no parallelism is exploited in any of those levels for this particular example.

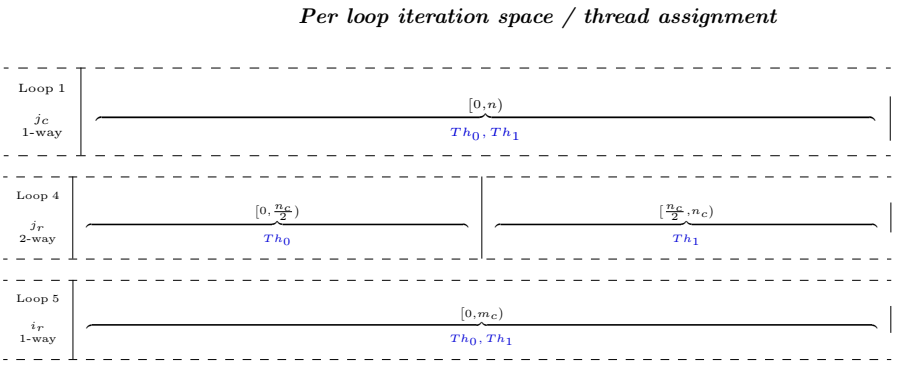


Fig. 2 Example of workload distribution in the standard version of BLIS.

2.2 APIs in BLIS

BLIS exposes a number of APIs to handle the framework internals that offer a path to invoke the BLAS routines with different levels of abstraction, functionality, complexity and internal exposability. To illustrate these APIs, Listing 1 highlights several prototypes for related routines provided within BLIS for the GEMM[1] kernel.

FORTRANAPI. The legacy BLAS interface exposes the standard Fortran-like specification of the routines, as introduced in [8]. When stored as two-dimensional arrays, the API receives typed pointers to input/output linear arrays storing the matrices, together with their corresponding dimensions (`m, n, k` in the listing) and leading dimensions (`lda, ldb, ldc`). `alpha` and `beta` are typed scalars, and transposition of the matrix operands `A` and `B` is controlled through the corresponding parameters.

OBJECTAPI. The BLIS Object API abstracts the floating-point types of the operands, and encapsulates ellaborated data structures that hide internal properties of the so-called *matrix objects*, including the possibility of arbitrary (an independent) row and column strides. This, in practice, overcomes the constrain of linear (column- or row-wise) storage of matrices and vectors forced by the legacy Fortran API.

TYPEDAPI. In terms of functionality, the BLIS Typed API is a hybrid implementation that merges some of the characteristics of the FORTRANAPI and the OBJECTAPI. It provides specialized (per-type) calls that expose raw typed data pointers and matrix dimensions; however, by means of additional parameters, it permits arbitrary row and column strides for each matrix.

[1] The listing displays the real double-precision floating-point API. Other realisations of this kernel operating with distinct datatypes and precisions present very similar interfaces.

ExpertApi. The BLIS Typed Expert API was recently developed to accommodate, among other functionality, local per-call multi-threading variations within the framework, hence introducing *moldability* into BLIS. In this work, we leverage this API to extend the functionality and offer support for *inter-routine malleability*. As shown in Listing 1, the ExpertApi extends the TypedApi by adding the following two new parameters:

- A *context* object (parameter `cntx_t * cntx`). This structure encodes several general parameters with a direct impact on the algorithmic behavior of BLIS for each specific routine, namely: loop ordering, block sizes (n_c, k_c, m_c, n_r and m_r) and specific micro-kernel to employ. In principle, these parameters should be adjusted to tune the behaviour of BLIS to the underlying micro-architecture and organization of the memory hierarchy.
- A *runtime* object (parameter `rntm_t * rntm`). This structure controls the degree and distribution of thread parallelism for the complete BLIS routine, and it can be modified on a per-routine invocation basis. We emphasize that the BLIS *runtime objects* do not have an analogue implementation in most other BLAS libraries[2], where parallelism is usually specified at a global level. The runtime object should be initialized and its values (amount and distribution of parallelism) set before passing it to any BLIS routine (by means of `bli_rntm_init` and `bli_rntm_set_ways`, see Listing 2).

Our strategy to enhance BLIS with *intra-routine malleability* leverages the runtime object to dynamically modify the parallel configuration of the BLAS routine while it is under execution. In order to attain this, we propose a number of modifications to the framework, and pass additional activation/deactivation information to the spawned threads, as explained in the next section.

3 Integration of Intra-Routine Malleability

In BLIS, a runtime object includes an array of five integers (one per BLAS level-3 loop) that specify the degree of parallelism to be extracted from each loop. This information is used to spawn a given number of threads at the beginning of the routine execution. By design, the number of spawned threads cannot be modified after this initialization; however, the behaviour of the threads can be dynamically *adjusted* on-demand. In order to do this, we embed into the runtime object an additional array, also of five integers, that will be used to set the number of *active* threads at each loop.

Armed with this information, at the beginning of each loop iteration, the workload (that is, the iteration range for that loop) is distributed using the new array to calculate the mapping of iterations to threads. Those threads without any workload assigned to them at a given level will advance to the end of that level, where they will remain blocked (in a passive wait to avoid

[2] Modern versions of NVIDIA CUBLAS provide a per-host-thread handler that, in some sense, resembles the use of context and runtime objects in BLIS. However, it is out-of-scope for this paper to discuss its functionality and possible application to GPU malleability.

```
/*** Exposed APIs for double-precision GEMM in BLIS. ***/

// BLAS Fortran API.
void dgemm (
  char* transa, char* transb,
  int* m, int* n, int* k,
  double* alpha, double* a,
  int* lda, double* b, int* ldb,
  double* beta, double* c, int* ldc );

// BLIS Object API.
void bli_gemm (
  obj_t* alpha, obj_t* a, obj_t* b, obj_t* beta, obj_t* c );

// BLIS Typed API.
void bli_dgemm (
  trans_t transa, trans_t transb, dim_t m, dim_t n, dim_t k,
  ctype*  alpha, ctype*  a, inc_t rsa, inc_t csa,
  ctype*  b, inc_t rsb, inc_t csb,
  ctype*  beta, ctype* c, inc_t rsc, inc_t csc );

// BLIS Typed Expert API.
void bli_dgemm_ex (
  trans_t transa, trans_t transb, dim_t m, dim_t n, dim_t k,
  ctype*  alpha, ctype*  a, inc_t rsa, inc_t csa,
  ctype*  b, inc_t rsb, inc_t csb,
  ctype*  beta, ctype* c, inc_t rsc, inc_t csc
  cntx_t* cntx,
  rntm_t* rntm /* Runtime object handle. */ );
```

Listing 1 Available APIs in BLIS for double precision GEMM.

```
/*** Parallelism-related functions. ***/

// Runtime object initialization.
bli_rntm_init( rntm_t rntm );

// Manually encode (maximum) ways of parallelism into runtime object.
void bli_rntm_set_ways(
  dim_t jc, dim_t pc, dim_t ic, dim_t jr, dim_t ir, rntm_t* rntm );

// Manually modify active threads.
void bli_rntm_set_active_ways(
  dim_t jc, dim_t pc, dim_t ic, dim_t jr, dim_t ir, rntm_t* rntm );
```

Listing 2 Available routines in BLIS for runtime parallelism management functions.

wasting computational resources) till all the active threads complete their work. Externally, a different application-level thread can asynchronously modify the run-time object at any time, including the active thread information –more specifically the recently added array that specifies the number of active

threads– of a given routine. As a consequence, this will vary the number of active threads that execute that routine.

From the user's perspective, an extra support routine is necessary to modify the amount and distribution of parallelism for a specific BLAS routine (independently of whether or not it is under execution), which is achieved by changing the internal field of a specific runtime object (`bli_rntm_set_active_ways`; see Listing 2).

Let us illustrate the approach with a practical example. Figure 3 provides an schematic overview of the internal work distribution in a typical scenario. Here, we propose an initial execution setup in which the maximum degree of parallelism is 8 (p_{max}), distributed as ($p1_{max} = 2$; $p4_{max} = 4$), where pX_{max} indicates the maximum degree of parallelism to be extracted from Loop X. In addition, the initial distribution of parallelism is set to ($p1_{curr} = 1$; $p4_{curr} = 2$); see diagram (a) in the top part of the figure. These initial conditions can be set by means of the API calls:

```
// Shared by threads.
rntm_t rntm;
bli_rntm_init( &rntm );

// Executed by any thread.
bli_set_ways( 2, 1, 1, 4, 1, &rntm );
bli_set_active_ways( 1, 1, 1, 2, 1, &rntm );

// Executed by any thread.
bli_dgemm( ..., &rntm );
```

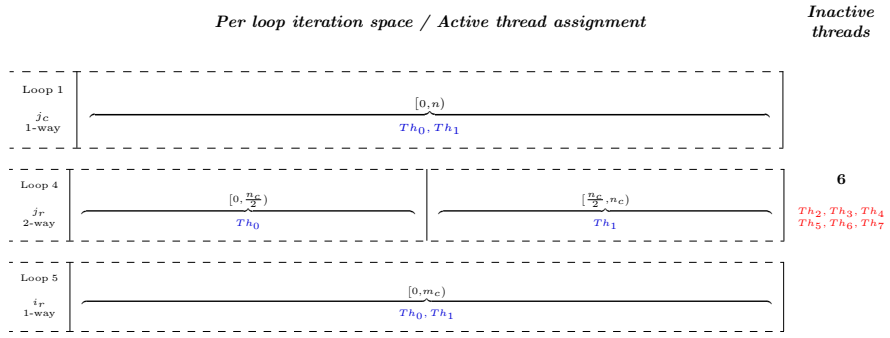
The bottom two diagrams in Figure 3 illustrate the effects of performing the following two modifications of the current degree of parallelism: (*i*) increasing parallelism at Loop 4 so that $p4_{curr} = 4$ (in diagram (b)); and (*ii*) increasing parallelism at Loop 1 so that $p1_{curr} = 2$ (in diagram (c)). These two variations are respectively attained via the execution (at any point or path of the same application,) of the following two API calls:

```
bli_set_active_ways( 1, 1, 1, 4, 1, &rntm );
bli_set_active_ways( 2, 1, 1, 4, 1, &rntm );
```

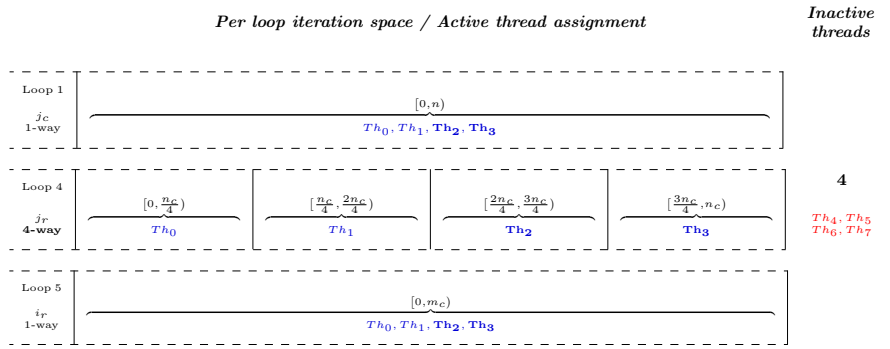
In all cases, p_{max} threads are spawned at the beginning, but only the *active* threads receive part of the workload (iterations of the corresponding loop) at each level.

4 Experimental results

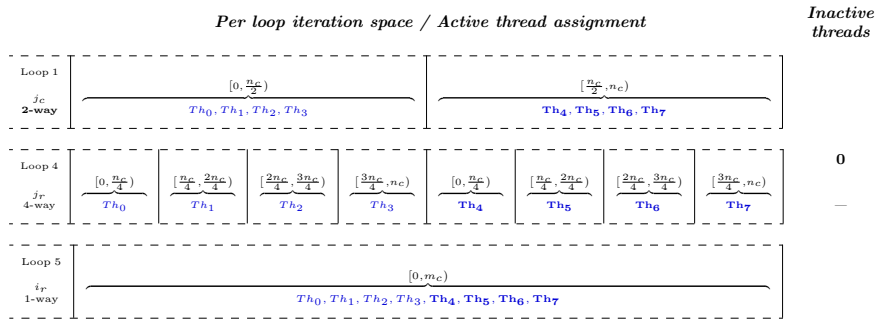
The experimental results reported in this section demonstrate the potential of malleability in a specific BLIS routine –GEMM–, and its impact when applied to a higher-level LAPACK routine, the LU factorization with partial pivoting.



(a) Initial iteration-to-thread assignment.



(b) Iteration-to-thread assignment after expanding parallelism to 4-way at Loop 4.



(c) Iteration-to-thread assignment after expanding parallelism to 2-way at Loop 1.

Fig. 3 Partitioning of the iteration space and assignment to threads/cores for a BLIS execution configured with a maximum amount of threads of 2 for Loop 1 and 4 for Loop 4. Threads in blue and red are considered active and inactive, respectively. For simplicity, Loops 2 and 3 are not included in the diagrams.

The experiments were carried out using IEEE double-precision on a server equipped with a 20-core Intel Xeon Gold 6138 (*Skylake* microarchitecture) running at a nominal frequency of 1.7 GHz. The reference codes were linked with BLIS v0.5.1 or with MKL 2018.1.163, and the malleability mechanism was integrated in BLIS v0.5.1. In all cases, parallelism is extracted (unless otherwise stated) only from Loop 3 of the BLIS kernels or, in the case of MKL, by setting the MKL_NUM_THREADS environment variable. In those experiments where the full socket is not used, the tests are carried out using the `taskset` command to simulate a socket with a smaller number of physical cores. Control in terms of thread-to-core affinity is delegated to the OS.

4.1 Evaluation of malleable BLIS: performance and overhead

The first experiment is designed to show the practical effects of malleability when integrated in the GEMM routine. Figure 4 represents a complete timeline of the execution of successive iterations of Loop 2 when malleability is applied to Loop 3. Specifically, we initially deploy two application-level threads. The first application-level thread executes a single GEMM of dimensions $m = n = k = 20,000$ and it is initially configured to proceed with eight active threads (the maximum number of threads is also set to eight). Every four seconds, the second application-level thread ciclycally varies the number of active threads in the GEMM call in the range $n_{th} = [4, 5, 6, 7, 8, 4, 5, \ldots]$.

The test starts with eight active threads, with an average execution time per iteration of Loop 2 of around $0.14s$; after the first reduction in the number of threads (down to 4), the execution time raises up to around $0.28s$. Similar successive modifications of the number of active threads yield the corresponding proportional increase/decrease in the execution times.

The results show that the response time of the malleability mechanism is not significant. This response time, however, strongly depends on the exact loop in which malleability is exploited: for the outermost loops, the implementation in terms of malleability is less responsive. In the end, the exact loop in which malleability should be applied is a trade-off between response time and overhead, as illustrated next.

The objective of the second round of experiments is to quantify the overhead introduced by the malleability mechanism. This extra cost depends on two variables, namely: *(i)* the specific loop at which malleability is exploited; and *(ii)* the loops in which the Level-3 BLAS routines check for changes in the number of active threads. A GEMM execution of dimensions $m = n = k$, ranging from $1,000$ to $20,000$ in steps of $1,000$, and eight physical cores is used to quantify this overhead; see Figure 5. The plots there displays the results in terms of GFLOPS (billions of double-precision floating-point operations per second). Two plots are included in the figure: the left-hand side plot reports the attained results when malleability is applied to Loop 3; the right-hand side plot offers equivalent results when the application is to Loop 4. Both plots include three configurations: *(i)* reference BLIS using 8 threads and no

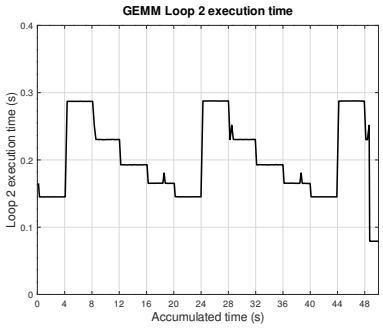


Fig. 4 Execution time (in seconds) of successive executions of Loop 2 for a malleable BLIS GEMM ($m = n = k = 20,480$), with changes in the number of threads every 4s.

malleability; *(ii)* malleable GEMM using 8 active threads, with a maximum of 10, and checking for changes in the number of active threads only in Loops 1 and 2; and *(iii)* malleable GEMM using 8 active threads, with a maximum of 10, and checking for changes in the number of active threads in all loops.

No remarkable overhead is observed when malleability is applied to Loop 3. In contrast, a visible additional cost appears when malleability is exploited at Loop 4. The source of this overhead is directly related to the number of times in which BLIS traverses loops, and hence the amount of synchronization points where unnecessary work is performed (as the threads are not spawn at loop-level, but at the routine invocation). If the malleability is set in Loop 3, the non-active threads are blocked at that level, and they do not iterate in the inner loops. In contrast, if it is set at Loop 4, the threads block at that point. This translates into a blocking phase per iteration of Loop 3 multiplied by the number of iterations of Loop 4. In addition, checking for changes in the number of active threads involves two extra synchronization points per test in order to ensure that all threads access the same value, increasing the overhead.

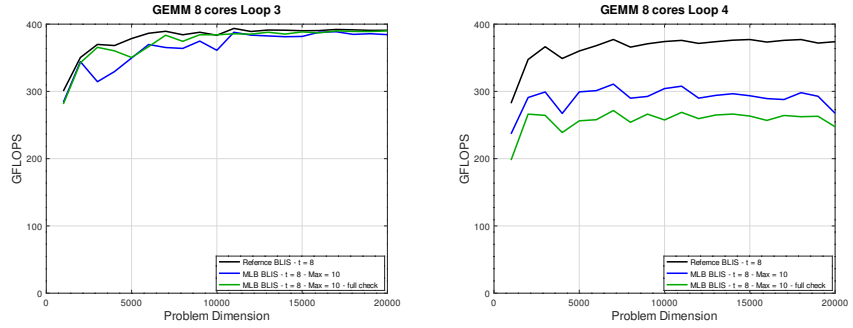


Fig. 5 Overhead introduced by the malleable version of BLIS, applying malleability on Loop 3 (left) and Loop 4 (right).

4.2 A Case Study: Application of malleability to the LU factorization

Many of the routines for dense matrix factorizations in LAPACK consist of a loop that processes the input matrix in steps of b columns/rows per iteration. At each step of the factorization, a certain block of columns (usually referred to as the *panel*) is factorized, while the trailing submatrix is updated with respect to the corresponding panel factorization. When the target platform is a multi-core processor, LAPACK simply relies on multi-threaded instances of the BLAS kernels. In this scenario, a well-known problem is that the panel factorization presents a reduced degree of parallelism. Thus, depending on the selected block size b and certain hardware parameters and organization of the target architecture, that particular kernel may easily constrain the global performance of the global factorization.

A strategy to tackle the hurdle of the panel factorization in a parallel execution consists in introducing a *look-ahead* technique into the algorithm [15]. Concretely, during each iteration of the decomposition, the look-ahead strategy aims to overlap the factorization of the "next" panel with the update of the "current" trailing submatrix, introducing two independent execution paths at the higher level, plus a synchronization point at the end of each iteration.

Keeping in mind these two execution paths, a common approach maps a few threads (p) to perform the next panel factorization (typically $p = 1$ *or* 2), while the remaining threads ($t - p$) are devoted to update the current trailing submatrix. At this point we note that the look-ahead technique can be easily implemented using the BLIS Typed Expert API, where the threading is expressed at the routine level. In comparison, older versions of the BLIS framework, or other BLAS instances, could only attain this effect via a much more complex strategy.

The experimental results in Figure 6 demonstrate the benefits of introducing look-ahead for small to mid-size problems, where the overall factorization execution time is dominated by the panel factorization. However, for large problems, no performance benefits are visible, and there is actually a performance degradation; compare the line labeled as "LU + BLIS" in the plots with those labeled as "LU LA X + Y", where X represents the threads in charge of the panel factorization and Y in charge of the trailing update. Note that the term "small to mid-size problems" depends on the total number of threads/physical cores in the target machine and on its distribution among the two execution paths: for example, in case there are 6 physical cores and two threads in charge of the panel factorization, this refers to problems of dimension ranging from 0 to 10,000 but, when there are 12 cores, this refers to problem dimensions ranging from 0 to 30,000. The reason for this behavior is that the workload of the two execution paths is *unbalanced*. For small to mid-size problems, the panel factorization (which is mostly sequential) takes longer time than the update of the trailing submatrix (which is processed in parallel, using the remaining threads), leading to an optimal load balance. On the other hand, for large problem dimensions, the trailing submatrix update dominates the time, although it is executed using all available threads except

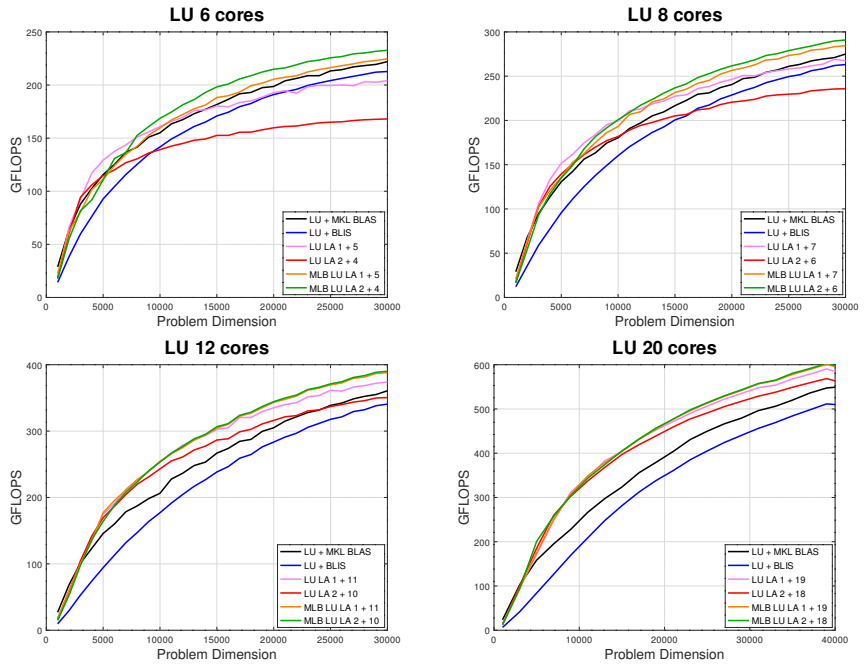


Fig. 6 Performance of the LU factorization with partial pivoting on 6 (top-left), 8 (top-right), 12 (bottom-left), and 20 (bottom-right) cores of an Intel Xeon Gold 6138 processor. The lines labelled as "LU + MKL BLAS" and "LU + BLIS" correspond to the conventional implementation of the factorization which simply relies on a multi-threaded instance of BLAS to extract parallelism. The line "LU LA X + Y" is for the LU factorization enhanced with look-ahead, with X threads in charge of the panel factorization and Y for the trailing update. Finally, "MLB LU LA X + Y" is the malleable variant of the latter.

those which participated in the panel factorization (that is, $t - p$,), yielding a performance degradation. The latter scenario is precisely the case of interest for a malleable Level-3 BLAS.

By leveraging the proposed malleable Level-3 BLAS, as soon as the panel factorization of a given iteration completes its execution, the application-level thread can modify the number of active threads updating the trailing submatrix. Thus, while the panel factorization is being computed, $t - p$ threads are updating the trailing submatrix; but when the former completes, t threads become responsible for performing the trailing update. This operation mode ensures that, at any time, a maximum of t threads are active, enhancing the performance of the factorization; see lines labeled as "MLB LU LA X + Y" in Figure 6. This lines are analogous to the lines labeled as "LU LA X + Y" with the difference that they leverage the malleability mechanism.

The plots also reveal that for the LU factorization with partial pivoting, and regardless of the number physical cores, the best thread mapping is to use two threads to perform the panel factorization. For clarity in the plots, lines

devoting more than two threads for the panel factorization are not depicted since they do not contribute to increase performance.

Figure 6 also reports results for the execution of the reference LU linked with Intel MKL. As seen, this option outperforms the results of the reference implementation linked with BLIS, but it is not able to surpass the performance of the Maleable Level-3 BLAS; see the line labelled as "LU + MKL BLAS".

5 Conclusions

In this paper, we have proposed a strategy to integrate *intra-routine malleability* into all level-3 routines by leveraging the BLIS Expert API. This enhancement allows a dynamic variation of the degree of parallelism extracted at each individual level (loop) within the BLIS framework by means of a simple API that can be invoked from the application invoking the BLIS kernels. The benefits of this approach are of wide appeal, especially in scenarios in which nested parallelism is extracted at two levels: application and library. Concretely, library malleability allows a perfect resource usage when application-level parallelism is not perfectly balanced. We illustrate the benefits of our approach through a practical example of such an application: the LU factorization enhanced with look-ahead; our approach attains remarkable performance gains with minor code modifications.

Considering future work, we believe that malleability can also offer significant advantadges for applications that combine inter-task and intra-task parallelism. These scenarios appear, for example, in runtime-based task scheduling (e.g., libflame's SuperMatrix [5] or popular task-based programming models such as StarPU [1] or OmpSs [9]). There, the scarce task-level parallelism in some parts of the execution [7] can be by-passed by means of dynamically increasing intra-task parallelism on demand. Hence, a fully malleable underlying library becomes mandatory.

Finally, the tradeoff between *simplicity* and *internal exposure* of current dense linear algebra APIs also deserves a discussion. The introduction of malleability into BLIS has been possible by leveraging specialized expert APIs that expose many of the internals of the framework so that users can tune its behaviour, on demand, at runtime. This type of APIs, however, are just a complement to simpler ones (e.g. legacy BLAS), useful for retro-compatibility and opaque use of the library. Modern and future BLAS-like APIs (e.g. Eigen [11] or the BLAS C++ API proposal [10]) do not provide this type of mechanisms, and hence malleability (or any other runtime-based modification, such as micro-kernel selection or on-demand mixed-precision) is harder to implement.

Acknowledgements The researchers from Universidad Complutense de Madrid were supported by the EU (FEDER) and Spanish MINECO (TIN2015-65277-R, RTI2018-093684-B-I00), and by Spanish CM (S2018/TCS-4423). The researcher from Universitat Politècnica de València was supported by the Spanish MINECO (TIN2017-82972-R).

References

1. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009 **23**, 187–198 (2011)
2. Catalán, S., Castelló, A., Igual, F.D., Rodríguez-Sánchez, R., Quintana-Ortí, E.S.: Programming parallel dense matrix factorizations with look-ahead and OpenMP. Cluster Computing (2019)
3. Catalán, S., Herrero, J.R., Quintana-Ortí, E.S., Rodríguez-Sánchez, R., Van De Geijn, R.: A case for malleable thread-level linear algebra libraries: The LU factorization with partial pivoting. IEEE Access **7**, 17617–17633 (2019)
4. Catalán, S., Igual, F.D., Mayo, R., Rodríguez-Sánchez, R., Quintana-Ortí, E.S.: Architecture-aware configuration and scheduling of matrix multiplication on asymmetric multicore processors. Cluster Computing **19**(3), 1037–1051 (2016)
5. Chan, E., Van Zee, F.G., Bientinesi, P., Quintana-Ortí, E.S., Quintana-Ortí, G., van de Geijn, R.: Supermatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 123–132. ACM, New York, NY, USA (2008)
6. Corporation, I.: Intel ® Math Kernel Library Developer Reference. Tech. rep., Intel Corporation (2019). URL https://software.intel.com/sites/default/files/mkl-2019-developer-reference-c_2.pdf
7. Dolz, M.F., Igual, F.D., Ludwig, T., Piñuel, L., Quintana-Ortí, E.S.: Balancing task- and data-level parallelism to improve performance and energy consumption of matrix computations on the intel xeon phi. Computers & Electrical Engineering **46**, 95–111 (2015)
8. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software **16**(1), 1–17 (1990)
9. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(2), 173–193 (2011)
10. Gates, M., Luszczek, P., Abdelfattah, A., Kurzak, J., Dongarra, J., Arturov, K., Cecka, C., Freitag, C.: C++ API for BLAS and LAPACK. Tech. Rep. 2, ICL-UT-17-03 (2017). Revision 02-21-2018
11. Guennebaud, G., Jacob, B., et al.: Eigen v3. http://eigen.tuxfamily.org (2019)
12. LAPACK project home page. http://www.netlib.org/lapack
13. Leung, J., Kelly, L., Anderson, J.H.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA (2004)
14. Smith, T.M., van de Geijn, R.A., Smelyanskiy, M., Hammond, J.R., Van Zee, F.G.: Anatomy of high-performance many-threaded matrix multiplication. In: 28th IEEE International Parallel & Distributed Processing Symposium (2014)
15. Strazdins, P.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia (1998)
16. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Computing **27**(1–2), 3–35 (2001)
17. Van Zee, F.G.: Implementing high-performance complex matrix multiplication via the 1m method. ACM Transactions on Mathematical Software Submitted
18. Van Zee, F.G., van de Geijn, R.A.: BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software **41**(3), 14:1–14:33 (2015)
19. Van Zee, F.G., Parikh, D.N., van de Geijn, R.A.: Supporting mixed-domain mixed-precision matrix multiplication within the BLIS framework. ACM Transactions on Mathematical Software Submitted
20. Van Zee, F.G., Smith, T.: Implementing high-performance complex matrix multiplication via the 3m and 4m methods. ACM Transactions on Mathematical Software **44**(1), 7:1–7:36 (2017)
21. Van Zee, F.G., Smith, T., Igual, F.D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T.M., Marker, B., Killough, L., van de Geijn, R.A.: The BLIS framework: Experiments in portability. ACM Transactions on Mathematical Software **42**(2), 12:1–12:19 (2016)